STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves

Yuan Xiao
The Ohio State University
xiao.465@osu.edu

Mengyuan Li
The Ohio State University
li.7533@osu.edu

Sanchuan Chen
The Ohio State University
chen.4825@osu.edu

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

ABSTRACT

Intel Software Guard Extension (SGX) offers software applications a shielded execution environment, dubbed *enclave*, to protect their confidentiality and integrity from malicious operating systems. As processors with this extended feature become commercially available, many new software applications are developed to enrich to the SGX-enabled software ecosystem. One important primitive for these applications is a secure communication channel between the enclave and a remote trusted party. The SSL/TLS protocol, which is the *de facto* standard for protecting transport-layer network communications, has been broadly regarded a natural choice for such purposes. However, in this paper, we show that the marriage between SGX and SSL may not be a smooth sailing.

Particularly, we consider a category of side-channel attacks against SSL/TLS implementations in secure enclaves, which we call the control-flow inference attacks. In these attacks, the malicious operating system kernel may perform a powerful *man-in-the-kernel* attack to collect execution traces of the enclave programs at the page level, the cacheline level, or the branch level, while positioning itself in the middle of the two communicating parties. At the center of our work is a *differential analysis framework*, dubbed STACCO, to dynamically analyze the SSL/TLS implementations and detect vulnerabilities—discernible execution traces—that can be exploited as decryption oracles. Surprisingly, in spite of the prevailing constant-time programming paradigm adopted by many cryptographic libraries, we found exploitable vulnerabilities in the latest versions of all the SSL/TLS libraries we have examined.

To validate the detected vulnerabilities, we developed a man-in-the-kernel adversary to demonstrate Bleichenbacher attacks against the latest OpenSSL library running in the SGX enclave (with the help of Graphene) and completely broke the `PreMasterSecret` encrypted by a 4096-bit RSA public key with only 57,286 queries. We also conducted CBC padding oracle attacks against the latest GnuTLS running in Graphene-SGX and an open-source SGX-implementation of mbedTLS (*i.e.*, mbedTLS-SGX) that runs directly inside the enclave, and showed that it only needs 48,388 and 25,717 queries, respectively, to break one block of AES ciphertext. Empirical evaluation suggests these man-in-the-kernel attacks can be completed within one or two hours.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10.
https://doi.org/10.1145/3133956.3134016

1 INTRODUCTION

Software applications' security heavily depends on the security of the underlying system software. In traditional computing environments, if the operating system is compromised, the security of the applications it supports is also compromised. Therefore, the trusted computing base (TCB) of software applications include not only the software itself but also the underlying system software and hardware.

To reduce the TCB of some applications that contain sensitive code and data, academic researchers have proposed many software systems to support *shielded execution—i.e.*, execution of a piece of code whose confidentiality and integrity is protected from an untrusted system software (*e.g.*, [26, 28, 30, 33, 37, 44, 45, 49, 56, 68, 74, 77]). Most of these systems adopted a hypervisor-based approach to protecting the memory of victim applications against attacks from malicious operating systems. Although promising, these academic prototypes have yet to see the light of real-world adoption. Not until the advent of Intel Software Guard eXtension (SGX) [2], a hardware extension available in the most recent Intel processors, did the concept of shielded execution become practical to real-world applications. SGX enforces both confidentiality and integrity of userspace programs by isolating regions of their memory space (*i.e.*, *enclaves*) from other software components, including the most privileged system software–no memory reads or writes can be performed inside the enclaves by any external software, regardless of its privilege level. As such, SGX greatly reduces the TCB of the shielded execution, enabling a wide range of applications [20, 36, 54, 59, 65, 76].

In typical application scenarios [20, 36, 76], shielded execution does not work completely alone; it communicates with remote trusted parties using secure channels, *e.g.*, SSL/TLS protocols. Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS), are transport-layer security protocols that provide secure communication channels using a set of cryptographic primitives. SSL/TLS protocols are expected, as part of their design goals, to prevent man-in-the-middle attackers who are capable of eavesdropping, intercepting, replaying, modifying and injecting network packets between the two communicating parties. Therefore, applications of Intel SGX [20, 36, 76] typically regard SSL/TLS modules

inside SGX enclaves as basic security primitives to establish end-to-end communication security.

Attacks against the SSL/TLS protocol have been reported over the years, unfortunately. One important category of these attacks is oracle attacks [31]. In an oracle attack, the adversary interactively and adaptively queries a vulnerable SSL/TLS implementation and uses the response (or some side-channel information, *e.g.*, the latency of the response) as an oracle to break the encryption. Well-known examples of oracle attacks include the Lucky Thirteen Attack [12], the Bleichenbacher attack [22], the DROWN attack [18], the POODLE attack [52], *etc.* Prior demonstration of these attacks have shown that they enable network attackers to decrypt arbitrary messages of the SSL record protocol or decrypt the *PreMasterSecret* of the SSL handshake protocol. We will detail these attacks in Section 2. Due to the broad adoption of the SSL/TLS protocol (*e.g.*, in HTTPS, secure email exchanges), any of these attacks is devastating and easily headlines of the security news (*e.g.*, [41]). Accordingly, the SSL/TLS protocol and its implementations have been frequently updated after the publicity of these attacks. A commonly used solution is to *hide* the oracles. For example, in cases where the oracle is the SSL Alert message indicating padding errors, the error message can be unified to conceal the real reason for the errors [35, 58] (so that the adversary cannot differentiate padding errors and MAC errors, see Section 2). As of today, almost all widely used SSL/TLS implementations are resilient to oracle attacks because the oracles have been successfully hidden from the network attackers [4, 10, 35, 58].

However, adoption of SSL/TLS in SGX enclaves brings new security challenges. Although SGX offers confidentiality protection, through memory isolation and encryption, to code and data inside secure enclaves, it has been shown vulnerable to side-channel attacks [43, 63, 73]. Side-channel attacks are a type of security attacks against the confidentiality of a system or application by making inferences from measurements of observable side-channel events. These attacks have been studied in the past twenty years in multiple contexts, most noticeably in desktop computers, cloud servers, and mobile devices where CPU micro-architectures [78, 79], software data structures [40, 57], or other system resources are shared between mutually-distrusting software components. What makes side-channel attacks on SGX different is that these attacks can be performed by the privileged system software, which enables many new attack vectors. For example, Xu *et al.* [73] demonstrated that by manipulating page table entries of the memory pages of secure enclaves, an adversary with system privilege could enforce page faults during the execution of enclave programs, thus collecting traces of memory accesses at the page-granularity. Recently, Lee *et al.* [43] demonstrated that the control flow of enclave programs can be precisely traced at every branch instruction by exploiting the shared Branch Prediction Units (BPU).

The *key insight* of this paper is that while SSL/TLS is designed to defend against *man-in-the-middle* attacks, its implementation in SGX enclaves must tackle a stronger *man-in-the-kernel* adversary who is capable of not only positioning himself in the *middle* of the two communicating parties, but controlling the underlying operating system kernel and manipulating system resources to collect execution traces of the enclave programs from various side channels. Particularly, we show that the powerful man-in-the-kernel attackers can create new decryption oracles from the state-of-the-art SSL/TLS implementations and resurrect the Bleichenbacher attack and CBC padding oracle attacks against SGX enclaves.

Stacco. At the core of our work is the Side-channel Trace Analyzer for finding Chosen-Ciphertext Oracles (Stacco), which is a software framework for conducting differential analysis on the SSL/TLS implementations to detect *sensitive control-flow vulnerabilities* that can be exploited to create decryption oracles for CBC padding oracle attacks and Bleichenbacher attacks. Particularly, to enable automated large-scale analysis of various off-the-shelf SSL/TLS libraries, we built Stacco on top of a dynamic instrumentation engine (*i.e.*, Pin [46]) and an open-source SSL/TLS packet generation tool (*i.e.*, TLS-Attacker [66]), so that we can perform standard tests to multiple libraries in an automated manner. To understand the exploitability of the vulnerabilities, we also modeled three types of control-flow inference attacks, including page-level attacks [63, 73], cacheline-level attacks [23, 60] and branch-level attacks [43], and empowered Stacco to analyze vulnerabilities on each of these levels. Our analysis results suggest all the popular open-source SSL/TLS libraries we have examined are vulnerable to both types of oracle attacks, raising the questions of secure development and deployment of SSL/TLS protocols inside SGX enclaves.

To validate the vulnerabilities identified by Stacco, we demonstrated several such *man-in-the-kernel* attacks against the latest versions of popular cryptographic libraries: Particularly, we implemented a Bleichenbarcher attack against the latest OpenSSL library [9] running in the SGX enclaves (with the help of Graphene-SGX [70], a library OS that supports unmodified applications to run inside SGX enclaves) and completely broke the `PreMasterSecret` encrypted by a 4096-bit RSA public key with only 57,286 queries. We also conducted CBC padding oracle attacks against the latest GnuTLS [3] running in Graphene-SGX and an open-source SGX-implementation of mbedTLS [8] that runs directly inside the enclave, and showed that it only needs 48,388 and 25,717 queries, respectively, to break one block of AES ciphertext from TLS connections using these libraries. Empirical evaluation suggests these man-in-the-kernel attacks can be completed within one or two hours. These demonstrated attacks not only provide evidence that Stacco can effectively identify exploitable sensitive control-flow vulnerabilities in SSL/TLS implementations, but also suggest these oracle attacks conducted in a man-in-the-kernel manner are efficient for practical security intrusion.

Responsible disclosure. We have reported the vulnerabilities and demonstrated oracle attacks to Intel, OpenSSL, GnuTLS, mbedTLS.

Contributions of this work include:

- The first study of critical side-channel threats against SSL/TLS implementations in SGX enclaves that lead to complete compromises of SSL/TLS-protected secure communications.
- The design and implementation of Stacco, a differential analysis framework for detecting sensitive control-flow vulnerabilities in SSL/TLS implementations, which also entails:
- A systematic characterization of control-flow inference attacks against SGX enclaves (*e.g.*, page-level attacks, the cacheline-level attacks, and branch-level attacks), which empowers Stacco to analyze the vulnerability with abstracted attacker models.

- A measurement study of the latest versions of popular SSL/TLS libraries using STACCO that shows that all of them, including OpenSSL, GnuTLS, mbedTLS, WolfSSL, and LibreSSL, are vulnerable to control-flow inference attacks and exploitable in oracle attacks.
- An empirical *man-in-the-kernel* demonstration of oracle attacks against the latest version of OpenSSL and GnuTLS running inside Graphene-SGX and an open-source SGX-implementation of mbedTLS, showing that such attacks are highly efficient on *real* SGX hardware.

Roadmap. The rest of this paper is outlined as follows. Section 2 introduces related background concepts. Section 3 systematically characterizes control-flow inference attacks. Section 4 describes a differential analysis framework for detecting sensitive control-flow vulnerabilities in SSL/TLS implementations. We demonstrate oracle attacks against some of the vulnerable SSL/TLS implementations to validate these detected vulnerabilities in Section 5, and then discuss countermeasures in Section 6. In Section 7, we briefly summarize related work in the field. Section 8 concludes our paper.

2 BACKGROUND

2.1 Intel Software Guard Extension

Intel SGX is a new processor architecture extension that is available on the most recent Intel processors (*e.g.*, Skylake and later processor families). It aims to protect the confidentiality and integrity of code and data of sensitive applications against malicious system software [2]. The protection is achieved through a set of security primitives, such as memory isolation and encryption, sealed storage, remote attestation, *etc.* In this section, we briefly introduce some of the key features of Intel SGX that is relevant to this paper.

Memory isolation and encryption. Intel SGX reserves a range of continuous physical memory exclusively for enclaves. This memory range is called Enclave Page Cache (EPC), which is a subset of Processor Reserved Memory (PRM). The EPC is managed in similar ways to regular physical memory and is divided into 4KB pages. Correspondingly, a range of virtual addresses, called Enclave Linear Address Range (ELRANGE), is reserved in the virtual address space of the applications. The page tables responsible for address translation are managed by the untrusted operating system. Therefore, the mapping of each virtual memory page to the physical memory, access permissions, cacheability, *etc.*, can be controlled by the system software that is potentially malicious. To maintain the integrity of the page tables, the memory access permission dictated by the developers are recorded, upon enclave initiation, in Enclave Page Cache Map (EPCM), which is also part of PRM (thus protected from the malicious system software). During the address translation, EPCM is consulted to enforce access permission by bitwise-AND the set of permissions in the EPCM entries and those in the page table entries.

The memory management unit (MMU) enforces integrity and confidentiality of EPC pages. Only code running in enclave mode can access virtual memory pages in the ELRANGE that are mapped to the EPC. Each EPC page has at most one owner at a time, and the EPCM serves as a revert page table that records virtual address space of each enclave that maps to the corresponding EPC page. An EPC page can be evicted and stored in the regular physical memory region. Evicted EPC pages are encrypted by Memory Encryption Engine (MEE) to guarantee their confidentiality.

2.2 SSL/TLS

Secure Sockets Layer (SSL) is a general purpose security protocol proposed by Netscape Communications in 1994, which was designed to transparently protect the confidentiality and integrity of the network communications between applications running on top of the TCP layer. Due to security flaws, SSL v1.0 was never released to the public. SSL v2.0 was released in 1995 and deprecated in 2011; SSL v3.0 was released in 1996 and deprecated in 2015 (after the publicity of POODLE attacks [52]). Its successor protocols, Transport Layer Security (TLS) v1.0, v1.1, and v1.2, were released in 1999, 2006 and 2008, respectively. They are all still being used broadly. TLS v1.3 is still a working draft as of August 2017. SSL and TLS are referred together as the SSL/TLS protocol. SSL/TLS has two sub-protocols, the *handshake protocol*, and the *record layer protocol*. The handshake protocol negotiates security primitives (ciphers, their parameters, and cryptographic keys), and the record layer protocol uses the negotiated security primitives for encryption and authentication of the payload data, such as HTTP, IMAP, SMTP, POP3, *etc.*

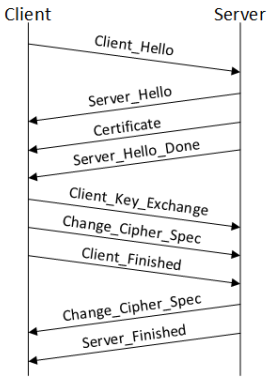

Figure 1: The SSL handshake protocol.

Handshake protocol. The SSL handshake protocol allows the communicating server and client to authenticate each other and negotiate an algorithm for message encryption and integrity protection. The protocol is illustrated in Figure 1. The client initiates the SSL connection with a `ClientHello` message, which tells the server the maximum SSL version it supports, a 28-byte random value, the identifier of the SSL session that this current SSL connection is associated to, the set of supported ciphers, and the compression algorithms. The server, upon receiving the client's request, responds with a `ServerHello` message, with the same set of information from the server. The server will then send a `Certificate` message, if it is the first connection of the session, to offer its certificate to the client. If the certificate used by the server is a certificate that uses Digital Signature Algorithms (DSA) or a signing-only RSA certificate, it does not have a key that can be used for encryption purposes. In this case, the server will send a `ServerKeyExchange` message to inform the client its Diffie-Hellman (DH) parameters to perform the key exchange.

Upon receiving the `ServerHelloDone` message from the server, the client sends a `ClientKeyExchange` message to the server. If RSA key exchanges are used, the `PreMasterSecret` will be encrypted using the RSA public key embedded in the certificate and sent along with the message; if Diffie-Hellman key exchange algorithms are

used, this message will only include the client's DH parameters—the `PreMasterSecret` is calculated by the server and client respectively without being sent over the network. After this step, the server and the client already share the secrets for generating the symmetric encryption keys and Message Authentication Code (MAC) keys. The `ChangeCipherSpec` messages from the client and the server notify the other party about the forthcoming changes to the cipher algorithms that have just been negotiated.

Particularly, when RSA-based key exchange method is selected, the `PreMasterSecret` is encrypted using the server's public RSA key. The format of the plaintext message of `ClientKeyExchange` conforms to a variant of PKCS#1 v1.5 format (shown in Figure 3): it must start with $0x0002$ which is followed by 205 bytes of non-zero paddings provided that the total message is 256-byte long (determined by the size of the server's private key). Then a $0x00$ byte following the padding is regarded as the segmentation mark, and the 48-byte `PreMasterSecret` is attached at the end. According to RFC5246 (TLS v1.2), in order to defeat Bleichenbacher Attacks, which we will detail shortly, the server first generates a random value, and then decrypts the `ClientKeyExchange` message. If the decrypted data does not conform to the PKCS#1 standard or the length of the `PreMasterSecret` is incorrect, the random value will be used for the rest of the computation, as if the decryption was successful.

TLS v1.0, v1.1, and v1.2 support a variety of cipher suites. For example, `TLS_RSA_WITH_AES_128_CBC_SHA` is one of the cipher suites which employs RSA for both authentication and key exchange, the symmetric encryption uses the AES block cipher in Cipher Block Chaining (CBC) mode, and SHA-1 based HMAC is used for integrity protection of the payload. Other key exchange algorithms can also be specified. For instance, `TLS_ECDHE_ECDSA` uses elliptic curve Diffie-Hellman key exchange and Elliptic Curve Digital Signature Algorithm for authentication.

Record layer protocol. The record protocol of TLS protects the confidentiality and integrity of the payload via symmetric encryption and MAC algorithms. Block encryption in the Cipher Block Chaining Mode (CBC) is one of the most widely used modes of operation for block ciphers. The encryption and authentication is conducted in the *MAC-pad-encrypt* scheme, as shown in Figure 2. The MAC of the data payload is first calculated to protect its integrity, and then the resulting data is padded with dummy bytes (conforming to SSL/TLS specifications) so that the total message size is multiples of the block size (*e.g.*, 16 bytes in AES). The resulting data blocks are then encrypted using the symmetric cipher in the CBC mode.

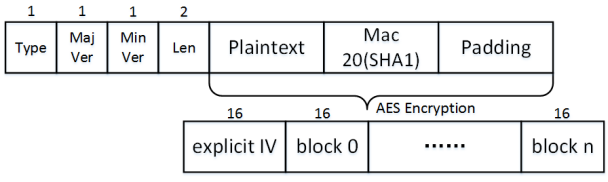

Figure 2: Packet format of SSL/TLS payload encryption.

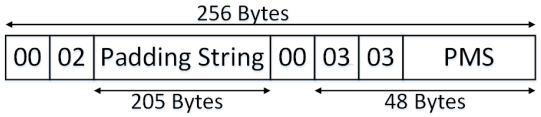

Figure 3: Format of the plaintext of the `ClientKeyExchange` message (with 2048-bit RSA keys).

2.3 Bleichenbacher Attacks against SSL/TLS

Bleichenbacher attacks [22] is the first practical adaptive chosen-ciphertext attack against RSA cryptographic algorithms conforming to the PKCS#1 v1.5 encoding schemes. It exploits the format correctness of the decrypted plaintext as an oracle and decrypts, by repeatedly querying the oracle about the correctness of carefully-crafted ciphertexts, an RSA public-key-encrypted message without the need of the RSA private keys. Multiple studies have shown that Bleichenbacher Attacks have practical implication in network security [19, 22, 42, 47]. Particularly, these attacks have been demonstrated to work against SSL/TLS protocols that adopt RSA algorithms to encrypt the `PreMasterSecrets` and at the same time reveal non-conformant error messages over the network. Most widely used SSL/TLS implementations today are believed to be immune to Bleichenbacher Attacks as the oracle-enabling error messages have been suppressed. We have summarized a brief history of the related studies in Section 7.

In this paper, we implemented the optimized Bleichenbacher attack proposed by Bardou *et al.* [19]. The attack relies on the artifact that a correctly formatted `ClientKeyExchange` message, before encryption, must begin with $0x0002$. Therefore, its value m must satisfy $2B \leq m < 3B$, where B is a constant that starts with $0x0001$ and ends with a running of $8(k-2)$ of 0s (k is the length of m in bytes). By repeatedly querying the oracle and finding a sequence of s_i so that each m_i ($c_i \equiv c \cdot (s_i)^e \bmod n, m_i \equiv (c_i)^d \bmod n$ and $m \equiv c^d \bmod n$) is also PKCS conformant, the adversary can gradually narrow down the possible value ranges for m until only one possibility remains. Interested readers can refer to Bardou *et al.* [19] for details of the algorithm.

The oracle strength is defined as the conditional probability of the oracle returning true given the decrypted plaintext message indeed begins with $0x0002$. A strong oracle means the adversary can complete the attack with fewer queries. This is because the probability that a query with message m beginning with $0x0002$ is returned by the oracle as false is smaller; hence the adversary can collect a sequence of such messages faster. For example, if the oracle always returns true when the plaintext message starts with $0x0002$, the oracle strength is 1. In contrast, if the oracle returns true if and only if the following three conditions hold at the same time, for instance, (1) the first two bytes of the plaintext are $0x0002$, (2) the next 8 bytes do not contain $0x00$, and (3) the next 246 bytes that follow (in the case of a 2048-bit RSA encryption) do have at least one byte of $0x00$, the oracle strength is

$$(\frac{255}{256})^{8} \times (1 - (\frac{255}{256})^{246}) \approx 0.599$$

.

2.4 CBC Padding Oracle Attacks

It is known that the *MAC-pad-encrypt* used in SSL/TLS is vulnerable to padding oracle attacks [12, 13, 25, 52, 71]. The vulnerability can be exploited when the CBC mode of operation is chosen. In such attacks, a carefully-crafted *application data* packet is repeatedly sent by the man-in-the-middle attacker to the vulnerable SSL/TLS server/client (collectively called the SSL agent). Each time the message is sent to the victim SSL agent for decryption, the attacker modifies the ciphertext slightly to conduct an *adaptively chosen ciphertext attack*. The SSL agent checks the correctness of the padding and the MAC after decrypting the message. If there are errors in the format of the padding or content of the MAC, an alert message will be returned. If the attacker can tell if the error message is caused by only the padding error or by both padding errors and MAC errors, she has a padding oracle that tells her whether her modification of the ciphertext is decrypted into a correct padding (very unlikely to have a correct MAC). In SSL v3.0, the padding format only specifies the last padding byte has the value of the total padding length, while the other bytes could have random values. In TLS specifications, all padding bytes have the same value, which is the number of the padding bytes. Therefore, a correct padding reveals the content of the last byte of the plaintext, which gives the attacker the power to decrypt some data without having the decryption key.

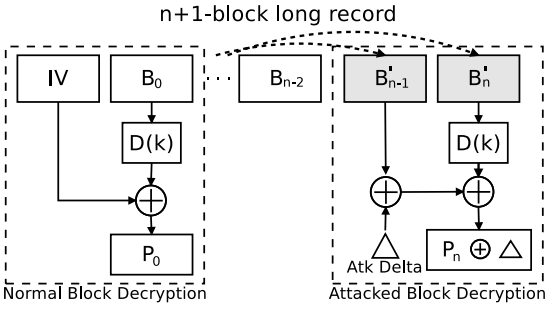

Figure 4: Illustration of CBC padding oracle attacks.

More specifically, as shown in Figure 4, the attacker can send a ciphertext of $n+1$ blocks to the victim, with the last two blocks replaced by the two blocks of interest, *e.g.*, B'_{n-1} and B'_n in the figure. Let's denote the plaintext of B'_n as P'_n, which is the value the attacker hopes to learn. Therefore, $P'_n = B'_{n-1} \oplus D_k(B'_n)$, where $D_k()$ is the decryption function of a block under the secret key, k. The attack proceeds from the bytes on the right in the block and gradually extends to bytes on the left: To decrypt the last two bytes of P'_n, the attacker, at each time she queries the padding oracle, XORs the last two bytes of B'_{n-1} with a value Δ, which has a value ranging from $0x0000$ to $0xFFFF$. Then the resulting plaintext is $B'_{n-1} \oplus \Delta \oplus D(B'_n) = (B'_{n-1} \oplus D(B'_n)) \oplus \Delta = P'_n \oplus \Delta$. When enumerating values of Δ from $0x0000$ to $0xFFFF$, one of the values will lead to a correct padding (*i.e.*, $0x0101$ as the last two bytes). Therefore, the plaintext of the last two bytes of P'_n is simply $0x0101 \oplus \Delta$. The attack continues to guess the other bytes on the right by altering the value of Δ and looking for a correct padding of $0x020202$, $0x03030303$, *etc.*

The key to the success of such attacks is the ability to differentiate the cases where a MAC error and a padding error occur at the same time from the cases where only the padding error happens. When the error type is not reported, which is the case in all SSL/TLS implementations after the publication of the original padding oracle attacks in 2002 [71], the CBC padding oracle attacks becomes very difficult. The recently published variants of the attack worked around this defense to re-enable the oracles: In Lucky Thirteen attacks [12] and Lucky Microseconds attacks [13], a remote timing-channel orcale was used to differentiate the two types of errors. In POODLE attacks [52], the oracle is created by eliminating the MAC errors when the plaintext is carefully crafted so that the length of the padding is exactly one block (*i.e.*, 16 bytes for AES). As such, only padding errors may occur, which can be exploited as an oracle. This attack only works for SSL v3 because a correct padding only requires the last byte to be 16, while other padding bytes are not specified. In contrast, because the TLS protocols specify the content of every padding byte to be the length of the padding, it is unlikely to decrypt an arbitrary ciphertext into the correct padding. So the attack won't work with TLS protocols. In this paper, the oracle is reconstructed using a new type of side channels and the attacks work on all SSL/TLS versions and implementations.

3 THREAT MODEL ANALYSIS

To analyze the security threats on Intel SGX imposed by side-channel attacks, in this paper, we systematically study one important category of side-channel attacks—control-flow inference attacks. In these attacks, the goal is to infer, by measuring side-channel observations, the indirect control-flow transfers of the enclave program and thereby learning sensitive information that is shielded by SGX.

Inferring program control flows and learning sensitive information are two separate steps. On one hand, the existence of side-channel attack vectors, *e.g.*, page-fault traces, cacheline access traces, branch instruction traces, *etc.*, enables control-flow inference. We name such attacks control-flow inference attacks. These attacks have been studied in previous studies. Here in this paper, we propose a systematic approach to model control-flow inference attacks, which enables discussion of these attacks *without specifying the exact attack techniques*. On the other hand, control-flow leakage does not always lead to a security breach. Only code with secret-dependent control flows is vulnerable to control-flow inference attacks.

3.1 Control-Flow Inference Attacks

In previous work, it has been shown that SGX enclaves are vulnerable to a variety of side-channel attacks. Some manipulate page table entries, some control shared caches, and others exploit shared branch prediction units. The methods to collect side-channel observations are also diverse: Some attacks use hardware timestamp counters to measure the execution time of specific code, some use Last Branch Record (LBR) to measure elapsed cycles between branch instructions, and some rely on deterministic events (*e.g.*, page faults). Regardless of the attack techniques, we categorize control-flow inference attacks into three levels: page-level attacks,

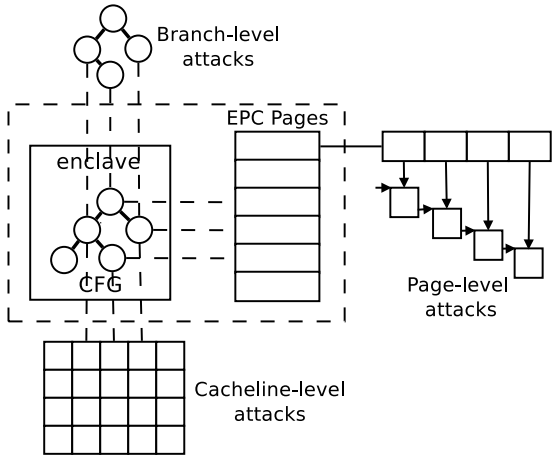

Figure 5: Three categories of control-flow inference attacks.

cacheline-level attacks, and branch-level attacks. We illustrate these three levels in Figure 5.

Page-level attacks. It was reported by Xu *et al.* [73] that by clearing the *Present* bit of the page table entries, the adversary controlling the OS can force the EPC page accesses by the enclave program to raise page fault exceptions and be trapped into the OS kernel controlled by the adversary. In this way, the adversary could observe the enclave program's page-level memory access pattern. In our own exploration, we found not only the *Present* bit, other bits in the page table entry, such as *Reserved* bits, *NX* bit, *etc.*, as well as Translation-Lookaside Buffers (TLB) and paging-structure caches [1] also enable similar attack semantics. In this paper, we model the side-channel observations collected in page-level attacks as a sequence of page faults: $< P_1, P_2, P_3, \cdots, P_n >$, where P_i is the virtual page frame number of the enclave program. With known binary code of the enclave programs, P_i maps to a specific page of executable code of the enclave program.

Cacheline-level attacks. Intel SGX does not prevent cache-based side-channel attacks. Therefore, most prior work on cache-based side-channel attacks is, in theory, applicable to SGX enclaves. While it is challenging to model every single attack technique that has been explored in previous studies, we abstractly model cache-based side-channel attacks as a sequence of observations of the victim enclave program's cacheline accesses: $< C_1, C_2, C_3, \cdots, C_n >$, where C_i is the virtual address of the beginning of the cacheline (*i.e.*, cacheline sized and aligned memory block). With known binary code of the enclave programs, C_i maps to a specific cacheline-sized block of executable code of the enclave program.

Branch-level attacks. Very recently, Lee *et al.* [43] demonstrated that the control flow of enclave programs can be precisely traced at every branch instruction because the Branch Prediction Units (BPU) inside the CPU core is not flushed upon Asynchronous Enclave Exit (AEX). Therefore, a powerful adversary could interrupt the enclave execution, which triggers an AEX, and then execute a piece of shadow code whose virtual addresses are the same as the victim code in the lower 32-bit range—so that they are mapped to the same entries in the Branch Target Buffer (BTB). The adversary employs LBR to learn whether each branch of the shadow code is correctly predicted or not, which apparently is influenced by the branch history of the enclave program that is just interrupted. To model these attacks, or other powerful attacks that are yet to be discovered, we consider the strongest control-flow inference attacks as a sequence of basic blocks that are executed in order: $< B_1, B_2, B_3, \cdots, B_n >$, where B_i is a basic block in the enclave program's control-flow graph (CFG).

3.2 Sensitive Control-Flow Vulnerabilities

If the enclave program has secret-dependent control flows, then it is potentially vulnerable to control-flow inference attacks. Such vulnerabilities are named sensitive control-flow vulnerabilities in this paper. In this work, our focus is one of the most critical applications for SGX enclave—SSL/TLS libraries. Although SSL/TLS libraries, *e.g.*, OpenSSL, are implemented in a way that constant-time execution is enforced, however, as we will show in this work, they still have sensitive control-flow vulnerabilities due to improper error handling and reporting, thus are vulnerable to control-flow inference attacks.

4 DETECTING SSL/TLS VULNERABILITIES WITH STACCO

In this section, we present the Side-channel Trace Analyzer for finding Chosen-Ciphertext Oracles (Stacco), a differential analysis framework for detecting sensitive control-flow vulnerabilities in SSL/TLS implementations under the threat model we laid out in Section 3. The core idea behind the framework is that when provided with encrypted SSL/TLS packets with non-conformant formats or incorrect paddings with different types of errors, the decryption code may exhibit different control flows that give rise to the decryption chosen-ciphertext oracles. To enable automated tests for *multiple* oracle vulnerabilities on *various* SSL/TLS implementations under *different* attack models, *i.e.*, page-level, cacheline-level and branch-level control-flow inference attacks, we developed a differential analysis framework (Section 4.1) and used it to evaluate 5 popular SSL/TLS libraries (Section 4.2).

4.1 Differential Analysis Framework

At the center of our differential analysis framework is a dynamic instrumentation engine to collect execution traces of the SSL/TLS implementation. The overall architecture of our framework is shown in Figure 6. Our framework consists of five components: a packet generator (*i.e.*, the TLS-attacker in the figure), an SSL/TLS program linked to an SSL/TLS library under examination, a trace recorder (*i.e.*, Pin), a trace *diff* tool, and a vulnerability analyzer.

A complete run of one differential analysis test follows *three* main steps. The *first* step is to collect two execution traces. The packet generator generates two SSL/TLS packets following specific rules (to be explained in Section 4.2) and sends them to the SSL/TLS program. The program which is linked to the library being analyzed runs on top of the Pin-based trace recorder, where the execution traces of the analyzed library are collected. The *second* step is to compare the two execution traces. Differences in the traces indicate potential sensitive control-flow vulnerabilities. The *final* step is to

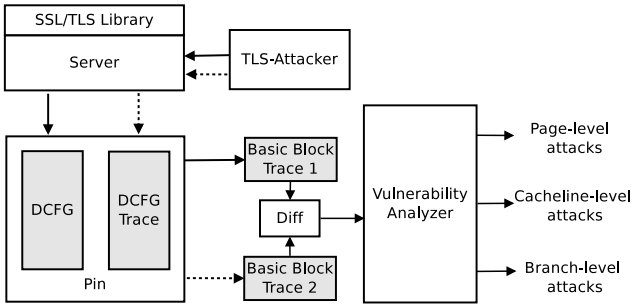

Figure 6: Architecture of the differential analysis framework.

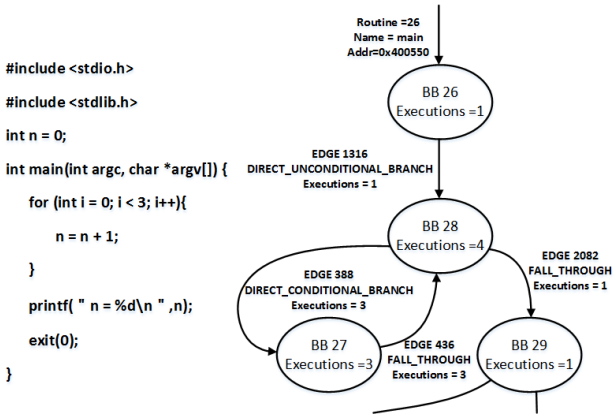

Figure 7: An example of DCFG.

decide whether the differences in the traces are exploitable by the attacker. Given a specific attack model, *e.g.*, page-level, cacheline-level, and branch-level control-flow inference attacks, the vulnerability analyzer is able to tell whether the tested library is vulnerable to such attacks and, if so, pinpoints the exploitable vulnerabilities.

Packet generator. The packet generator is in charge of generating the input to the framework. It prepares encrypted packets with specified plaintext or ciphertext (with specified errors) to be sent to the peer at any stage of an SSL/TLS connection. In our implementation, we adopted an open-source tool, TLS-attacker [67]. It is able to complete an SSL/TLS handshake or replace any packet in this process. It is also able to send arbitrary data records after the SSL/TLS connection has been successfully established.

Trace recorder. A core component of the framework is the execution trace recorder. We implemented the trace recorder on top of Intel Pin. Pin [46] is a dynamic binary instrumentation framework that is suitable for a range of program analysis tasks. It enables various tools, called *Pintools*, to be developed using the framework. Of interest to our purpose is its capability of dynamic instrumenting a software program *without changing its memory layout*, which is essential for detecting sensitive control-flow vulnerabilities. Particularly, a Pintool provided by Pinplay kit [55] can be used to create the Dynamic Control-Flow graph (DCFG) of a program [75].

DCFG is defined by Intel as an extension of the control-flow graphs (CFG) [14]. An example of a DCFG can be found in Figure 7. Generally speaking, a DCFG shows the portion of a CFG that has been executed. An edge in a DCFG is augmented with a counter, which records the number of times this edge is executed. Pinplay kit also provides an option to record the exact sequence of the executed edges in the DCFG, which is called DCFG-Trace. Combining the DCFG with the DCFG-Trace, Stacco is able to generate a trace of basic blocks that has been executed by the instrumented programs.

In order to improve the runtime performance and to facilitate data analysis, we need to specify which parts of the execution we are most interested in. For example, if we are looking for vulnerabilities in the handshake protocol, the execution trace should be recorded only when the handshake APIs are called. However, we found that such a selective tracing functionality that could have been enabled by control options `enter_func` and `exit_func` in the Pintool could not work properly with the SSL/TLS libraries. As a solution, we added in the SSL/TLS program two empty functions `foo()` and `bar()` to wrap the functions that we are interested in, by adding a call to `foo()` before it and `bar()` after it. Thus we could control the Pintool to selectively trace functions when the option `-log:control start:address:foo,stop:address:bar` is enabled.

The output of the trace recorder consists of two `JSON` files. One includes the DCFG as well as basic information such as base addresses of the libraries and the offsets of each of the basic blocks in the libraries. The other `JSON` file contains a trace of DCFG edges. We then extended the Pintool using Pin DCFG APIs to merge the two files into a trace of basic blocks. Because the base addresses of the libraries change every time we run the program, we use the name of the library and the offset of the basic block to uniquely identify a basic block.

It might be worthwhile noting here, that Pin has bugs when executing certain functions (*e.g.*, `gnutls_record_recv()` in GnuTLS) and, as a result, the DCFG traces of these functions cannot be correctly recorded. For this special case, we replace the Pin-based trace recorder with a Callgrind-based trace recorder. Callgrind is a tool of Valgrind [53, 72] which is also a dynamic instrumentation framework. We extend Callgrind to include timestamps for each function call in order to recover the complete call trace from the call graph generated by Callgrind. Unfortunately, Callgrind does not provide fine-grained basic block tracing (*e.g.*, DCFGs) like Pin.

The diff tools. Because the Pin-generated basic-block traces are ordered sequences of basic blocks, the *diff* command of Linux OS turns out to be enough to identify the differences between the two traces. In contrast, the Callgrind-generated function call traces are less structured. We, therefore, implemented a Python tool based on *difflib* to compare Callgrind-generated function call traces, which first converts the call traces to call trees with nodes associated with timestamps, and then represent the call trees using XML, which can be compared using *difflib*.

Vulnerability analyzer. Given the results of the *diff* tools, we build a vulnerability analyzer in Python to examine sensitive control-flow vulnerabilities. Particularly, the differences in the basic-block traces are by themselves vulnerability to branch-level attacks. To detect vulnerabilities at the cacheline level or the page level, we

need to convert the basic-block traces into traces of cachelines and pages. Specifically, the virtual address of the beginning of each basic block is calculated. The corresponding page trace can be obtained by dividing the virtual addresses of the basic blocks by 4096, the size of a memory page, and then merge consecutive basic blocks together if they have the same page address. Similarly, cacheline traces can be generated by dividing virtual addresses of the basic blocks by 64, which is the size of a cacheline. After the conversion, if the trace differences in the basic-block sequences render the same cacheline trace or page trace, the program, as per this test, is not vulnerable to cacheline-level or page-level control-flow inference attacks.

4.2 Evaluation and Results

We applied Stacco to detect two types of oracle attacks, CBC padding oracle attacks, and Bleichenbacher attacks, in the latest versions (as of February 2017) of five popular open-source libraries (see Table 1)[1].

4.2.1 Bleichenbacher Tests. To detect vulnerabilities that enable Bleichenbacher attacks, we conducted a series of differential tests. In each of the tests, we differentially analyzed two variations of the `ClientKeyExchange` messages in the handshake protocol. One of the two variations is a non-conformant message in which the first two bytes of the plaintext is not $0x0002$, which we call the "`standard error`"; the other variation is a message following one of the ten rules specified in Table 1. For example, "`PKCS#1 conformant`" means that the `ClientKeyExchange` message is correctly formatted according to PKCS#1 standard, but with an incorrect PMS. If this message is not differentiable from the "`standard error`", the library is not vulnerable to Bleichenbacher attacks. "`Wrong Version`" stands for the two version-number bytes are incorrect. In the "`No 0x00 Byte`" test case, the delimiter $0x00$ byte after the padding bytes is changed into a non-$0x00$ byte. The test cases "`0x00 in PKCS Padding`" and "`0x00 in Padding`" mean that some bytes in the corresponding padding bytes are modified to $0x00$, which they should not if the messages are conformant. Note that the first 8 bytes of the padding string are the PKCS paddings while the rest bytes are regular paddings. They are treated by the SSL/TLS library differently. "`PMS Size`" test cases are performed by moving the $0x00$ byte to somewhere in the middle of the `PreMasterSecret` (PMS) so that PMS is truncated. For example, "`PMS Size=2`" is done by moving $0x00$ to the third last byte. Note all the `PreMasterSecret` (PMS) in these tests are invalid so that the error handling procedure is always triggered.

The results of the analysis are shown in Table 1. "`D`" suggests that in the differential analysis, the two traces, when converted to the corresponding level, are differentiable; "`N`" means the two traces are not differentiable. If "`PKCS#1 conformant`" is differentiable from "`standard error`", it means we can construct an oracle that when it returns true, we are certain that the corresponding plaintext message starts with $0x0002$. This means the tested library is considered exploitable by a Bleichenbacher attack (labeled "✓" in the row with the header "`Exploitable`"). However, what we do not know is whether the oracle returning false means the message does not begin with $0x0002$. If, at the same time, some of the other 9 tests yield differentiable traces, it means we have a higher probability to assert that when the oracle returns false the message does not start with $0x0002$, which leads to a stronger oracle.

In the cacheline-level and branch-level control-flow inference attacks, the oracle strength is 1 for all libraries; this is because the oracle only returns false when "`standard error`" happens. In these cases, we have a very strong oracle that can help break the secret with fewer queries. The page-level attacks against GnuTLS and mbedTLS also have an oracle strength of 1, but those for OpenSSL and LibreSSL are lower, which is roughly $(\frac{255}{256})^8 \times (1 - (\frac{255}{256})^{49}) \approx 0.1691$ when the RSA key size is 2048 (see Figure 3)[2]. It means the adversary needs to send more (roughly, $\frac{1}{0.1691} = 5.9\times$) queries to the "weaker" oracle compared to using a stronger oracle (*i.e.*, oracle strength is 1). The oracle strength for page-level control-flow inference attacks against WolfSSL is the lowest, because most of the differential tests render *non-differentiable*, making the oracle attack very slow.

4.2.2 Padding Oracle Tests. To detect vulnerabilities that give rise to CBC padding oracle attacks, we also performed a series of tests. In each test, the framework is provided with two *application data* messages that are encrypted with symmetric keys in the CBC mode. All messages are four blocks in length. One message only has an incorrect MAC, the "`standard error`", and the other message has both an incorrect MAC and one of the six padding errors listed in Table 1. Specifically, the six test cases can be divided into two groups: The first group of tests is conducted by modifying the "`Padding Length Byte`" which is the last byte of a record; the second group modify the last "`Padding Byte`" which is the second last byte of a record. The two groups each generate one error padding case by (1) XORing the target byte with 1, (2) setting it to 0x00 and (3) setting it to 0xFF. We note that the test of "`Padding Length Byte = 0x00`" is special. When it yields differentiable traces, it means the padding length cannot be 0x00, therefore our test should break the last two bytes together (by looking for paddings of $0x0101$). Otherwise, the attack can start with guessing the last byte, greatly reducing the complexity of the attacks. None of the SSL/TLS implementations we tested allow padding length to be $0x00$. But it only makes the attack slightly longer and does not eliminate its vulnerability.

When the "`standard error`" can be differentiated from the various padding errors, the library is vulnerable to padding oracle attacks. The results of the analysis are shown in Table 1. We can see that almost all libraries, except for OpenSSL, are vulnerable to all levels of control-flow inference attacks. OpenSSL 1.0.2j is not vulnerable to page-level attacks because all its distinguishable traces are contained in the same page. Another exception is GnuTLS. As we have mentioned before, the Pintool does not support GnuTLS's `gnutls_record_recv()` function due to a bug in the tool. Consequently, we used Callgrind-based trace recorder, which does not support analysis at the branch level and the cacheline level.

4.2.3 Findings. Although some cryptographic libraries, such as OpenSSL, aim to enforce constant-time implementations, Stacco

[1] As of August 2017, Intel's SGX SDK [7] only contained a cryptographic library; and the SSL/TLS implementation was not completed. Therefore, we could not conduct tests on Intel's official SGX SSL/TLS implementation.

[2] It requires $0x00$ in the 8-byte PKCS padding and No $0x00$ in the last 48 + 1 bytes.

Table 1: Experiment results of the differential analyses. B: vulnerable to branch-level attacks? C: vulnerable to cacheline-level attacks? P: vulnerable to page-level attacks? D: Differentiable; N: Not differentiable; N/A: unable to test.

	Test Name	OpenSSL 1.0.2j			GnuTLS 3.4.17			mbedTLS 2.4.1			WolfSSL 3.10.0			LibreSSL 2.5.0		
		B	C	P	B	C	P	B	C	P	B	C	P	B	C	P
Bleichenbacher attacks	PKCS#1 Conformant	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
	Wrong Version	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	No 0x00 Byte	D	D	N	D	D	D	D	D	D	D	D	N	D	D	N
	0x00 in Padding	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	0x00 in PKCS Padding	D	D	N	D	D	D	D	D	D	D	D	D	D	D	N
	PMS Size=0	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	PMS Size=2	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	PMS Size=8	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	PMS Size=16	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	PMS Size=32	D	D	D	D	D	D	D	D	D	D	D	N	D	D	D
	Exploitable	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Padding Oracle attacks	Padding Length Byte XOR 1	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Padding Length Byte = 0x00	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Padding Length Byte = 0xFF	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Last Padding Byte XOR 1	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Last Padding Byte = 0x00	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Last Padding Byte = 0xFF	D	D	N	N/A	N/A	D	D	D	D	D	D	D	D	D	D
	Exploitable	✓	✓	✗	N/A	N/A	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

finds that all five SSL/TLS libraries are vulnerable to control-flow inference attacks. In most cases, page-level attacks are sufficient to create an oracle and perform oracle attacks against these libraries. We scrutinized the identified vulnerabilities and found that there are primarily two reasons for the leakage. First, the oracles for Bleichenbacher attacks are typically caused by the improper error logging and reporting mechanisms in the library. Second, the oracles for padding oracle attacks are typically created by the improper implementation of constant-time cryptography in the patches for the existing side-channel attacks (*e.g.*, the Lucky Thirteen attack, cache attacks, *etc.*). We briefly summarize one example in this section, and list vulnerable code for other vulnerabilities in Appendix A.

Particularly, the CBC padding oracle in GnuTLS v3.4.17 can be constructed by monitoring the execution order of the function `ciphertext_to_compressed()` (see Listing 1) and the function `_gnutls_auth_cipher_add_auth()` (Listing 2). Specifically, `dummy_wait()` is called in `ciphertext_to_compressed()` when the padding or MAC is incorrect. This function was designed to defeat the timing-based Lucky Thirteen attack [12] by introducing intentional delays. However, `dummy_wait()` checks if the error is caused by incorrect padding (line 3 of Listing 2), and calls `_gnutls_auth_cipher_add_auth()` (line 8 and 12 of Listing 2) if the padding is correct (and the MAC is incorrect). In this example, the decryption oracle is introduced by the defense against timing attacks, but the control flow of the additional delay is leaked to the more powerful man-in-the-kernel attackers.

5 VULNERABILITY VALIDATION

To validate the detected sensitive control-flow vulnerabilities by Stacco, in this section, we describe in details two types of oracle attacks against SSL/TLS implementations in enclaves: CBC padding oracle attacks and Bleichenbacher attacks.

Listing 1: Snippet of `ciphertext_to_compressed()`.

```
1  ...
2
3  ret =
4      _gnutls_auth_cipher_tag(&params->read.cipher_state,
           tag, tag_size);
5  if (unlikely(ret < 0))
6       return gnutls_assert_val(ret);
7
8  if (unlikely
9      (gnutls_memcmp(tag, tag_ptr, tag_size) != 0 ||
           pad_failed != 0)) {
10      /* HMAC was not the same. */
11      dummy_wait(params, compressed, pad_failed, pad,
12             length + preamble_size);
13
14      return
            gnutls_assert_val(GNUTLS_E_DECRYPTION_FAILED);
15 }
16
17 ...
```

5.1 Attack Implementation

We implemented the page-level control-flow inference attacks by extending the mainstream Linux operating system kernel. Our implementation of the attack is shown in Figure 8. The extensions to the kernel include several loadable kernel modules (LKM) and some modification of the core kernel components.

In the core kernel components, particularly, we modified the page-fault handling routine so that it is able to tackle a category of page faults that are triggered because one of the reserved bits (*e.g.*,

Listing 2: Snippet of `dummy_wait()`.

```
...

if (pad_failed == 0 && pad > 0) {
    len = _gnutls_mac_block_size(params->mac);
    if (len > 0) {
        if ((pad + total) % len > len - 9 && total
             % len <= len - 9) {
            if (len < plaintext->size)
                _gnutls_auth_cipher_add_auth
                    (&params->read.cipher_state,
                    plaintext->data, len);
            else
                _gnutls_auth_cipher_add_auth
                    (&params->read.cipher_state,
                    plaintext->data,
                        plaintext->size);
        }
    }
}
```

bit 51 in our implementation) in the page table entries (PTE) is set. This reserved bit in PTE is not already used by the Linux kernel, so only the attack code could have triggered this type of faults. When such page faults are intercepted, the page-fault handler resets the reserved bit of the corresponding PTE to 0 so that future accesses to the same page will be allowed (because otherwise the process will hang due to frequent page faults); it also sets the reserved bit of the last accessed page, tracked in a global variable in the kernel, to 1 in order to capture future access of it. In sum, the kernel only allows one executable page in the ELRANGE of the victim process (that the attacker is interested in monitoring) to be accessible at a time.

A data array, dubbed *Page Trace Recorder (PTR)*, is added to the kernel space for the page-fault handler to record the list of virtual pages that has been accessed by the enclave program. Each time a page fault triggered by the reserved bit in PTEs occurs, the faulting page is appended to the list, which also increments a global counter by one.

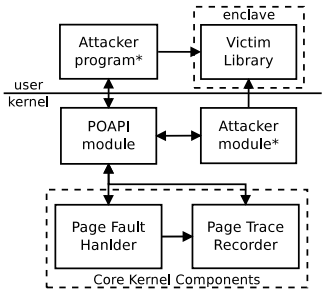

Figure 8: Overview of the attack implementation.

The attacks can be initiated either from the userspace or from the kernel. To facilitate the attacks, we implemented a set of kernel interfaces, dubbed Paging Oracle Attack Program Interface (POAPI), that can be triggered from both userspace and kernel space. The interfaces are encapsulated as a kernel module, *i.e.*, the POAPI module in Figure 8. The interfaces are either used by a userspace program, the attacker program in the figure, or by another kernel module, the attack module in the figure. As either an attack program or an attack module is needed in the two attacks we describe shortly, they are labeled with asterisks to indicate only one of them is needed in an attack.

To initiate the attack, the POAPI module is provided with the name of the victim process, the virtual addresses of the EPC pages to be monitored, and the specific page sequence (specified using page indices rather than virtual addresses) to be monitored for oracle construction. The sequence of pages is also called the *template sequence*. POAPI locates the page tables of the victim process in the kernel and sets the reserved bits of the PTEs to be 1 so that accesses to these pages by the enclave code will be trapped into the kernel. The template sequence is translated into the sequence of virtual pages in this step, so it can be matched later with virtual page sequences in PTR. POAPI provides two addition interfaces: First, a `Reset()` call that will reset the PTR to empty. This functionality is important in our oracle attacks as we need to repeatedly query the oracles. Second, an `Oracle()` call that will return true or false: If the template sequence matches the entire sequence in PTR, then `Oracle()` returns true; otherwise it returns false.

5.2 CBC Padding Oracle Attacks

The oracle. We demonstrated the CBC padding oracle attack on the implementation of TLS v1.2 in GnuTLS 3.4.17 (latest version as of February 2017). Stacco suggests that this implementation is vulnerable to page-level control-flow inference attacks. More specifically, the correctness of the paddings can be revealed by the execution order of two functions, `ciphertext_to_compressed()` and `_gnutls_auth_cipher_add_auth()`. We found in our experiments that by monitoring only the memory pages that contain these two functions the adversary is able to construct a powerful oracle for plaintext recovery. Therefore, our template sequence only contains two memory pages. Note that `ciphertext_to_compressed()` is large and spans two pages. We selected the second page to monitor. This is instructed by the differential analysis tool already, so no manual analysis is needed. By labelling the memory page containing `ciphertext_to_compressed()` as index 0 and that containing `_gnutls_auth_cipher_add_auth()` as index 1, the template sequence is "1010101010".

Detailed implementation. We run the victim library inside SGX with help of Graphene-SGX [5]. Particularly, the victim GnuTLS library we attack is loaded as `sgx.trusted_files` into enclaves with the victim server programs. However, GnuTLS does not support Intel SGX: The initialization of the library will check the availability of accelerated encryption instructions with the `CPUID` instruction—an instruction not supported by SGX. Thus we modified the library slightly by simply removing the check to allow it to run directly in enclave (inside Graphene).

The padding oracle attack is implemented as a kernel module that leverages the POAPI to query the padding oracle constructed from the page-level control-flow inference attacks. The attack starts after the SSL/TLS server in Graphene has been launched. The process name, the virtual address of the two memory pages that contain the two functions, and a template sequence "1010101010" are provided through POAPI. If the encrypted message has a valid padding but invalid MAC, `Oracle()` will find a match in the PTR and return

true. If both the padding and the MAC are invalid, a sequence of "10101010" will be found in the PTR.

Following prior studies [12, 38], the padding oracle attack is performed over multiple TLS sessions. This attack is practical when the victim client can be triggered to repeatedly establish new TLS connections with the victim server and send the same message in each new connection. Particularly, the victim client first establishes a TLS connection with the victim server using the SSL handshake protocol and negotiates to use the `TLS_RSA_WITH_AES_128_CBC_SHA` ciphersuite in TLS v1.2 (through a process that can be heavily influenced by the man-in-the-kernel attacker). Then it sends an encrypted data record to the server. The man-in-the-kernel attacker modifies the ciphertext to prepare its query to the oracle. If the server receives a data record with incorrect MAC or incorrect padding, it sends a *bad_record_mac* alert to the client and shuts down the current TLS session. When the client receives the alert, it immediately restarts a new TLS connection to start a new query. The ciphertext will be different from the first time, as the symmetric key, *i.e.*, AES key, used to encrypt the data is different. The adversary will intercept the message, again, and make modifications according to its adaptive query strategies.

The attack kernel module we implemented for the CBC padding oracle attacks, upon kernel module initialization, also registers a Netfilter to intercept all the traffic sent to and from the server process, by filtering traffic with specific port number. More specifically, two hooks were registered with hooknum of `NF_IP_LOCAL_IN` and `NF_IP_LOCAL_OUT`. With this functionality, the adversary is able to examine each SSL/TLS packet and determine the packet type by reading the first five bytes in the data segment of packets. Byte 1 indicates the content type. The adversary is interested in two particular types: $0x15$ and $0x17$. $0x15$ means the packet is an Alert message and $0x17$ means Application Data. Byte 2 and 3 are TLS versions. Since we are attacking TLS 1.2, they should be $0x0303$. The last two bytes indicate the compressed plaintext length. If an Alert message sent from the SSL/TLS server to the client is observed by the kernel module, it means that the server has decrypted the malformatted record and sent the client a `bad_record_mac` alert, the adversary immediately checks the whether the corresponding plaintext padding of the modified record is valid by calling `Oracle()`. Notice that when the Netfilter intercepts the packet and modifies the ciphertext, all the `checksums`, such as IP and TCP `checksums`, will fail when checked by the kernel. Thus, a special flag is added to the modified packets and the kernel is modified to bypass all packet integrity checks upon appearance of this flag.

Evaluation. The complexity of plaintext recovery with AES encryption is at most $2^{16} + 14 \times 2^8 = 69,120$ queries. This is because the last two bytes need to be enumerated together, but the rest of the bytes can simply be decrypted one byte after another, leading to a linear complexity in the size of the block. In our experiment to decrypt one block with random data, the number of queries was 48388 and the execution time of the attack was 51m13s (less than an hour).

Breaking mbedTLS-SGX. We also succeeded in carrying out the CBC padding oracle attacks against an open-source SGX implementation of mbedTLS, mbedTLS-SGX [8], which can be loaded natively in enclaves. Guided by Stacco, we chose two pages containing the functions `sha1_process_wrap()` and `mbedtls_sha1_process()` to monitor. The template sequence is "0101...10" (15 zeros and 14 ones). Incorrect-padding traces are "0101...010" (14 zeros and 13 ones) in all cases. In our experiment, the attack took 29 minutes and 29 seconds with 25,717 queries to complete the decrypting of one random AES block.

5.3 Bleichenbacher Attacks

The oracle. Our attack target was the implementation of TLS v1.2 in OpenSSL 1.0.2j (latest as of February 2017). Stacco identified a vulnerability in the implementation: the control flows involving `ERR_put_error()` and `RSA_padding_check_PKCS1_type_2()` may leak sensitive information regarding the correctness of the formatting. We label the two memory pages that contain the two functions, respectively, as page 0 and page 1. The template sequence is "1010". Therefore, if the page sequence in the PTR matches the template, the `Oracle()` returns true. Otherwise, in which case the sequence in the PTR is typically "10101010", the oracle returns false.

Detailed implementation. We use Graphene-SGX to run *unmodified* OpenSSL inside SGX enclaves. Unlike GnuTLS, OpenSSL does not have enclave-illegal instructions and can be loaded and ran directly by an SSL/TLS server as `sgx.trusted_files` in the enclave with Graphene. To complete the attack, we extended the open-source tool, TLS-Attacker [66], and implemented an add-on module. We chose TLS-Attacker because it enables us to easily replace the `ClientKeyExchange` message with any message we would like the oracle to test. We did not implement any additional kernel modules besides POAPI, as the desired computation in kernel space is rather inefficient. All the attack steps were accomplished in the userspace with the support of POAPI for querying the oracle.

With an intercepted `ClientKeyExchange` message, the attacker conducts the Bleichenbacher attack to decrypt it and extract the `PreMasterSecret`. To do so, the attacker initializes the man-in-the-kernel attacks through the POAPI module and provides the server process name, the virtual addresses of the two target pages, and the template sequence "1010". Then the attacker establishes a series of queries: Before sending each query, he initiates a new TLS handshake with the victim server. Right before sending the crafted `ClientKeyExchange` message, the attacker calls `Reset()` to POAPI to reset the PTR. Then the crafted message is sent and the attacker waits until receiving the Alert message from the server. Then the attacker calls `Oracle()` to query the oracle, depending on the return value, the next ciphertext is calculated. This process continues until the plaintext of the `ClientKeyExchange` message is recovered.

Evaluation. The numbers of queries that are needed to break the `ClientKeyExchange` message encrypted with a 1024-bit RSA key, a 2048-bit RSA key and a 4096-bit RSA key are shown in Table 2. It can be seen that the numbers of queries for breaking `ClientKeyExchange` encrypted with 1024-bit key and 2048-bit key are similar, this is because the oracle strength is not linear in the size of the keys. Breaking the 2048-bit key encrypted `ClientKeyExchange` takes roughly half an hour. Once the `PreMasterSecret` is known, the attacker can decrypt all the intercepted `application data` packets and hijack the future communication if the session is still

Table 2: Performance of the Bleichenbacher attacks against OpenSSL with different key size.

	1024	2048	4096
Num. of queries	19,346	19,368	57,286
Time to succeed	28m20s	33m24s	1h31m39s

alive. We anticipate an optimization in the attacks will further speed up the process, possibly making an online SSL/TLS hijacking attack feasible.

6 COUNTERMEASURES

Countermeasures to the demonstrated attacks can be pursued in three different layers:

Preventing control-flow inference attacks. Although Intel claims side-channel attacks are outside the threat model of SGX [6], given the severity of the demonstrated attacks (among the others [43, 63, 73]), we believe it is reasonable for Intel to start exploring solutions to these side-channel attacks, particularly control-flow inference attacks. Some academic research studies have already made some progress towards this direction [27, 32, 62, 63]. However, all of these prior work only considers some of the side-channel attack vectors. But effective solutions require a complete understanding of the attack surfaces. Due to the lack of systematic knowledge, none of the prior solutions have successfully prevented control-flow inference attacks on all levels (*i.e.*, page-level, cache-level, and branch-level). We believe a considerable amount of research in this direction is warranted.

Patching sensitive control-flow vulnerabilities. An alternative solution is to patch the vulnerabilities in the SSL/TLS implementations. Constant-time cryptography has been regarded the best practice to address side-channel issues. But as shown in our study, what create the oracles are not always the cryptographic operations themselves, but sometimes the error handling and reporting functions (see Appendix A). Therefore, although some previous work has attempted to verify the constant-time implementation in OpenSSL [15, 16], our attacks suggest that the entire software package needs to be analyzed together rather than individual algorithms. By contrast, Stacco can be used to dynamically analyze the whole software program and pinpoint the vulnerabilities that violate the constant-time programming paradigm. However, we admit that patching the vulnerability is still a manual work. For instance, as shown in Listing 4 and Listing 5, the oracle can be removed by eliminating the `RSAerr()` call in `RSA_padding_check_PKCS1_type_2()` as well as that in its caller function, `RSA_eay_private_decrypt()`, since errors will be reported again after the `PreMasterSecret` is recognized as invalid. If different error types are to be reported, different return values could be used to tell them apart. After applying the patches manually, the SSL/TLS libraries can be tested again using Stacco to identify the remaining vulnerabilities. Future work should emphasize the automation of program analysis and patching.

Avoiding using the vulnerable ciphersuites. The root cause of the Bleichenbacher attacks is the use of the RSA algorithm for key exchanges, which gives the man-in-the-middle adversary opportunities to query oracles and decrypt the key materials. It has been shown that both PKCS#1 v1.5 and PKCS#1 v2 (*i.e.*, RSAES-OAEP) [47] are vulnerable to such attacks. Therefore, to completely mitigate Bleichenbacher attacks, RSA-based key exchange must be prohibited and the use of DH key exchanges must be enforced. To mitigate CBC padding oracle attacks, it is recommended to replace `MAC-Then-Encrypt` with the Authenticated Encryption with Associated Data (AEAD) mode, *e.g.*, AES-GCM. In the draft version of TLS v1.3 [11], it is recommended to use ciphersuites that employ DH for key exchanges and AEAD modes for symmetric encryption. However, because most SSL/TLS implementations need to be backward compatible, it may take years before RSA-based key exchanges and CBC mode symmetric encryptions completely phase out. Given the severity of the demonstrated attacks in this paper, we recommend enforcing the use of secure ciphersuites for enclave programs, but it also means that any entities communicating with the secure enclaves must design special security policies to disallow the use of these vulnerable ciphersuites also.

7 RELATED WORK

7.1 SSL/TLS Oracle Attacks

CBC padding oracle attacks. The first discussion of the CBC padding oracle attacks was by Vaudenay in this seminal paper [71]. It was shown that plaintext recovery is possible without decryption keys if an oracle that differentiates correct padding from incorrect padding is available to the adversary. Implication on TLS v1.0 and SSL v3.0 was discussed in the paper, which suggested TLS v1.0 was potentially exploitable by such attacks because the padding error message is observable as a reply, but SSL v3.0, due to its unified error message, is more challenging to exploit. The Vaudenay attack has been mitigated thereafter by eliminating the error-message oracles.

The work of Vaudenay was followed up by several studies. Canvel *et al.* [25] exploited the timing differences in the SSL/TLS message decryption process as a padding oracle. The timing differences are caused by the absence of MAC checks when the format of the padding is incorrect. The countermeasure implemented in popular SSL/TLS libraries, *e.g.*, OpenSSL, is to compute MAC regardless of the padding correctness. However, almost 10 years later, this defense mechanism was circumvented by more sophisticated timing analysis attacks. AlFardan *et al.* [12] describe a new padding oracle attack, dubbed Lucky Thirteen Attack, in which the adversary is able to distinguish the latency of the returned SSL error message when the number of hash function calls is different. The nuance can serve as a padding oracle because the correctness of the padding also dictates the number of hash function calls. Although the Lucky Thirteen Attack was soon patched by adding dummy hash operations to enforce constant-time execution, new vulnerabilities were later discovered by Albrecht *et al.* [13], who proposed new variants of Lucky Thirteen attacks against the SSL/TLS implementation in Amazon's s2n. Möller *et al.* [52] performed a downgrade attack against SSL/TLS, by forcing the use of SSL v3.0 during the negotiation of cipher suite in the SSL handshake protocol. This attack is also called POODLE attacks. SSL v3.0 is inherently vulnerable to padding oracle attack because its the MAC does not protect the

padding and also padding is nondeterministic except for the last byte. Due to POODLE attacks, SSL v3.0 has been deprecated.

Closer to our study is Irazoqui *et al.* [38], who demonstrated padding oracle attacks enabled by Flush-Reload cache side channels. Our work goes beyond their study in two dimensions: first, we systematically model various types of control-flow inference attacks under the scenarios of secure enclaves, instead of considering only cache side-channel attacks. Second, they only manually studied the source code of SSL/TLS implementations. In contrast, we proposed a differential analysis framework to detect vulnerabilities in a wide range of SSL/TLS implementations, enabling examination of future implementations in an automated and black-box fashion.

Bleichenbacher attacks. Oracle attacks due to format errors in asymmetric encryption, *i.e.*, RSA algorithms, can date back to 1998 when Bleichenbacher [22] brought forward the first attack against PKCS#1. These attacks rely on oracles of correctly formatted plaintext message conforming to PKCS#1 v1.5 standard (*i.e.*, plaintext message must start with 0x0002). PKCS#1 v2 introduced RSAES-OAEP which employs Optimal Asymmetric Encryption Padding (OAEP) to mitigate this original Bleichenbacher attack. A few years later, it was discovered by Manger [47] that in RSAES-OAEP, the length limitation of the plaintext to be encrypted renders its first byte to be 0x00; failure of conformant to this standard will produce an error message, which can serve as a new Bleichenbacher oracle. Two years later, a new, so-called bad-version oracle, was discovered by Klima *et al.* [42] by checking error message regarding incorrect SSL/TLS version number in the formatted message. The efficiency of the original Bleichenbacher attack was improved by Bardou *et al.* [19], which was also the basis of our attacks. The solution to these attacks was to unify the error messages so that the adversary is not able to distinguish this particular format error. To achieve this, TLS v1.0, v1.1, and v1.2 specification all prescribe that a random number is generated and used as the `PreMasterSecret` to enforce approximately equal processing time for both compliant and non-compliant `ClientKeyExchange` messages.

In 2014, Meyer *et al.* [51] found that SSL/TLS was vulnerable to timing-based Bleichenbacher attacks. Their attack was enabled by a timing oracle due to the extra time used to generate pseudo-random numbers when messages were non-compliant. Most recently, Aviram [18] managed to leverage Bleichenbacher attacks to break TLS 1.2, if the private key is shared with an SSL/TLS server that supports the legacy SSL v2.0 protocol, which is still vulnerable to simple Bleichenbacher attacks. A large number of servers were vulnerable to this so-called DROWN attack. In the non-TLS setting, Bleichenbacher attacks have been employed to break XML encryption [39, 79].

Our work suggests that under the scenario of secure enclaves, even the latest SSL/TLS implementations are vulnerable to Bleichenbacher attacks because the oracles due to sensitive control-flow vulnerabilities are difficult to conceal even in shielded enclaves.

7.2 Intel SGX: Applications and Attacks

Intel SGX is a revolutionary technology for applications that require shielded execution—execution that is isolated from interference or inspection by any other software components including the privileged system software. It also offers remote attestation and sealed storage primitives for trusted computations. Applications that utilize these new SGX features have been proposed in previous studies [17, 20, 36, 50, 59, 69, 76]. Others have been working on facilitating the development and security protection of SGX enclaves [48, 61, 64].

Side-channel attacks against SGX enclaves have been described in a few studies. For example, Xu *et al.* [73] and Shinde *et al.* [63] explored leakage of page-level memory access pattern due to induced page-fault traces. Lee *et al.* [43] explored processor Branch Target Buffers (BTB) to exploit sensitive control-flow vulnerabilities in secure enclaves. Besides these new attacks in SGX contexts, existing cache attacks are also applicable against SGX enclaves [23, 60], since SGX provides no additional protection against such attacks. Defenses against these attacks are implemented on the hardware level [32] or as compiler extensions [27, 62, 63]. These defenses primarily work on page-fault attacks [27, 32, 62, 63] or interrupt-based side-channel attacks [27, 62]. Therefore, they only remove a portion of the entire attack surface. Completely eliminating control-flow inference attacks that we model in Section 3 is extremely challenging. Therefore, our study of SSL/TLS implementations' sensitive control-flow vulnerabilities is not completely addressed by any of these specific defense techniques.

7.3 Security Analysis of TLS Implementations

There has been work on verifying constant-time implementation for SSL/TLS libraries [15, 16]. However, our findings suggest that control-flow leakages still exist even when constant-time mechanisms are employed, especially when the constant-time implementation is enforced by making dummy function calls which may include control flows that depend on the error types. We also find leakage occurs when the internal error logging and reporting functions reveal the reasons for the errors.

Differential analysis has been applied to examine the implementation of certificate validation in TLS libraries [24, 29]. Others focus on determining whether a TLS implementation correctly follows the TLS protocol [21, 34, 66]. Our work is different from them both in the design goals and the methodologies.

8 CONCLUSION

In this paper, we studied oracle attacks against SSL/TLS implementations in SGX. These attacks are enabled by sensitive control-flow vulnerabilities in SSL/TLS libraries and are exploitable by branch-level, cacheline-level, and page-level control-flow inference attacks. Our implementation of man-in-the-kernel attacks empirical demonstrated that the resulting oracle attacks are highly efficient. We also designed a differential analysis framework to help detect these vulnerabilities automatically. We show that all the open-source SSL/TLS libraries we examined are exploitable, thus raising the questions of secure development and deployment of SSL/TLS in SGX enclaves.

ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers for their constructive comments. This work was supported in part by NSF 1566444.

REFERENCES

[1] 2014. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes:1,2A,2B,2C,3A,3B and 3C. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. (2014). version 052, retrieved on Dec 25, 2014.

[2] 2014. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (2014). October 2014.

[3] 2017. GnuTLS Release News. http://www.gnutls.org/news.html. (2017). [Online; accessed February-2017].

[4] 2017. GnuTLS Security Advisories. https://www.gnutls.org/security.html. (2017). [Online; accessed February-2017].

[5] 2017. Graphene-SGX. https://github.com/oscarlab/graphene. (2017). [Online; accessed February-2017].

[6] 2017. Intel Software Guard Extensions, Enclave Writer's Guide, revision: 1.02. (2017). https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf, retrieved on May, 2017.

[7] 2017. Intel Software Guard Extensions SSL. https://github.com/01org/intel-sgx-ssl. (2017). [Online; accessed August-2017].

[8] 2017. mbedTLS-SGX. https://github.com/bl4ck5un/mbedtls-SGX. (2017). [Online; accessed February-2017].

[9] 2017. OpenSSL 1.0.2 Changes. https://www.openssl.org/news/cl102.txt. (2017). [Online; accessed February-2017].

[10] 2017. OpenSSL Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html. (2017). [Online; accessed February-2017].

[11] 2017. The Transport Layer Security (TLS) Protocol Version 1.3, draft-ietf-tls-tls13-latest. (2017). https://tlswg.github.io/tls13-spec/, retrieved on May, 2017.

[12] Nadhem Al Fardan and Kenneth Paterson. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*. IEEE.

[13] Martin Albrecht and Kenneth Paterson. 2016. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 622–643.

[14] Frances Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.

[15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François" Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *23rd International Conference on Fast Software Encryption*. Springer Berlin Heidelberg, 163–184.

[16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security Symposium*. 53–70.

[17] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.

[18] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. 2016. DROWN: breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*.

[19] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. 2012. Efficient padding oracle attacks on cryptographic hardware. *Advances in Cryptology–CRYPTO 2012* (2012), 608–625.

[20] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[21] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 535–552.

[22] Daniel Bleichenbacher. 1998. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*. Springer.

[23] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC.

[24] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 114–129.

[25] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. 2003. Password Interception in a SSL/TLS Channel. In *23rd Annual International Cryptology Conference on Advances in Cryptology - CRYPTO*. Springer Berlin Heidelberg.

[26] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. 2007. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. (2007).

[27] Sanchuan Chen, Xiaokuan Zhang, Michael Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 7–18.

[28] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 2–13.

[29] Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *10th Joint Meeting on Foundations of Software Engineering*. ACM, 793–804.

[30] Yueqiang Cheng, Xuhua Ding, and R Deng. 2013. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13* 101 (2013).

[31] Jeremy Clark and Paul van Oorschot. 2013. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society.

[32] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*. 857–874.

[33] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[34] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security*, Vol. 15. 193–206.

[35] Tim Dierks. 2008. The transport layer security (TLS) protocol version 1.2. (2008).

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

[37] Owen Hofmann, Sangman Kim, Alan Dunn, Michael Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 265–278.

[38] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 strikes back. In *10th ACM Symposium on Information, Computer and Communications Security*. ACM, 85–96.

[39] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. 2012. Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption. In *17th European Symposium on Research in Computer Security*. Springer Berlin Heidelberg.

[40] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 143–157.

[41] Leo Kelion. 2016. 'Thousands of popular sites' at risk of DROWN hack attacks. http://www.bbc.com/news/technology-35706730. (2016). [Online; accessed February-2017].

[42] Vlastimil Klima, Ondrej Pokorný, and Tomáš Rosa. 2003. Attacking RSA-based sessions in SSL/TLS. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 426–440.

[43] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2016. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952* (2016).

[44] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. MiniBox: A Two-way Sandbox for x86 Native Code. In *2014 USENIX Annual Technical Conference*.

[45] David Lie, Chandramohan Thekkath, and Mark Horowitz. 2003. Implementing an Untrusted Operating System on Trusted Hardware. In *19th ACM Symposium on Operating Systems Principles*. ACM.

[46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[47] James Manger. 2001. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS# 1 v2. 0. In *Annual International Cryptology Conference*. Springer, 230–238.

[48] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. *IACR Cryptology ePrint Archive* 2017 (2017), 48.

[49] Jonathan McCune, Bryan Parno, Adrian Perrig, Michael Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 315–328.

[50] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *HASP@ ISCA* 10 (2013).

[51] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. 2014. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*

(USENIX Security 14). 733–748.
[52] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. (2014). https://www.openssl.org/~bodo/ssl-poodle.pdf.
[53] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.
[54] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*. 619–636.
[55] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2–11.
[56] Dan R. K. Ports and Tal Garfinkel. 2008. Towards Application Security on Untrusted Operating Systems. In *3rd Conference on Hot Topics in Security*.
[57] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*. 431–446.
[58] Eric Rescorla. 2006. The transport layer security (TLS) protocol version 1.1. *Transport* (2006).
[59] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.
[60] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. Springer International Publishing, Cham, 3–24. https://doi.org/10.1007/978-3-319-60876-1_1
[61] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.
[62] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.
[63] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *11th ACM Symposium on Information, Computer and Communications Security*.
[64] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *2017 Annual Network and Distributed System Security Symposium(NDSS), San Diego, CA*.
[65] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2016. A Design and Verification Methodology for Secure Isolated Regions. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.
[66] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.
[67] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1492–1504.
[68] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation*.
[69] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy*. IEEE, 19–34.
[70] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Ninth European Conference on Computer Systems*. ACM.
[71] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding-Applications to SSL, IPSEC, WTLS.... In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 534–545.
[72] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. 2004. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*. Springer, 440–447.
[73] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.
[74] Jisoo Yang and Kang Shin. 2008. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM.
[75] Charles Yount, Harish Patil, Mohammad Islam, and Aditya Srikanth. 2015. Graph-matching-based simulation-region selection for multiple binaries. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 52–61.
[76] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 270–282.
[77] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *23rd ACM Symposium on Operating Systems Principles*. ACM.
[78] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 858–870.
[79] Yinqian Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 990–1003.

A EXAMPLES OF SENSITIVE CONTROL-FLOW VULNERABILITIES

A.1 Padding Oracles in mbedTLS(-SGX)

In mbedTLS v2.4.1 and mbedTLS-SGX, the decryption oracle can be constructed by monitoring the function `sha1_process_wrap()` and `mbedtls_sha1_process()`. Particularly, as shown in Listing 3, the function `ssl_decrypt_buf()` calls `mbedtls_md_process()`, which is a wrapper function that calls both `sha1_process_wrap()` and `mbedtls_sha1_process()`, to conceal the timing difference caused by removing the paddings before calculating the MAC. However, the number of times `mbedtls_md_process()` is called depends on the value of `extra_run`, which is calculated from the length of the padding, `padlen`. Particularly, when the padding is incorrect, `padlen` will be 0, and `mbedtls_md_process()` is called only once. Therefore, the number of calls to `sha1_process_wrap()` and `mbedtls_sha1_process()`, which are located on different pages, has been exploited as the oracle in our demonstrated attacks. We note that the padding oracle is created due to the improper constant-time implementation of defenses to existing attacks.

A.2 Bleichenbacher Attack Oracles in OpenSSL

The oracle in OpenSSL 1.0.2j is created by function `RSAerr()`. As shown in Listing 5, in `RSA_padding_check_PKCS1_type_2()`, when any error is detected during the PKCS decoding procedure, `mlen` will be set to -1. Thus `RSAerr()` will be called to report the error before the function returns. After returning to the caller function `RSA_eay_private_decrypt()` (shown in Listing 4), `RSAerr()` is called one more time. These two calls to the `RSAerr()` reveals a non-PKCS-conformant formatting, which can be exploited as an oracle for Bleichenbacher attacks. The vulnerabilities in OpenSSL is not because of a failed constant-time implementation, but the redundant error reporting and logging mechanisms. One possible suggestion is to avoid repeated error reporting that are due to different reasons.

A.3 Bleichenbacher Oracles in GnuTLS

Similar to OpenSSL, the RSA decryption oracle in GnuTLS is also due to error logging and reporting. As shown in Listing 6, the function `_gnutls_debug_log()` is called for either an incorrect PKCS format or incorrect version numbers. Although GnuTLS applies the countermeasure against Bleichenbacher attack by using

Listing 3: Snippet of `ssl_decrypt_buf()`

```
...

padlen &= correct * 0x1FF;
...

size_t j, extra_run = 0;
extra_run = ( 13 + ssl->in_msglen + padlen + 8 ) / 64
       - ( 13 + ssl->in_msglen + 8 ) / 64;

extra_run &= correct * 0xFF;

mbedtls_md_hmac_update(
     &ssl->transform_in->md_ctx_dec, ssl->in_ctr, 8 );
mbedtls_md_hmac_update(
     &ssl->transform_in->md_ctx_dec, ssl->in_hdr, 3 );
mbedtls_md_hmac_update(
     &ssl->transform_in->md_ctx_dec, ssl->in_len, 2 );
mbedtls_md_hmac_update(
     &ssl->transform_in->md_ctx_dec, ssl->in_msg,
                ssl->in_msglen );
mbedtls_md_hmac_finish(
     &ssl->transform_in->md_ctx_dec,
                ssl->in_msg + ssl->in_msglen );
/* Call mbedtls_md_process at least once due to cache
     attacks */
for( j = 0; j < extra_run + 1; j++ )
    mbedtls_md_process(
         &ssl->transform_in->md_ctx_dec, ssl->in_msg
         );
...
```

Listing 4: `RSA_eay_private_decrypt()`

```
...

switch (padding) {
     case RSA_PKCS1_PADDING:
        r = RSA_padding_check_PKCS1_type_2(to, num,
             buf, j, num);
        break;
     ...
}
if (r < 0)
        RSAerr(RSA_F_RSA_EAY_PRIVATE_DECRYPT,
          RSA_R_PADDING_CHECK_FAILED);
...
```

random `PreMasterSecrets`, the logging functions expose the error messages to the adversary with the capability of conducting control-flow inference attacks.

Listing 5: `RSA_padding_check_PKCS1_type_2()`

```
...

if (tlen < 0 || flen < 0)
        return -1;
if (flen > num)
        goto err;
if (num < 11)
        goto err;
...

good = constant_time_is_zero(em[0]);
good &= constant_time_eq(em[1], 2);
...

good &= constant_time_ge((unsigned int)(zero_index),
     2 + 8);
...

good &= constant_time_ge((unsigned int)(tlen),
     (unsigned int)(mlen));
if (!good) {
        mlen = -1;
        goto err;
}

err:
if (em != NULL)
        OPENSSL_free(em);
if (mlen == -1)
        RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2,
               RSA_R_PKCS_DECODING_ERROR);
return mlen;
```

Listing 6: Snippet of `proc_rsa_client_kx()`

```
...

if (ret < 0 || plaintext.size != GNUTLS_MASTER_SIZE)
     {
     _gnutls_debug_log("auth_rsa: Possible PKCS #1
          format attack\n");
     use_rnd_key = 1;
} else {
     if (_gnutls_get_adv_version_major(session) !=
plaintext.data[0] ||
(session->internals.priorities.allow_wrong_pms == 0
&& _gnutls_get_adv_version_minor(session) !=
     plaintext.data[1])) {
          _gnutls_debug_log("auth_rsa: Possible PKCS
               #1 version check format attack\n");
     }
}
...
```