This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs

Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken (Yihang) Bai, Kevin Butler, and Dave (Jing) Tian

Abstract—Confidential computing aims to secure the code and data in use by providing a Trusted Execution Environment (TEE) for applications using hardware features such as Intel SGX. Timing and cache side-channel attacks, however, are often outside the scope of the threat model, although once exploited they are able to break all the default security guarantees enforced by hardware. Unfortunately, tools detecting potential side-channel vulnerabilities within applications are limited and usually ignore the strong attack model and the unique programming model imposed by Intel SGX. This paper proposes a precise side-channel analysis tool, ENCIDER, detecting both timing and cache side-channel vulnerabilities within SGX applications via inferring potential timing observation points and incorporating the SGX programming model into analysis. ENCIDER uses dynamic symbolic execution to decompose the side-channel requirement based on the bounded non-interference property and implements byte-level information flow tracking via API modeling. We have applied ENCIDER to 4 real-world SGX applications, 2 SGX crypto libraries, and 3 widely-used crypto libraries, and found 29 timing side channels and 73 code and data cache side channels. We also compare ENCIDER with three state-of-the-art side channel analysis tools using their benchmarks. ENCIDER does not only report most of the bugs with 20%-50% run time improvement and 65%-92% memory usage improvement, but also detects 9 missing bugs from these tools. We have reported our findings to the corresponding parties, e.g., Intel and ARM, who have confirmed most of the vulnerabilities detected.

Index Terms—Software side channels, symbolic execution, SGX, API modeling, information-flow tracking.

1 INTRODUCTION

Confidential computing aims to secure the code and data in use by providing a Trusted Execution Environment (TEE) for applications using hardware features such as Intel SGX [1]. These efforts have led to wide-spread industry initiatives. The recent, Confidential Computing Consortium (CCC) project [2], unites Intel, Microsoft, Red Hat, etc. to collaborate on and accelerate the adoption of confidential computing. Major cloud providers including Azure, IBM, and Google Cloud Platform (GCP) have already offered SGX-as-a-Service, providing different SGX SDKs to ease the development of enclave applications. Communication and cryptocurrency applications such as Signal [3] and Ledger [4] have implemented SGX enclaves designed to maintain security even if the cloud provider is compromised. Widely-used crypto libraries such as OpenSSL and mbedTLS have also been ported to SGX enclaves [5], [6] supporting more enclave applications.

While confidential computing provides runtime confidentiality and integrity guarantees, side-channel attacks are often outside the scope of its threat model. For instance, SGX enclave implementations are vulnerable to timing [7], [8], [9], [10] and memory-access attacks [11], [12]. Recent work on micro-architectural vulnerabilities [13], [14] and their applicability to SGX [15] highlight the impact of side-channel attacks against confidential computing technologies by breaking all the default security guarantees. Crypto libraries such as OpenSSL and GnuTLS have combatted side-channel attacks for years [10], [16], [17], and developed various defenses including blinding [7], constant-time programming, pre-loading, hiding different types of failure cases, and simplifying the API complexity [18]. Their use in confidential computing such as Intel SGX demands strict scrutiny since a vulnerability within them can easily defeat the security guarantees enforced by hardware. Meanwhile, vulnerabilities have been discovered within the Intel IPP library [19], the default crypto library used by the Intel SGX SDK [20], breaking the confidentiality of SGX enclaves.

There have been several efforts on automatically detecting side channels in SGX enclaves [17], [21], [22]. However, these studies either treat SGX enclaves as typical software artifacts without considering the SGX programming model- thus, potentially missing timing observation points- or only focus on secret-dependent branches without considering whether a cache side channel is actually feasible from the code. In this paper, we present a precise side-channel analysis tool, ENCIDER, which can detect both timing and cache side channels while incorporating the SGX programming model into the analysis. While the current implementation focuses on SGX enclaves, ENCIDER can be applied to other confidential computing technologies and even traditional software components.

ENCIDER uses a programming model-guided symbolic execution to facilitate the analysis of enclave implementations and cryptographic library functions. It uses precise byte-level information flow tracking to minimize false positives, employing a novel decomposition-based and incremental non-interference analysis that can detect both timing and cache side channels. Finally, ENCIDER uses API modeling to achieve scalability. We have applied ENCIDER to 4 real-world SGX enclaves, 2 SGX cryptographic libraries, 4 TLS implementations, and 3 widely-used cryptographic libraries, and found 29 timing side channels and a total of 73

(27 code and 46 data) cache side channels. If exploited, these would break the security guarantees enforced by Intel SGX. We have notified the corresponding parties and confirmed most of the vulnerabilities that we found.

In summary, our contributions are as follows:

- We develop a novel decomposition-based side-channel analysis approach that can detect both timing and cache side channels while optimizing the number of calls to the underlying SMT solver [23].
- We design and implement ENCIDER, which leverages the memory model of symbolic execution for precise information-flow tracking and achieves scalability through API modeling. Its ability to detect three types of side channels can support secure code development against remote as well as local attackers. ENCIDER will be released at https://github.com/sysrel/ENCIDER.
- We evaluate ENCIDER against real-world SGX enclaves and crypto libraries. ENCIDER has found a local timing side channel in mbedTLS-SGX that can be exploited by Lucky 13 attacks [10], cache side channels in the Signal Contact Discovery Service, s2n, and mbedTLS, timing side channels in the SGX IPP and SSL APIs and in the Ledger BOLOS enclave. ENCIDER is also able to detect previously known timing and cache side channels in s2n, polarSSL, openSSL, mbedTLS, and libgcrypt libraries. ENCIDER achieves 96% precision. We have reported our findings to the corresponding parties, e.g., Intel[1] and ARM[2], who have confirmed most of the vulnerabilities detected.
- We compare ENCIDER with three state-of-the-art side channel analysis tools ct-verif [29], CacheS [30], and DATA [31] using their benchmarks. ENCIDER confirms the side channel freedom of the benchmarks with 50% and 65% run time and memory improvement comparing to ct-verif. ENCIDER not only detects bugs with 20% and 92% run time and memory improvement comparing to CacheS, but also detects 9 missing bugs from CacheS. ENCIDER reports some of the leakages found by DATA, a dynamic analysis tool, while covering more cases and not relying on the availability of tests.

This paper is organized as follows. Section 2 provides background information on Intel SGX. Section 3 discusses the threat model and Section 4 provides a motivating example and gives an overview of our approach. Section 5 presents the technical details of our approach and discusses its correctness. Section 6 provides details on the implementation. Section 7 presents an evaluation of ENCIDER using real-world enclaves and cryptographic libraries. Section 8 discusses related work. Section 9 concludes with directions for future work.

1. Intel issued CVE-2021-0001 [24] and a security advisory [25], [26]
2. ARM issued CVE-2020-16150 [27] and a security advisory [28].

2 Background

2.1 Intel SGX & Programming Model

The Intel Software Guard eXtensions (SGX) [1] are a set of Instruction Set Architecture (ISA) extensions to the Intel x86 and x86_64 processor architectures, which aim to defend ring-3 applications (unprivileged and user mode) against attacks from ring-0 (kernel and operating system), Virtual Machine Monitor Mode (VMM, ring -1), System Management Mode (SMM, ring -2) or even Intel's Management Engine (Intel ME, ring -3) [32]. This secure execution environment in SGX is called an *enclave* and acts as a secure and attestable storage for program code and data, providing runtime confidentiality and integrity protections at the same time. With the help of Intel's EPID [33], a challenger can ensure that the desired enclave has the correct measurement and is running on a genuine Intel CPU with SGX enabled. This enables remote verifiers to assure that a program runs securely and as expected on an untrusted third-party's platform.

Since an SGX enclave only runs within ring 3 (i.e., no *syscall*), the SGX programming model requires developers to partition applications and place only the most security-sensitive code and/or data into an enclave [34]. To enter an enclave, application code needs to execute `ECALL`s, which are a set of fixed entry points defined by the enclave. Similarly, when an enclave needs to communicate with the application, e.g., opening a file, enclave code needs to execute `OCALL`s, which are predefined functions between enclaves and applications. Intel SGX SDK [20] offers the Enclave Definition Language (EDL) to ease the definition of `ECALL` and `OCALL` interfaces and provides the corresponding setup and cleanup code when entering and leaving an enclave. As a result, both `ECALL`s and `OCALL`s need to be carefully designed and implemented to reduce the attack surface to an enclave and the possible information leakage from an enclave.

2.2 Side-channel Attacks Against SGX

While SGX enclaves provide both runtime confidentiality and integrity, side channels are considered to be outside the threat model, and it is the developer's responsibility to prevent these attacks [35]. A wide variety of side-channel attacks have been demonstrated in the academic literature. Controlled-channel attacks [36] use memory access patterns to exfiltrate sensitive information from secure enclaves. Cache-based side-channel attacks [37], [38] have also been effectively deployed against SGX. Meanwhile, memory side-channel hazards were discovered by Wang et al. [39] that affect system elements ranging from TLBs to DRAM modules. CacheZoom [40] demonstrated how SGX amplifies cache side channels by recovering AES keys in a production environment.

Other side-channel vulnerabilities [21] are also found within the Integrated Performance Primitives (IPP) cryptographic library used by Intel SGX SDK. More recently, microarchitectural attacks have been demonstrated to work on SGX enclaves, notably the high-profile Meltdown [13] and Spectre [14] attacks, while Foreshadow [15] attacks extract the attestation key from enclaves, thus breaking SGX remote attestation. These side-channel attacks are enabled

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

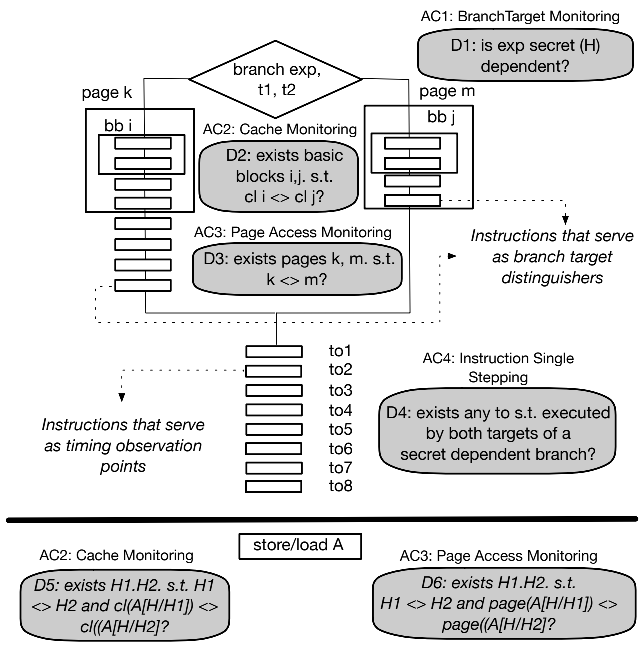


Fig. 1. Attacker capabilities (AC) of an SGX attacker and various rules (D) ENCIDER uses to detect the leakages.

by vulnerabilities within both hardware (e.g., microarchitecture and caches) and software (e.g., input-dependent secret processing). While the former can be fixed by microcode updates or trampolines using the *LFENCE* instruction [41], no tools currently exist to help detect side-channel vulnerabilities within the code itself with a focus on SGX enclave implementations.

In addition, side-channel attacks that occur within libraries developed before SGX can maintain their vulnerabilities when placed within an enclave. An example is the OpenSSL cryptographic library used by a number of SGX applications. OpenSSL was shown to be vulnerable to cache [42] and timing [16] side-channel attacks, and such vulnerabilities could be exploited within an enclave to exfiltrate secret data.

3 Security Model

Although we are targeting SGX enclaves within the paper, ENCIDER is general enough to be applied to any confidential computing technologies such as ARM TrustZone [43] and AMD SEV [44]. Accordingly, we trust the corresponding hardware features employed by CPUs, e.g., SGX instruction extension, providing the claimed runtime confidentiality and/or integrity guarantees. We also assume the basic secure coding practices applied to confidential computing environments including SDKs and applications, e.g., Intel SGX SDK and enclave applications, to reduce the possibility of compromise.

We consider a piece of data as secret if it represents confidential information such as private keys and plaintext data that needs to be protected against unauthorized access. As all other works on side channel analysis we assume that the user identifies data that is considered as secret and we refer to secret (non-secret) data as high (low) security sensitive.

Figure 1 illustrates the various capabilities of an attacker within the context of SGX and the detection rules used by ENCIDER to find the relevant leakages. We categorize attacker capabilities in terms of branch target prediction monitoring (AC1), cache access monitoring (AC2), page access monitoring (AC3), and instruction single-stepping (AC4).

```
1 static int ssl_decrypt_buf( mbedtls_ssl_context *ssl )
2 { ...
3   unsigned char *dec_msg;
4   dec_msglen = ssl->in_msglen;
5   dec_msg = ssl->in_msg;
6
7   if( ( ret = mbedtls_cipher_crypt(
8                 &ssl->transform_in->cipher_ctx_dec,
9                 ssl->transform_in->iv_dec,
10                ssl->transform_in->ivlen,
11                dec_msg, dec_msglen,
12                dec_msg_result, &olen ) ) != 0 )
13  { return( ret ); }
14  ...
15  padlen = 1 + ssl->in_msg[ssl->in_msglen - 1];
16
17
18  if( ssl->in_msglen < ssl->transform_in->maclen + padlen &&
19          auth_done == 0 )
20  {
21    padlen = 0;
22    correct = 0;
23  }
24  ssl->in_msglen -= padlen;
25  size_t j, extra_run = 0;
26  extra_run = ( 13 + ssl->in_msglen + padlen + 8 ) / 64 -
27            ( 13 + ssl->in_msglen          + 8 ) / 64;
28  extra_run &= correct * 0xFF;
29  ...
30  mbedtls_md_hmac_update( &ssl->transform_in->md_ctx_dec,
31        ssl->in_msg, ssl->in_msglen );
32
33  mbedtls_md_hmac_finish( &ssl->transform_in->md_ctx_dec,
34                    ssl->in_msg + ssl->in_msglen );
35
36
37  for( j = 0; j < extra_run + 1; j++ )
38    mbedtls_md_process( &ssl->t..., ssl->in_msg );
39
40 mbedtls_md_hmac_reset( &ssl->transform_in->md_ctx_dec );
41   ...
42 }
```

modeled API
secret output
alias of (see line 5)
secret tainted value
secret dependent branch
secret dependent computation time
time equalizer code
observation point for a local timing side channel

Fig. 2. The local timing side channel that was detected by ENCIDER in the `ssl_decrypt_buf` function in mbedtls 2.6.0 that is used in mbedtls-SGX.

ENCIDER focuses on detecting two different attacks against confidential computing: 1) remote and local timing side-channel attacks caused by potential code path interference when handling secret dependent data within enclave programs, 2) cache side-channel attacks caused by potential cache line sharing among data of differing security levels within enclave programs.

A local timing side channel is associated with a *timing observation point*, which is an action of the software that can be observed by a local attacker and potentially leak information about the secret data due to yielding different timing measurements for different secret inputs. We consider three special types of timing observation points: 1) end of the computation (that is assumed by the related work), 2) ocall callsites as predefined observation points, and 3) the first time execution of a function on an execution path. Both timing and cache side-channel attacks can leverage high resolution timers, e.g., programmable APIC timers [40], to extract the secret information protected by enclaves. Note that attacks exploiting microarchitecture vulnerabilities such as Spectre [14] and Meltdown [13] are out of scope. Similar to other work on IR level timing analysis [29], ENCIDER needs to be supplemented with timing model data for instructions whose timing cost are data-dependent.

4 Motivation and Overview

In this section, we present example code within mbedtls-SGX that has a local timing side channel found by ENCIDER and use it to demonstrate its salient features. The code in Figure 2 is an excerpt from the `ssl_decrypt_buf` function in mbedtls 2.6.0 [45] that is used in mbedtls-SGX [6]. This function is part of the TLS protocol implementation and it handles various modes of encryption. We focus on the CBC mode that uses the Mac-Encode-Encrypt (MEE) approach to generate the cipher text. Inside the `ssl_decrypt_buf` function, a series of reverse operations that consists of Decrypt-Decode-Mac is performed. The decryption is performed by the `mbedtls_cipher_crypt` function at line 7. The secret (high-security sensitive) data is the plaintext message stored in the `dec_msg` array that is the 4th parameter of the `mbedtls_cipher_crypt` function. Note that `dec_msg` includes the plaintext, the padding, and the message authentication code (MAC). The padding length is an important source of information that can be exploited to perform the Lucky 13 attacks [10], which can result in recovery of the plaintext. The MAC verification stage, which includes lines 30-31, may take different amounts of time depending on whether a message's padding is valid (less time) or not (more time).

The code in Figure 2 has gone through various fixes in response to a series of side channel vulnerabilities. The first vulnerability was a timing side channel [46] due to a MAC verification that could be exploited by a remote attacker; the code at lines 37-38, which performs time equalization by performing extra compression operations when padding was valid, was absent. The timing side channel was addressed by introducing the time equalization code, in which the variable `extra_run` denoted the number of times the compression functions should be executed to eliminate the timing difference. However, in the fix, the loop condition shown at line 37 was `j<extra_run`, which was later found to be vulnerable to a code-based cache side channel [47] as the `mbedtls_md_process` function ended up being executed only when the padding was valid. A local attacker observing accesses to the cache lines that correspond to the `mbedtls_md_process` basic blocks would reveal the key information that would enable the Lucky13 attacks.

The fix for the code cache side channel mentioned above was executing the loop at line 37 at least once by changing the loop condition to `j<extra_run+1` as shown in Figure 2. However, this ensures that the `mbedtls_md_process` function gets executed for both cases of padding validity. A local attacker can leverage the timing difference observable when the `mbedtls_md_process` function gets called for the first time to perform Lucky 13 attacks. Note that performing such attacks in the SGX setting becomes easier due to the strong attacker model as mentioned in Section 2.2. A possible fix is to replace the call to the `mbedtls_md_process` at line 38 with an indirect call that executes the specific compression function, which also gets executed from the update function, and, hence, does not serve as a timing observation point [3].

As a novel feature, ENCIDER detects this local timing side channel thanks to its ability to infer timing observation points, which appear on both the true and the false branches of a secret dependent branch and reveal the secret dependent timing difference to an attacker much earlier than the exit point of the vulnerable code. An important criterion for a timing observation point is for it to be discernible. ENCIDER leverages its path-sensitive analysis to identify callsites that execute a function for the first time and reports the callsite, if any, revealing the maximum timing difference for each secret dependent branch. Note that other approaches that detect timing side channels such as [29], [48], [49], [50] consider the end of the execution as the only type of timing observation point. As demonstrated through the example in Figure 2, code eliminating timing side channels for a remote attacker may still host a timing side channel that can be exploited by a local attack, which may be even more powerful in the SGX setting. Evolution of the `ssl_decrypt_buf` function demonstrates the difficulty of developing side channel free code and the importance of tools like ENCIDER that can detect multiple types of side channels as each fix to a side channel may end up introducing another type of side channel.

Figure 3 presents the architecture of ENCIDER, which detects both timing and cache side channels in cryptographic libraries, SGX enclaves, and SGX SDKs. The underlying side channel detection algorithms, which are the same for these different analysis targets, can be configured to perform more precise analysis by providing the relevant input specifications. As an example, for SGX enclaves and SGX SDKs, specifying `OCALL`s enables detection of timing side channels that are visible to the untrusted operating system or the application through these special APIs. Therefore, in addition to automatically inferring timing observation points for local attackers, ENCIDER can also leverage the programming model of SGX. Another important type of input specification has to do with the sensitive arguments of API functions.

ENCIDER uses dynamic symbolic execution as the underlying program analysis technique. The precise memory model provided by symbolic execution is leveraged to perform byte-precise labeling and precise tracking of high/low attributes as information flows. However, dynamic symbolic execution is known to have the path explosion problem, which limits the scalability of the analysis. ENCIDER deals with the path explosion problem by using the information flow and timing models of certain API functions. As an example, the local timing side channel can be detected faster by abstracting away the `mbedtls_cipher_crypt` function in Figure 2 by only modeling the key information-flow characteristic, i.e., that the array pointed by `dec_msg` (the fourth parameter) gets filled with secret data once the function returns successfully. So, assuming that the user specifies `dec_msg` as a high-security sensitive output of `mbedtls_cipher_crypt` as part of the API specification in

3. We have responsibly disclosed the local timing side channel to mbedTLS developers, who recently fixed the issue and issued CVE-2020-16150.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

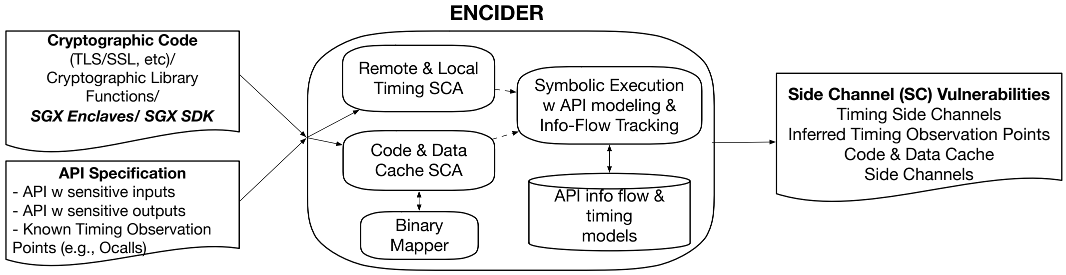


Fig. 3. The architecture of ENCIDER. Solid arrows denote data-flow and dashed arrows denote control-flow.

Figure 3, ENCIDER models the `mbedtls_cipher_crypt` function by tracking the sensitive information flow and detects the local timing side channel at least an order of magnitude faster than the case that also analyzes this function (see Section 6.2). This also enables modular analysis by focusing the side channel analysis on the code that uses the modeled API. To automate side channel analysis, ENCIDER uses under-constrained symbolic execution [51] to lazily initialize function arguments of arbitrarily complex data types and to avoid manually prepared test harnesses. This makes ENCIDER useful for developers as well as pentesters.

As shown in Figure 3, ENCIDER keeps a generic API model database, which stores the information flow and the timing models of APIs that get typically used by cryptographic libraries or SGX enclaves, such as Intel intrinsics instructions [52] and the SGX cryptographic API. The information flow models specify the specific regions of the API arguments that affect the specific region(s) of the output arguments. ENCIDER uses the information flow model at callsites of the modeled API functions to propagate high/low security sensitive labeling on regions of memory objects (see Section 6.2 for a more detailed discussion).

ENCIDER performs side channel analysis at the LLVM Intermediate Representation (IR) [53] level. Similar to other work on timing side channels [29], [50], it computes the cost of each path as the number of IR instructions; modeled API functions are, therefore, specified at the IR level. ENCIDER utilizes the API model database for all types of analysis targets. Finally, ENCIDER incorporates binary metadata into code based cache side channel analysis to compute the accuracy of analysis. We leverage the source location information at the IR level and at the binary level to compute a mapping between source lines and the virtual address regions. ENCIDER can be configured to report code cache side channels with an accuracy above a chosen threshold value to let developers focus on fixing more likely vulnerabilities. ENCIDER also reports secret dependent branches, timing observation points, and the branch target distinguishing points as illustrated in Figure 1 to address various capabilities of an SGX attacker.

5 Approach

In this section, we start with the adaptation of a well-established formulation of side channel freedom in Section 5.1. We explain the details of our novel side channel detection algorithm that can detect both timing and cache side channels in Section 5.2, and conclude with a discussion of the correctness in Section 5.3.

```
void foo(int L, int H)
 if (H > 0) {
    if (L < 1) {
       // ru1
    }
    else if (L < 5) {
       // ru2
    }
    else {
       // ru3
    }
 }
 else {
   if (L < 0) {
      if (H > -10) {
         if (L == -1) {
            // ru4
         }
         else {
            // ru5
         }
      }
      else {
         if (L == -1) {
            // ru6
         }
         else {
            // ru7
         }
      }
   }
   else {
      // ru8
   }
 }
}
```

Fig. 4. Sample code with low security (L) and high security (H) inputs.

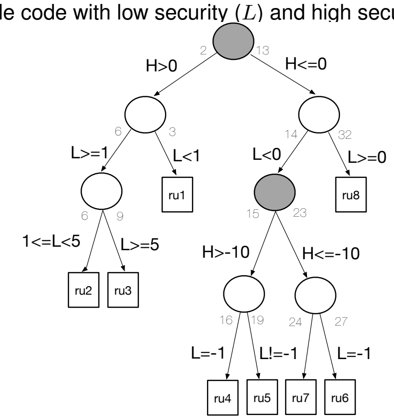


Fig. 5. Symbolic execution tree for the sample code in Figure 4. Circles represent the internal nodes and squares represent the leaves annotated with the resource usage. Filled circles represent the H-ancestors.

5.1 Side Channel Freedom

Secure information flow is often expressed in terms of the *non-interference* property [54], which informally states that any legal run in a system produces the same low outputs for the same low security inputs, regardless of the values of the high security inputs. To check non-interference property on a system, one needs to check every pair of runs in the system. This implies that non-interference is not a safety property [55], which can be checked by analyzing each run individually.

An important type of output a system may implicitly disclose is the usage of some computational resource such as the execution time or memory units such as cache lines. In the presence of an adversary that can monitor the resource usage, it is important that resource usage does not reveal high security inputs. An adaptation of the non-interference property to side channel freedom states that a system uses the same amount of resource for the same low security

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

inputs, regardless of the values of the high security inputs. This property is relaxed with a parameter ϵ by permitting resource deviations for the same low security inputs up to ϵ units.

We extend the well-established bounded side channel freedom formulation by introducing a resource usage difference computation function, $\mathcal{DIFF}^{\mathcal{T}}$, which is parameterized on the resource type T:

Definition 1 (Bounded Side Channel Freedom)**.** *Let $\mathcal{H}$ and $\mathcal{L}$ denote the sequence of high-security and low-security inputs of a program, $\mathcal{P}$, respectively, and $\mathcal{T}$ denote the resource type. Let $\mathcal{R}^{\mathcal{T}}(\mathcal{P},\mathcal{L},\mathcal{H})$ denote the amount of resource type $\mathcal{T}$ usage upon termination given the inputs $\mathcal{L}$ and $\mathcal{H}$. We use a domain $\mathcal{RU}^{\mathcal{T}}$ to denote the set of resource type $\mathcal{T}$ usages and a function $\mathcal{DIFF}^{\mathcal{T}} : \mathcal{RU}^{\mathcal{T}} \times \mathcal{RU}^{\mathcal{T}} \mapsto \mathbb{R}$ that quantifies the difference between two resource type $\mathcal{T}$ usages. A system is free of resource type $\mathcal{T}$ side channels iff*

$$\begin{aligned} \forall \mathcal{L}_1, \mathcal{L}_2, \mathcal{H}_1, \mathcal{H}_2.\ \mathcal{L}_1 = \mathcal{L}_2 \wedge \mathcal{H}_1 \neq \mathcal{H}_2 \rightarrow \\ \mathcal{DIFF}^{\mathcal{T}}(\mathcal{R}^{\mathcal{T}}(\mathcal{P},\mathcal{L}_1,\mathcal{H}_1), \mathcal{R}^{\mathcal{T}}(\mathcal{P},\mathcal{L}_2,\mathcal{H}_2)) < \epsilon \end{aligned} \tag{1}$$

Note that for some resource types, e.g., the execution time, resource usage can be computed per path and independent from the alternative paths and the difference in resource usage can be found from these independently computed values. However, for some resource types the resource usage difference needs to be computed relative to the alternative paths. For instance, to precisely detect the difference in code (instruction) cache line accesses, one needs to analyze all peer basic blocks[4] that would be executed by the alternative branches of a secret dependent branch [5]. So, the generic $\mathcal{DIFF}^{\mathcal{T}}$ function allows us to unify the side channel freedom formulation for both types of resources. In this work, we consider the resource types of execution time and code cache lines and leverage path sensitive analysis for partitioning the input space into equivalence classes and for computing relative resource usage.

5.2 Finding Resource Side Channels

Unlike the verification approaches that reduce verification of non-interference to safety verification using techniques such as self-composition [55], ENCIDER does not transform the program under analysis. Instead, ENCIDER analyzes the original program at the IR level by leveraging symbolic execution's capability to generate a decomposition of the input space. Since resource side channels involve the same low security inputs, one needs to analyze each equivalence class of paths w.r.t. low security inputs separately. We design ENCIDER to detect both timing and code cache side channels at the same time. For the former, resource usage corresponds to the duration of the computation whereas for the latter resource usage corresponds to the utilized cache lines.

Figure 4 shows a sample code with various conditional statements that check the high security variable `H` or the low security variable `L`. `rui` denotes the resource usage of path `i` and shown as the comment. Figure 5 shows the symbolic execution tree annotated with the branch conditions and the source lines of branches. To find resource side channels for this example, we do not need to compare `ru4` with `ru5` and `ru6` with `ru7` as they trivially satisfy the property given in Equation (1) by differing on the low security inputs. Also, `ru1`, `ru2`, and `ru3` do not need to be compared to each other as they have the same high security input leading to trivial satisfaction of the property. However, `ru1` and `ru4` need to be compared as they agree on the low security variable, `L == -1`, and differ on the high security variable as `H > 0` contradicts `-10 < H <= 0`.

Algorithm 1 An algorithm for computing resource usage and detecting data-independent interference using symbolic execution.

ComputeResourceUsage(P: Program, $rt \in$ {time, cache}, H: MemLoc, L: MemLoc, ϵ: $\mathcal{REAL}$, τ: $\mathcal{N}$)
  s_0: $SEState$; $s_0.RU \leftarrow \lambda hc.lc.term.undef$; $s_0.HA \leftarrow undef$;
  $s_0.HC \leftarrow true$; $s_0.LC \leftarrow true$;$s_0.term \leftarrow undef$;
  $s_0.ru \leftarrow \mathcal{INIT}(rt)$; $s_0.INST \leftarrow \emptyset$; $s_0.reached \leftarrow \emptyset$,
  $s_0.IM \leftarrow \lambda hc.lc.term.\emptyset$, $Paths \leftarrow \{s_0\}$, $HAnc \leftarrow \emptyset$
  Let s_0 denote the initial symbolic execution state/path for P
  Make H and L symbolic in s_0
  while τ seconds not elapsed and $Paths \neq \emptyset$ **do**
    $s \leftarrow$ **chooseNext**($Paths$)
    $s.succ \leftarrow$ **ExecuteNextInstruction**(s)
    $Paths \leftarrow Paths \cup s.succ \setminus \{s\}$
    for each $s' \in s.succ$ **do**
      $s'.ru \leftarrow \mathcal{EXT}(rt, s.ru, s.nextInst)$
    end for
    if $|s.succ| > 1$ **then**
      if $\exists s' \in s.succ. \exists new.$ such that $s'.PC \equiv s.PC \wedge new$ and $H \cap Loc(new) \neq \emptyset$ **then**
        $HAnc \leftarrow HAnc \cup \{s\}$
        $s.numSucc \leftarrow |s.succ|$
        for each $s' \in s.succ$ **do**
          Let $s'.PC \equiv s.PC \wedge new$
          $s'.HA \leftarrow s$; $s'.HC \leftarrow new \downarrow H$; $s'.LC \leftarrow true$
        end for
      else if $s.HA \neq undef$ **then**
        for each $s' \in s.succ$ **do**
          Let $s'.PC \equiv s.PC \wedge new$
          $s'.HA \leftarrow s.HA$; $s'.HC \leftarrow s.HC$; $s'.LC \leftarrow s'.LC \wedge new$
        end for
      end if
    end if
  end while
  for each $s \in Paths$ and s has terminated **do**
    PropagateAndDetectLeakage(s, rt, ϵ)
  end for

Algorithm 1 shows how to use symbolic execution to find resource side channels. We assume that each symbolic path is represented with a symbolic execution state that records the path condition, PC, and the program counter or the next instruction to execute, $nextInst$. We extend this minimal symbolic execution state representation with metadata needed for resource side channel analysis.

We distinguish states that branch on high security variables by storing them in the $HAnc$ set. For each execution state, we keep track of the H-ancestor, HA, which is the closest ancestor in the symbolic execution tree that branches on the high security variables. In Figure 5, H-ancestors are represented with the filled circles. Part of the path condition that relates to the high security variables is stored in HC representing the segment of the path condition that was

4. Two basic blocks are peer basic blocks if each gets executed as a different branch target of the same branch instruction.

5. Note that two alternative paths that emerge from the same secret dependent branch may have peer basic blocks due to multiple branches, some of which may implicitly depend on the secret.

TABLE 1
Specifics of computing resource usage based on the type of resource (T for execution time and C for code cache lines). ru and R denote the resource usage of a single path and a set of paths, respectively, and i, BB_0, and $BB(i)$ denote an IR instruction, the first IR basic block, and the IR basic block that contains the instruction i, respectively.

$\mathcal{T}$	ru	R	$\mathcal{INIT}$	$\mathcal{EXT}(ru,i)$	$\mathcal{JOIN}(R,ru)$
T	$\mathcal{N}$	Range $[min, max]$ $(undef = [\infty, -\infty])$	0	$ru + Cost(i)$	$[min(R.min, ru), max(R.max, ru)]$
C	$2^{\mathcal{N}}$	$2^{\mathcal{N}}$ $(undef = \emptyset)$	$\{BB_0\}$	$ru \cup \{BB(i)\}$	$R \cup ru$

introduced at the H-ancestor. Similarly, part of the path condition that does not involve high security variables is stored in LC representing the segment of the path condition that was introduced after the H-ancestor. We also keep a resource usage field, ru, representing the total amount of resource used so far on the path. For H-ancestor nodes we keep a resource usage map, RU, that records resource usage range for various combinations of HC and LC values that hold for its descendants.

The inputs to the algorithm consist of the program under analysis, P, the type of resource, rt, which can be time or cache, the memory locations that correspond to the high security and the low security variables, H and L, a time bound for symbolic execution, τ, and a resource bound, ϵ. In Algorithm 1, we represent resource usage computation as parameterized by the resource type. Specialization of the resource usage computation according to the resource type is explained in Table 1. We use $\mathcal{N}$ and $2^{\mathcal{N}}$ to denote the set of natural numbers and the power set of natural numbers, respectively.

The algorithm first applies symbolic execution to the program under analysis while recording H-ancestor, HC, and LC information for each node. For the initial node resource usage map is initialized to undefined for any possible combination of H relevant and L relevant constraint pairs, H-ancestor is set to undefined, HC and LC are set to true, and the resource usage, ru, is set to 0. Also, we keep a map, RU, from a combination of H relevant and L relevant constraint pairs and termination type to the range of resource usage that gets initialized (lines 2-4) as specified in Table 1 . Additionally, we keep metadata to detect generic and special timing observation points, which represent callsites that a local attacker can monitor and observe secret dependent timing differences. A function call is considered as a potential timing observation point if it has not been executed before on that specific path. We keep the set of functions reached on a path in the *reached* field (initialized at line 4) so that we can identify such callsites.

Until the time bound is reached or there are no more active paths, Algorithm 1 chooses the next symbolic execution path from the set of active paths, executes the next instruction, updates the resource usage, and keep tracks of the metadata (lines 8-30). When there is no branching, the execution state preserves the current values of the metadata except for the set of reached functions and whether the state is a candidate timing observation point. However, when there is branching, it checks if it is H relevant (D1 in Figure 1). If so, it updates the H-ancestor of its successors to the parent, sets HC to the projection of the branching condition

Algorithm 2 An extension to the baseline symbolic execution of an instruction for collecting timing observation point candidates.

1: **ExecuteNextInstruction**(s: $SEState$)
2: **if** $s.HA \neq undef$ **then** $s.INST \leftarrow s.INST \cup \{s.nextInst\}$
3: **end if**
4: $TO \leftarrow \emptyset$
5: **if** $s.nextInst$ is a call instruction **then**
6: Let cs denote the callsite to be executed by $s.nextInst$
7: **if** $cs.function \notin s.reached$ and $s.HA \neq undef$ **then**
8: Let s'' denote a copy of s that gets terminated at $s.nextInst$
9: $s''.term \leftarrow StackTrace(cs)$
10: $TO \leftarrow \{s''\}$
11: **end if**
12: $s.reached \leftarrow s.reached \cup \{cs.function\}$
13: **else if** $s.nextInst$ is a path terminating return instruction **then**
14: $s.term \leftarrow exit$
15: **end if**
16: **return ExecuteNextInstructionBaseline**(s) $\cup$ TO

on the H variables, and sets LC to true (lines 16-22). If it is not H relevant branching, the algorithm considers whether the branching node has an H-ancestor. If so, it updates LC by conjoining it with the branching condition (lines 23-28). If the branching node does not have an H-ancestor then it does not need to make any updates as there are no deviations w.r.t. the H variables on the execution tree.

Figure 5 shows the symbolic execution tree for the sample code in Figure 4. Nodes are labeled with a pair of source line numbers that correspond to the related branch statements. Nodes (2,13) and (15,23), illustrated as filled circles in Figure 5, perform H-branching. Node (2,13) is the H-ancestor of nodes (6,3), (6,9), (14,32), (15,23), ru1, ru2, ru3, and ru8. Node (15,23) is the H-ancestor of nodes (16.19), (24,27), ru4, ru5, ru6, and ru7. HC for ru3 is $H > 0$ and HC for ru7 is $H \leq -10$. LC for ru3 is $L \geq 1 \wedge L \geq 5$ and LC for ru5 and ru7 is $L \neq -1$.

Once the symbolic execution stage terminates, Algorithm 1 executes Algorithm 3 to propagate resource usage information from the leaf nodes that represent terminated symbolic execution paths to the H-ancestors and to detect interferences (lines 31-33).

5.2.1 Identifying Timing Observation Points

Algorithm 2 shows how we extended the logic for executing an instruction symbolically to detect code locations that are possible timing observation points (D4 in Figure 1). If the executed instruction is on a secret-dependent branch, it records the instruction for that state (lines 2-3). If the instruction to be executed is a call instruction and the callee is a function that has not been reached on the current path (lines 5-7), we clone the current path in s'', which gets terminated at that callsite, and mark it as a candidate timing observation point by recording the context of the callsite in $term$, which represents the termination point for each path (line 9). Note that paths that terminate regularly, i.e., due to executing the return instruction of the entry function, are represented with the generic $exit$ token (lines 13-14). The set of functions reached in the current path gets updated (line 12) and the instruction is executed using baseline symbolic execution and the successors are returned along with the cloned state s'' (line 16).

Algorithm 3 An algorithm for propagating resource usage of a path in the symbolic execution tree and incrementally checking resource usage leaks.

1: **PropagateAndDetectLeakage**(s: $ExecutionState$, rt $\in$ {time, cache}, ϵ: $\mathcal{REAL}$)
2: **if** $s.HA \neq undef$ **then**
3: $a \leftarrow s.HA$
4: **if** $s \notin HAnc$ **then**
5: Let key denote $(s.HC, s.LC, s.term)$
6: $a.RU[key] \leftarrow \mathcal{JOIN}(rt, a.RU[key], s.ru)$
7: $source[key] \leftarrow s$
8: $a.IM[key] \leftarrow a.IM[key] \cup s.INST$
9: **else**
10: **for** each $hc, lc, term$ s.t. $s.RU[(hc, lc, term)] \neq undef$ **do**
11: Let key denote $(hc, lc, term)$
12: $a.RU[(s.HC \wedge hc, s.LC \wedge lc, term)] \leftarrow s.RU[key]$
13: $source(s.HC \wedge hc, s.LC \wedge lc, term) \leftarrow s$
14: $a.IM[(s.HC \wedge hc, s.LC \wedge lc, term)] \leftarrow s.IM[key]$
15: **end for**
16: **end if**
17: **if** all terminated descendants of a have been processed **then**
18: **if** $\exists h_1, h_2. \exists l_1, l_2. \exists t_1, t_2. source(h_1, l_1) \neq source(h_2, l_2) \wedge$
19: $h_1 \not\equiv h_2 \ \wedge \ l_1 \wedge l_2 \neq false \ \wedge \ t_1 = t_2$ **then**
20: $topset_1 \leftarrow a.IM(h_1, l_1, t_1)$
21: $topset_2 \leftarrow a.IM(h_2, l_2, t_2)$
22: **if** rt is time **then**
23: **if** $diff(a.RU(h_1, l_1, t_1), a.RU(h_2, l_2, t_2)) \geq \epsilon$ **then**
24: **if** $t_1 = exit$ **then**
25: print timing leakage at $a.progLoc$
26: **else**
27: print leakage at timing obs. point t_1 (t_2)
28: **end if**
29: print $|topset_1 \cap topset_2|$ as # of timing observation points
30: **end if**
31: **else**// rt is cache
32: print $|(topset_1 \setminus topset_2) \cup (topset_2 \setminus topset_1)|$ as # of branch target distinguishing points
33: **if CacheDiff**($a.RU(h_1, l_1, t_1)$, $a.RU(h_2, l_2, t_2)$) $\geq \epsilon$
34: **CacheDiff**($a.RU(h_2, l_2, t_2)$, $a.RU(h_1, l_1, t_1)$) $\geq \epsilon$ **then**
35: print code based cache leakage at $a.progLoc$
36: **end if**
37: **end if**
38: **end if**
39: **PropagateAndDetectLeakage**(a, rt, ϵ)
40: **end if**
41: **end if**

5.2.2 *Propagating Resource Usage and Detecting Interference*

Algorithm 3 propagates resource usage bottom-up over the symbolic execution tree focusing on the leaf nodes (terminated paths) and all H-ancestors reachable from them. The algorithm is recursive as there may be multiple H-ancestor type nodes on a symbolic execution path. So, in addition to propagating resource usage from the leaf nodes to their H-ancestors, resource usage ranges at any H-ancestor type node need to be propagated to their H-ancestors, and so on, until the algorithm reaches the top level H-ancestor.

The inputs to Algorithm 3 consist of the symbolic execution path, s, the type of resource, rt, which can be time or cache, and the bound ϵ. If s is not an H-ancestor then we need to propagate the resource usage of s, i.e., $s.ru$, to its H-ancestor (lines 4-8). Basically, the RU map of the H-ancestor is updated by updating the resource usage range for the combination of the constraints and the termination location $(s.HC, s.LC, s.term)$ to include resource usage of s (line 6). Unlike the leaf nodes of the symbolic execution tree, symbolic execution states of H-ancestor type nodes need to propagate all possible resource usages due to various combinations of H relevant and L relevant constraints that have been propagated by their descendants. Therefore, it propagates each resource usage combination corresponding to a different constraint and termination location combination by updating both the H relevant and L relevant components with $s.HC$ and $s.LC$, respectively (lines 10-15).

Algorithm 4 An algorithm for detecting code based cache side channels.

1: **CacheDiff**($bbset_1$, $bbset_2$: Set of Basic Blocks): $\mathbb{R}$
2: Let $diff \leftarrow bbset_1 \setminus bbset_2$
3: Let $BinMap$: $SourceLocInfo \rightarrow 2^{Range}$ denote source info to virtual address range mapping
4: $accuracy \leftarrow 0$
5: **for** $bb \in diff$ **do**
6: Let $siblings = \{bb' \mid bb'' \in pred(bb) \wedge bb' \in succ(bb'') \cap bbset_2\}$
7: $s_1 \leftarrow \{r \mid i \in bb \ \wedge \ r \in BinMap(i.source)\}$
8: $s_2 \leftarrow \{r \mid bb' \in siblings \ \wedge \ i \in bb' \ \wedge \ r \in BinMap(i.source)\}$
9: $mismatch \leftarrow 0$
10: **for** $var_1 \in s_1$ **do**
11: **for** $var_2 \in s_2$ **do**
12: **if** $\exists va.\ var_1.min \leq va \leq var_1.max \wedge CacheLine(va) \notin CacheLines(var_2)$ **then**
13: $mismatch \leftarrow mismatch + 1$
14: **end if**
15: **end for**
16: **end for**
17: **if** $mismatch/(s_1.size \times s_2.size) > accuracy$ **then**
18: $accuracy \leftarrow mismatch/(s_1.size \times s_2.size)$
19: **end if**
20: **end for**
21: **return** $100 \times accuracy$

Once all descendants of an H-ancestor type node have propagated their resource usages (line 17), Algorithm 3 checks for resource usage deviations for every pair of equivalence classes whose constraints differ w.r.t. their H constraints, intersect w.r.t. their L constraints, and agree on the termination locations. For time side channels (lines 22-30), it uses the resource usage ranges to detect the variation in resource usage. If the two ranges differ at least ϵ units (line 23), it reports the leakage along with the source code location for the H-ancestor type node a. By checking the type of termination location, our approach can report timing leakages that can be locally observed at timing observation points, i.e., $term$ is not $exit$ (lines 26-27). It also reports the number of timing observation points (line 29). For code based cache side channels (lines 31-37), it executes **CacheDiff** (see Algorithm 4), which is explained below, to check if there exists a cache line that can be used to leak information about the high security sensitive inputs with at least ϵ accuracy. It also prints the number of branch target distinguishing points (line 32). Another action taken by the algorithm when all descendants of an H-ancestor type node have been processed is to propagate its resource usage ranges to its H-ancestor, if any, via recursion (line 33).

To minimize the number of comparisons, Algorithm 3 records a `source` information for (HC, LC) pairs. As an example, `ru4` and `ru6` would get compared to each other when all descendants of the H-ancestor (15,23) in Figure 5 gets processed. To avoid their comparison again at node (2,13), we track the source information. Initially, the source information refers to the identifier for the leaf node that corresponds to the (HC, LC) (line 7). At the time that an H-ancestor gets visited, resource usage for its descendants

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

have already been made as this is performed when its last descendant gets traversed. So, in addition to propagating its resource usage map to its H-ancestor, Algorithm 3 also updates the source information for every (HC, LC) to its identifier so that these pairs do not get compared to each other at its H-ancestor (line 13). Finally, we avoid redundant comparisons by comparing the source information for the candidate (HC, LC) pairs when resource leakage is checked (line 18).

5.2.3 *Finding Cache Side Channels*

ENCIDER detects two types of cache side channels: those that involve the instruction cache due to executing different code locations for different secret values (D2 and D3 in Figure 1) and those that involve secret dependent data accesses (D5 and D6 in Figure 1)[6]. The former is implemented as part of the unified resource side channel Algorithms 1 and 3. We first present detecting code based cache side channels and then explain detection of data cache side channels.

5.2.3.1 Code based cache side channels: Algorithm 4 detects code based cache side channels and takes as input two sets of basic blocks, where each set represents one of the two symbolic execution paths with a common H-ancestor, and a bound, ϵ, which represents the accuracy of computing mismatching cache lines that can reveal information about the high security sensitive inputs. The algorithm first computes the difference between the two sets (line 2) and for each basic block in the difference set it computes the sibling basic blocks (line 6) using the predecessor ($pred$) and the successor ($succ$) functions defined over the basic blocks in a control-flow graph. Sibling basic blocks of a basic block represent those that would be executed as different targets of a branch instruction than that basic block. Note that some basic blocks may not have siblings if the predecessor is a basic block with an unconditional branch instruction. However, there must exist at least one basic block in $diff$ such that the corresponding sibling set is not empty.

The algorithm leverages the source location information in the IR and uses that to find out potential mappings to virtual address ranges using a source location to virtual address range mapping by considering source location information about individual instructions in the basic blocks (lines 3, 7, and 8). So the algorithm checks every pair of virtual address ranges that correspond to the basic block in $diff$ and those that correspond to those in $siblings$, to see if there is a mismatch in terms of the cache lines (line 12). If so, it increments the number of mismatches. We compute the accuracy as the ratio of mismatches over the total number of pairs of virtual address ranges and record the maximum one (lines 17-19) among all the basic blocks in $diff$. Finally, the algorithm either returns 0 if no cache line difference or returns a non-zero accuracy value of a cache line mismatch. We provide more details about computing the cache lines below and those about generating a mapping from the source locations to virtual address ranges in the Appendix.

6. Note that the mask used in cache line computation can be configured for page granularity to detect leakages that can be exploited via page access monitoring

5.2.3.2 Data based cache side channels: ENCIDER also checks for data based cache side channels that can be leveraged in an access-based attack, which monitors accesses to specific cache elements. Modern microprocessor architectures are equipped with several layers of caches. In Intel architectures, the L1 cache serves a single core and is divided into an instruction cache and a data cache. The L2 cache also serves a single core while unifying the instruction and data. The L3 cache is shared by all cores, unifies the instruction and data, and is inclusive of the L1 and L2 cache.

To perform a cache based attack, one needs to consider the specifics of the cache hierarchy, the cache placement policy such as direct mapped or set associativity and the cache size. Incorporating these details during side channel analysis incorporates an unnecessary overhead to achieve precise results. To strike a balance between efficiency and usefulness, ENCIDER checks whether a secret dependent address can compute two different indices that can be used in cache placement. ENCIDER represents all secrets as symbolic. So, similar to the approach of `CacheS` [30], it first checks if an address involves secret dependent data. If so, such an address is a formula, denoted by $A[H]$, where H represents the vector of secret variables. ENCIDER uses a mask, M, as wide as the address size and checks for satisfiability of the following formula:

$$A[H] \;\&\; M \neq A[H'] \;\&\; M \qquad (2)$$

where H' is a renaming of the vector H and $A[H']$ is rewriting of the secret dependent address with the new versions of the secret variables. Unlike `CacheS` [30], we do not abstract symbolic expressions and we precisely track flow of both secret and public data including the expressions that can be used as array indices.

Similarly, given a virtual address V, we compute the corresponding cache line as $V \;\&\; M$ for code based cache side channel analysis. The flexibility of using a mask can be realized through the fact that it can detect placement at the page and cache line granularities as well as at a smaller granularity within the cache line to assist detection of CacheBleed [16] like side channels.

We have implemented ENCIDER on top of the KLEE [56] symbolic execution engine, which analyzes LLVM bitcode. In LLVM IR, an indirect memory access operation consists of a `getElementPtr` (GEP) instruction followed by a load instruction. The role of the GEP instruction is to compute the address expression, which is used by the subsequent load instruction as the operand. ENCIDER intercepts the GEP instructions and analyzes the computed address expression using Equation 2 to check for secret dependent memory accesses.

ENCIDER achieves precise information-flow tracking by labelling high/low regions in memory objects and propagating these labels during symbolic execution. However, ultimately, we need to decide whether a given symbolic expression has any high/low information. Specifically, we need to be able to do this for branch conditions to check for timing side channels, and for derived memory addresses computed by the GEP instructions to check for data cache side channels.

Given an arbitrarily complex symbolic expression, we traverse the abstract syntax tree (AST) of the symbolic ex-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

pression and map each subexpression to a set of high or low regions. If this process returns an empty set then it means the symbolic expression does not carry the label of interest (high or low). The base case of this recursive traversal is a read expression or array access/read at a specific offset, which translates to a singleton of high/low range, if any, or an empty set. If the offset is a symbolic expression then ENCIDER conservatively assumes that the range spans the whole memory object. Symbolic offsets are possible for array accesses[7]. In our experience, this has not led to any false positives as in our benchmarks the whole array was meant to be labeled as high security sensitive.

Complex cases of the AST traversal include a binary or a unary operator, for which we combine the set of ranges computed for each operand based on the semantics of the operator. As an example, assume that the $privatekey$ is 256 bits and the whole region is marked as high sensitive: [0,255] and that we need to map the left shift expression $(read\ w256\ 2\ privatekey) \ll 4$ to the set of high sensitive regions if any. Here, $w256$ denotes the width of the expression being 256 and 2 is the offset at which the read starts. The sensitive region that corresponds to $(read\ w256\ 2\ privatekey)$ is [2,255] and after applying the semantics of the left shift operation, we would get [6,255] as the high sensitive region in $(read\ w256\ 2\ privatekey) \ll 4$.

5.3 Correctness

In this section, we discuss the correctness of our approach. We assume that resource usage of an instruction is not data dependent. This is a simplifying assumption rather than a claim as certain instructions, such as the floating point operations, may incur different overheads for different operand values. However, this assumption has been made by all related work that work at the IR level [29], [48], [49], [50].

Definition 2. *Given a symbolic execution path P, the data independent resource usage of P is determined by the set of instructions executed by P and not by the operands of those instructions.*

Definition 3. *Two symbolic execution paths P_1 and P_2 interfere if the path conditions of P_1 and P_2 agree on the low security variables, differ on the high security variables, and the data-independent resource usage differs by at least ϵ.*

Claim 1. *For a given program, Algorithm 1 generates data independent resource usage for every feasible combination of high security and low security constraints that have been explored by the underlying symbolic execution engine.*

Proof. Follows from the fact that resource usage is computed for every leaf node in the symbolic execution tree that represent terminated paths with feasible constraints. □

Claim 2 (Conditional Completeness)**.** *If two symbolic execution paths P_1 and P_2 interfere then Algorithm 1 detects the interference provided that these paths are explored to completion by the underlying symbolic execution engine.*

7. Note that accesses to struct fields involve constant offsets computed by the compiler based on the struct layout

Proof. We prove this claim by contradiction; assume that equation 1 does not hold and Algorithm 1 fails to detect the interfering paths, p_1 and p_2, despite these paths being explored by the underlying symbolic execution engine. Since there is an interference, there must be a secret dependent branch and, hence, at least one common H-ancestor of p_1 and p_2 due to line 21 of Algorithm 1. Let s denote the common H-ancestor that is closest to p_1 and p_2, i.e., lowest in the symbol execution tree. Let C_i denote the condition that there are no H-ancestors between the leaf node p_i and s, for $1 \leq i \leq 2$. There are four cases regarding s, p_1, and p_2: *Case 1: Both C_1 and C_2 hold.* For both p_1 and p_2, the respective (HC, LC) combinations are reflected to RU map of s through the lines 5-7 in Algorithm 3. *Case 2: C_1 holds and C_2 does not hold.* p_1's (HC, LC) combination would be propagated to the RU map of s as in Case 1. p_2's (HC, LC) combination would get propagated first to the closest H-ancestor. Then the RU map of each intermediate H-ancestor would be propagated in a consistent way with its H-ancestor's branching conditions until the (HC, LC) combination of p_2 gets propagated to RU map of s (lines 9-12 in Algorithm 3). *Case 3: C_1 does not hold and C_2 holds.* (Similar to Case2). *Case 4: Neither C_1 nor C_2 hold.* Both p_1's and p_2's (HC, LC) combinations are ultimately get propagated to the RU map of s as explained in Case 2. Since the (HC, LC) combinations of p_1 and p_2 will be propagated to the RU of s in each of the four cases and along with their resource usage (see Claim 1) and since p_1 and p_2 do interfere, this would be detected by Algorithm 3 after all terminated descendants of s get processed (lines 15-17), leading to a contradiction. □

6 Implementation

To implement ENCIDER, we have extended the KLEE [56] symbolic execution engine in several dimensions to precisely track information flow and to apply security sensitivity aware API modeling.

6.1 Information-Flow Tracking

Side channel analysis is an information-flow tracking problem. Symbolic execution facilitates the tracking of information-flow when the memory regions that carry the information of interest are labeled as symbolic. However, in real-world applications the high security sensitive and the low security sensitive information are mixed inside the same data structure.

The underlying symbolic execution engine, KLEE, uses a single block of memory region to represent an aggregate data type such as `bolos_persistent_context_t` that is shown in Figure 6.

In KLEE a memory region is marked symbolic as a whole and a unique identifier is used to refer to the symbolic region. However, since the memory region is represented as an array of bytes, accesses to individual fields are represented as array accesses. To precisely label high and low sensitive inputs in the KLEE memory model, ENCIDER expects the user to specify high and low regions using a byte offset and the size within the data type. As an example, the offset & size tuple (72,32) refers to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

TABLE 2
Example Intel intrinsic instructions and their modeling in terms of information flow.

Intel Intrinsic	Semantics	Information Flow Specification in ENCIDER
_mm_loadu_si128	$dst[127:0] \leftarrow MEM[mem\_addr + 127 : mem\_addr]$	$H(\&dst, 0, 127) \leftarrow H(mem\_addr, 0, 127)$
_mm_xor_si128	$dst[127:0] := (a[127:0]\ XOR\ b[127:0])$	$H(\&dst, 0, 127) \leftarrow H(\&a, 0, 127) \cup H(\&b, 0, 127)$

```
struct cx_ecfp_private_key_s{
 cx_curve_t   curve; // low
 int           d_len; // low
 unsigned char d[32]; // high
};
typedef struct cx_ecfp_private_key_s
        cx_ecfp_private_key_t;
typedef struct bolos_persistent_context_s {
 // high
 uint8_t deviceWrappingKey[...];
 // mixed
 cx_ecfp_private_key_t endorsement_private_key1;
 // high
 uint8_t endorsement_private_key1_hash[32];
 // mixed
 cx_ecfp_private_key_t endorsement_private_key2;
 ...
} bolos_persistent_context_t;
```

Fig. 6. A sample data structure from the Ledger enclave with mixed (high and low) security sensitive fields.

endorsement_private_key1_hash and is marked as high whereas the offset & size tuple (26,4) refers to the d_len field of the endorsement_private_key1 and is marked as low.

ENCIDER uses two types of specifications for security sensitivity. Type 1 specifies the high and low regions in a given data structure type. Type 2 specifies high and low arguments of function parameters. The user can choose one of the three labels: high, low, and mixed. For pointer arguments, if the sensitivity is high or low, that sensitivity gets applied to all memory regions that can be reached through dereferencing that pointer for any depth in the dereferencing expression[8]. If the sensitivity is mixed then, the user must have provided the type sensitivity specification, which gets used by ENCIDER to decide how to treat various regions of the memory object(s) that can be reached from the pointer. Assuming that the user provides a correct and complete specification for Type 1 and Type 2 specifications and that the sensitivity specification of a type applies to all instances of that type, ENCIDER keeps track of byte-precise sensitivity of the memory regions that get created for each execution path and propagates them as symbolic execution proceeds.

6.2 API Modeling

To tame the path explosion problem in symbolic execution, ENCIDER uses user specified models of API functions. An API function can be modeled in three ways: 1) by implementing a model function in C and with the same signature as the original function, 2) as a side-effect free function that returns a symbolic value, if the return type is not a void, and 3) by defining its information flow. ENCIDER handles callsites that involve modeled functions based on the type of the model. Case 1) is handled by calling the model function instead of the original function when handling all the relevant callsites. Case 2) is handled by treating the callsite as a no-op while copying a fresh symbolic value to the register that holds the return value. Case 3) is handled by applying the information flow specification.

The idea with information flow modeling is to capture potential flow of high security sensitive bytes as a result of executing a modeled API function. Type 3 specification identifies which memory regions of the arguments flow into which memory regions of the return value. We have analyzed the functional specification of 262 Intel intrinsics instructions [52] and manually modeled the information flow for each of these instructions. Table 2 shows modeling of some of the Intel intrinsics instructions. $H(A, min, max)$ denotes the set of high sensitive ranges $[r_1, r_2]$ at memory address A such that $min \leq r_1 \leq r_2 \leq max$ or an empty range if none of the bits at address A are high sensitive. We define a similar map L for low sensitive data.

7 Evaluation

We have applied ENCIDER to various real-world SGX enclave implementations, SGX crypto APIs, and other widely-used crypto libraries. We have run our experiments on an Intel Xeon CPU 2.30GHz with 256 GB memory. We have set ϵ to 0. A summary of results is provided in Table 3. ENCIDER is able to detect new timing and code as well as data based cache side channels. We also applied ENCIDER to the benchmarks of three state-of-the-art side channel analysis tools, ct-verif[9] [29], CacheS [30], and DATA [31]. In addition to detecting the side channels discovered by CacheS, ENCIDER can also detect new types of side channels in those benchmarks.

7.1 New Side Channels found by ENCIDER

7.1.1 Timing Side Channels

ENCIDER detects a timing observation point that can be leveraged in a local timing side channel attack in mbedTLS-SGX, which is discussed in Section 4 in detail. ENCIDER detected similar timing observation points in Amazon's s2n library (at line 4 in Figure 7) and openSSL 1.0.1c (at line 7 in Figure 8). So, ENCIDER automatically detected timing observation points in three different TLS implementations that can be exploited by a local attacker to perform the

8. Note that it is possible that this may lead to inconsistencies in terms of sensitivity labeling. In such cases, the user should use the mixed label and let ENCIDER use the sensitivity specification of the type information, if provided by the user and if possible, i.e., the sensitivity specification of the type applies to all instances of that type, to precisely track the sensitivity information.

9. We have obtained the ct-verif benchmarks from [57].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

TABLE 3
A summary of vulnerabilities detected by ENCIDER: timing side channels (TSC), code based cache side channels (CCSC), and data based cache side channels (DCSC). **TOPS** and **BTDP** denote maximum number of timing observation points and branch target distinguishing points, respectively. ★ denotes new vulnerabilities detected by ENCIDER, ♢ denotes vulnerabilities detected exclusively by ENCIDER but fixed recently, ♣ denotes new types of vulnerabilities detected by ENCIDER but not by CacheS, ♡ denotes additional vulnerabilities detected by ENCIDER, □ denotes known vulnerabilities detected by both ENCIDER and CacheS.

Case Studies	# Func.	# TSC	# CCSC	# DCSC	TOPS	BTDP	Time (secs)		Memory (MB)		Coverage (%)	
							Min	Max	Min	Max	Min	Max
mbedTLS-SGX	1	1 ★	1 ★	1 ★	93	147	5014.97	5014.97	410.42	410.42	49.80	49.80
Signal Enclave	7	0	1 ★	1 ★	13	57	2.82	500.41	29.90	436.85	10.54	80.00
Ledger Bolos Enclave	8	1 ♢	0	0	46	70	0.81	50.28	35.92	149.25	22.09	95.45
Tresor SGX	4	0	0	0	0	0	15.07	51.01	23.11	68.98	78.64	83.78
SGX IPP	8	1 ♢	0	0	784	613	21.45	501.60	229.02	403.97	48.43	73.47
SGX SSL	9	1 ★	1 ★	0	423	398	56.27	589.01	15.36	678.58	21.69	41.94
openSSL 1.0.1c	1	1 ★, 2 ♢	0	1 ♢	155	173	2555.27	2555.27	2262.24	2262.24	81.25	81.25
s2n	1	1 ★ 1 ♢	1 ★	2 ★	226	422	475.94	475.94	654.32	654.32	78.57	78.57
polarssl	1	4 ♣	4 ★	3 ★	38	63	526.77	526.77	209.31	209.31	72.32	72.32
libgrcypt	4	6 ♣	7 ♣	6 □	203	417	3.48	5241.25	16.08	759.39	82.03	100.00
openSSL	3	6 ♣	8 ♣	4 ♡ 9 □	89	129	2.37	796.14	56.94	579.80	47.50	94.74
mbedTLS	3	4 ♣	4 ♣	19 □	156	132	50.50	999.92	130.86	537.27	83.85	100.00
Total/Overall	50	29	27	46	784	613	0.81	5014.97	5.26	2262.24	10.54	100.00

```
1 int s2n_verify_cbc(struct s2n_connection *conn, struct
2 s2n_hmac_state *hmac,struct s2n_blob *decrypted) {  ...
3  s2n_hmac_update(hmac,decrypted->data,payload_length);
4  s2n_hmac_digest_two_compression_rounds(hmac,...);
5  ...
6  s2n_constant_time_equals(decrypted->data +
7    payload_length, check_digest, mac_digest_size) ^ 1;
8
9  s2n_hmac_update(copy, decrypted->data +
10   payload_length + mac_digest_size, decrypted->size -
11   payload_length - mac_digest_size - 1);
```

Fig. 7. ENCIDER detects a timing observation point at line 4 and secret dependent memory accesses at lines 6-7 (79) and 9-10 (82) in s2n 0.9 . payload_length is secret dependent.

```
1 static int ssl3_get_record(SSL *s) {
2   enc_err = s->method->ssl3_enc->enc(s,0);
3   ... // may set decryption_failed_or_bad_record_mac
4   i=s->method->ssl3_enc->mac(s,md,0);
5   ... // may set decryption_failed_or_bad_record_mac
6   if (decryption_failed_or_bad_record_mac) {
7             ... SSLerr(...); ...
```

Fig. 8. ENCIDER detects a timing observation point at line 7 in openSSL 1.0.1c.

Lucky 13 attacks. This suggests constant time solutions, e.g., the one implemented in openSSL in later versions [58], are more secure than those that add equalization code, e.g., s2n and mbedTLS, as even though the remote timing attacks are thwarted with the latter approach, local timing attacks may still remain feasible and even more so in the SGX setting, e.g., mbedTLS-SGX. ENCIDER detected a local timing side channel in the sgx_rsa3072_sign function in SGX IPP library with multiple timing observation points. The side channel is due to the secret dependent branch in the ippsSet_BN function, which repeatedly checks the last byte of a big number that happens to be the private key when called from sgx_rsa3072_sign to compute the actual length in a macro called FIX_BN: `for( ((srcLen)>1) && (0==(src)[(srcLen)-1]); (srcLen)--)`. Functions that serve as timing observation points include cpSizeof_RSA_privateKey1 and ippsRSA_GetSizePrivateKeyType1.

We have found a new timing side channel in one of the functions of the cryptographic library libsecp256k1 [59]. A secret dependent composite condition (line 5 in Figure 9) gets translated into branch instructions by the clang compiler. For an error case, i.e., a return value of zero, the timing difference reveals whether the failure was due to

```
1 static int secp256k1_scalar_set_b32_seckey(
2  secp256k1_scalar *r, const unsigned char *bin) {
3  int overflow;
4  secp256k1_scalar_set_b32(r, bin, &overflow);
5  return (!overflow) &&  (!secp256k1_scalar_is_zero(r)); }
```

Fig. 9. The timing side channel that was found by ENCIDER in a cryptographic API used by the Ledger Enclave involves the composite condition at line 4.

an overflow or the scalar value being zero. This function gets called from the exchange ecall in the Ledger Enclave, which is designed to host blockchain and cryptocurrency applications that are developed for the BOLOS operating system [4].

ENCIDER detected the remote timing side channels in the TLS implementations of openSSL, s2n and polarsll, which can be exploited by the Lucky 13 attacks [10]. ENCIDER reported that some of the secret dependent branches that were reported by CacheS [30] do lead to timing side channels.

```
1 sgx_status_t sgxsd_enclave_remove_pending_request(...) {
2   uint64_t pending_request_count_mask = ((uint64_t){1} <<
3     g_sgxsd_enclave_pending_requests_table_order) - 1;
4 sgxsd_pending_request_t *p_found_pending_request =
5     &g_sgxsd_enclave_pending_req[p_pending_request->id_val &
6     pending_request_count_mask];
7 if (p_found_pending_request->id_val ==
8           p_pending_request->id_val) { ... } ...
9 }
```

Fig. 10. ENCIDER detects a code based cache side channel due to the secret dependent branch at lines 7 and 8 and a cache side channel at lines 5-6 in the Contact Discovery Service.

Table 3 shows the maximum number of timing observation points for each benchmark. We observe that when one of the targets of the secret dependent branch is an error case that gets handled immediately, e.g., by returning an error code, there are fewer number of timing observation points for a local attacker to exploit as in the case of Signal Enclave. On the other hand, when different targets of a secret dependent branch have long common computations, e.g., s2n_verify_cbc, the number of timing observation points increases.

7.1.2 Code based Cache Side Channels

ENCIDER detected a code based cache side channel in the Signal Enclave. Signal [3] implements its Private Contact Discovery Service using Intel SGX, securing user's contact information within an enclave from service operators. Figure 10 shows the code snippet that has a secret dependent branch at lines 7-8. Leveraging the binary metadata that maps source line information to virtual address ranges in Algorithms 1-4, ENCIDER computes 100% accuracy of this cache side channel.

Analyzing the SGX SSL API using openSSL 1.1.0.j, revealed a code based cache side channel in the implementation of the `sgx_ecc256_compute_shared_dhkey` function. The vulnerability, as shown in Figure 11, is due to the `BN_lebin2bn` function which tries to detect the leading zero bits in a given big number, which turns out to be the private key when it gets called from `sgx_ecc256_compute_shared_dhkey`. ENCIDER reports this vulnerability with an accuracy of 33.33%. We realized that another OpenSSL function that may get called in an alternative implementation of `BN_lebin2bn` inside SGX SSL has the same type of side channel. Also, we found out other SGX APIs, which pass secret values to these functions and, hence, vulnerable to the same type of side channels: `sgx_ecc256_compute_shared_point`, `sgx_ecc256_calculate_pub_from_priv`, and `sgx_create_rsa_priv2_key`.

```
1 BIGNUM *BN_lebin2bn(const unsigned char *s, int len,
2 ...) { ...
3     s += len;
4     /* Skip trailing zeroes. */
5     for ( ; len > 0 && s[-1] == 0; s--, len--)
6         continue;
```

Fig. 11. The code based cache side channel that was found by ENCIDER due the secret dependent branch at line 5.

Additionally, ENCIDER reports code based side channels for some of the secret dependent branches that were found by CacheS in all the three libraries. For these vulnerabilities, the number of vulnerabilities and the accuracy ranges are as follows: libgrcypt 1.6.1 (7): [50%,100%], openSSL 1.0.2.f (8): [100%,100%], and mbedTLS 2.5.1 (4): [75%,75%].

```
1 ssl->in_msglen-=(ssl->transform_in->maclen+padlen); ...
2 memcpy(tmp,ssl->in_msg+ssl->in_msglen,MAX_MAC_SIZE);
```

Fig. 12. The secret dependent memory accesses detected by ENCIDER in polarssl 1.2.4 at line 2 (1437). `padlen` is secret dependent.

Table 3 shows the maximum number of branch target distinguishing points. We observe that if the code performs various checks on a value that gets computed based the on secret dependent condition, the number of branch target distinguishing points increases. For instance, `srcLen`, which is computed by the `FIX_BN` macro in the SGX IPP library, is checked for additional cases after the secret dependent branch location and has the highest value among our benchmarks.

7.1.3 *Data based Cache Side Channels*

ENCIDER found a true cache side channel in the `sgxsd_enclave_remove_pending_request` function, which is shown in Figure 10. The enclave stores the active tickets in a pending requests table. The ticket values are sent to the client application in encrypted form and they get decrypted before being used inside the enclave. The size of the table is computed as 2^{order} and the lowest $order$ many bits in the ticket value is used as an index to the pending requests table. This leads to a secret dependent address computation. ENCIDER reports that for different values of the ticket the entry may be placed in a different cache line when cache line size is 64 bytes.

We have found some data based cache side channels in `openSSL`, `s2n`, two of them, shown in Figure 7, and in `polarssl`, one of three as shown in Figure 12, and the other two due to the `DES_ROUND` macro that looks up from a table based on secret data in the `des3_crypt_ecb` function (lines 678 and 679). It is well-known that to protect against Lucky 13 attacks, secret dependent accesses like those found in `s2n` and `polarssl` must be avoided [60]. The cache side channel that ENCIDER detected in `openSSL` has been fixed in later versions [58].

TABLE 4
Comparison of time to detect side channel leakage without (wo) and with (w) API modeling. T, SB, CC, and DC denote timing side channel, secret dependent branch, code cache side channel, and data cache side channel.

Case Study	SC Type	Location	wo API mod. (secs)	w API mod. (secs)	Ratio wo/w
polarSSL	SB	ssl_tls.c:1393	112.59	7.83	14.95
	SB	ssl_tls.c:1406	117.09	7.94	14.75
	SB	ssl_tls.c:1408	127.26	8.27	15.39
	SB	ssl_tls.c:1425	122.57	8.09	13.92
Signal	CC	sgxsd-e.c:444	127.14	45.45	2.80
	DC	sgxsd-e.c:439	206.08	2.38	86.59
openSSL	T	s3_pkt.c:448	5247.41	1993.93	2.63
	T	s3_pkt.c:487	5249.38	2156.69	2.43

TABLE 5
Information on the API specifications for the evaluation benchmarks.

Case Study	API Input Specs		API Output Specs		Mix. Type	# API
	#fnc	#args	#fnc	#args		
SIGNAL Enclave	7	14	4	5	0	8
Ledger Enclave	6	7	4	4	1	6
TresorSGX	15	45	0	0	0	15
SGX IPP	10	12	0	0	0	10
SGX SSL	8	23	0	0	0	8
s2n_verify_cbc	3	3	0	0	2	0
ssl_read_record	1	1	0	0	0	1
_gcry_mpih_add_1	1	1	0	0	0	1
_gcry_mpih_sub_1	1	1	0	0	0	1
_gcry_mpih_cmp	1	2	0	0	0	1
_gcry_mpi_powm	1	1	0	0	1	1
BN_is_bit_set	1	2	10	0	0	1
BN_mod_exp.	1	1	0	0	1	1
BN_num_bits.	1	1	0	0	0	1
mbedtls_internal.	1	1	0	0	1	1
mbedtls_mpi_exp.	1	1	1	0	1	1
mpi_mul_hlp	1	2	0	0	0	1

7.2 Impact of API Modeling

One of the contributions of ENCIDER is to model API functions while incorporating the secret tainted parameters into information-flow tracking. Table 5 shows the details of API specifications for the evaluation benchmarks by reporting the number of API functions modeled for input arguments and output arguments and the number of mixed type specifications. To evaluate the impact of API modeling on the analysis performance, we have applied ENCIDER to the case studies which call some API function to perform decryption. Table 4 shows the timing information for without and with API modeling. We have not included mbedTLS-SGX and s2n in this table, as without API modeling the vulnerabilities could not be detected within 12 hours. The speedup for openSSL is modest compared to other case studies as some part of the cipher is implemented in assembly and is not handled by ENCIDER. We tried compiling openSSL with the `no-asm` option and in that case, similar to mbedTLS-SGX, without API modeling the vulnerability could not be detected within 12 hours. So, by modeling the decrypt APIs ENCIDER can detect the side channel vulnerabilities while achieving an order of magnitude speedup on average.

TABLE 6
Comparison of ENCIDER with ct-verif. **CT** and **TSC** denote constant-time and timing side channels.

	ct-verif			ENCIDER		
	CT?	**Time**	**Mem**	**TSC?**	**Time**	**Mem**
tea	yes	3.63	0.04	no	2.00	0.17
curve-donna	yes	891.33	7.51	no	119.60	3.89
mee-cbc-nacl	yes	191.68	4.67	no	165.89	1.24
ssl3_cbc_rem._padding	yes	3.41	0.41	no	1.01	0.02
tls1_cbc_rem._padding	yes	3.56	0.49	no	1.01	0.02
ssl3_cbc_copy_mac	yes	4.08	0.44	no	1.01	0.02
ssl3_cbc_dig._record	yes	27.19	0.79	no	245.24	0.23
rlwe_sample_ct	yes	4.8	0.04	no	1.00	0.02
salsa20	yes	6.04	0.08	no	1.00	0.02
chacha20	yes	11.82	0.1 0	no	8.04	0.03
sha256	yes	40.65	0.2 0	no	117.60	0.05
sha512	no	143.53	2.23	no	132.73	0.03
fix_pow	no	53.42	0.31	no	83.59	0.62
fix_cmp	yes	34.09	0.18	no	1.00	0.04
fix_convert	yes	51.36	0.17	no	1.00	0.04
fix_div	yes	38.75	0.17	no	1.00	0.04
fix_exp	yes	36.88	0.17	no	1.00	0.04
fix_sin	yes	42.08	0.17	no	1.00	0.04
fix_mul	yes	40.06	0.17	no	1.00	0.04
fix_sqrt	yes	50.7	0.17	no	1.00	0.04
fix_eq	yes	79.06	0.17	no	1.00	0.04
fix_ln	yes	47.23	0.21	no	5.02	0.05
Average		82.06	0.86		40.58	0.30

7.3 Comparing ENCIDER with ct-verif

We compare ENCIDER with ct-verif, which verifies constant-time implementations. Due to dependencies of ct-verif, we needed to install ct-verif and ENCIDER on an Ubuntu 14 Virtual Machine with an 8GB RAM and a 128GB disk. We used a timeout of 500 secs for ENCIDER. Table 6 shows the results of running both tools on the benchmarks from [29]. ENCIDER is faster than ct-verif on 18 out of 22 benchmarks, and achieves 50% improvement of run time and 65% improvement of memory usage on average. The two tools agree on the verification results except for `sha512` and `fix_pow` as ct-verif reports errors on the violation of some loop invariants related to constant-time behavior. According to [29], all testing cases are constant-time implementations. ENCIDER achieved zero false positive whereas ct-verif missed two cases in our evaluation. We contacted the ct-verif developers and confirmed that these two false positives were due to the missing assumptions that the instrumentation component of ct-verif failed to generate, and that they would require manual intervention to get fixed.

7.4 Comparing ENCIDER with CacheS

We have chosen CacheS among the related work due to being the most recent static analysis work that can detect both secret branches and cache side channels. We have used the leakages reported by CacheS to compare CacheS and ENCIDER in terms of precision and performance. We have excluded the leakages that require analysis of assembly or perl scripts. Results are shown in Table 7. Comparing to CacheS, ENCIDER achieves 20% run time improvement and 92% memory usage improvement in general. The only exception is the libgcrypt benchmarks where the detection of some of the leakages in the function `_gcry_mpi_powm` required exploration of more number of paths.

Among the 81 leakages, ENCIDER detected 65 of them. Four of the remaining 16 leakages were already reported as false positives of CacheS. Inspecting the remaining 12 leakages revealed them as being false positives. ENCIDER did not report any false positives for these benchmarks. It also detected 9 additional leakages that were not reported by CacheS within the analyzed functions. Our hypothesis on why CacheS missed those leakages is that we compiled the benchmarks for a 64-bit architecture and CacheS works for a 32-bit architecture. Some configuration option in openSSL applies to both architecture types and that's why ENCIDER got better coverage and detected those.

TABLE 7
Comparison of ENCIDER with CacheS. **SB**, **CSC**, **T**, and **F** denote secret branches, cache side channels, true positives and false positives, respectively. M, O, and L denote mbedtTLS 2.5.1, openSSL 1.0.2f, and libgcrypt 1.6.1.

	CacheS						ENCIDER					
	SB		CSC		**Time**	**Mem**	SB		CSC		**Time**	**Mem**
	T	**F**	**T**	**F**	**(secs)**	**(GB)**	**T**	**F**	**T**	**F**	**(secs)**	**(GB)**
M	8	0	19	3	808.70	9.65	8	0	19	0	323.16	0.54
O	12	0	5	0	205.30	6.11	16	0	9	0	194.65	0.58
L	15	6	6	7	228.80	7.75	16	0	6	0	457.05	0.76
Sum	35	6	30	10	1242.80	23.51	40	0	34	0	974.65	1.87

TABLE 8
Comparison of ENCIDER with DATA. **LKGSL**, **SB**, and **CSC** denote the number of source lines reported to have leakages, secret branches, and cache side channels, respectively.

File	**DATA**	**ENCIDER**		**COMMON**
	LKGSL	**SB**	**CSC**	
`bn_add.c`	8	6	19	4
`bn_div.c`	14	10	7	4
`bn_gcd.c`	11	5	0	0
`bn_lib.c`	13	18	11	6
`bn_mul.c`	1	5	5	0
`bn_rand.c`	1	1	0	0
`bn_shift.c`	8	4	4	3
`evp/encode.c`	4	9	1	2

7.5 Comparing ENCIDER with DATA

We compared ENCIDER with DATA [31], which uses differential dynamic analysis for detecting secret leakages. DATA uses random key generation to prepare secret inputs to the cryptographic code under analysis. Since DATA does not employ taint-tracking, it performs statistical tests to find code locations that reveal statistically significant differences among the generated dynamic traces. We were able to install DATA and ENCIDER on an Ubuntu 16.04 machine with an Intel Core i7@2.80GHz CPU and a 32 GB RAM. We applied DATA to the `rsa` benchmark of openSSL as mentioned in [31], and generated the source code locations that correspond to the leaky assembly instructions that pass the statistical tests. We applied ENCIDER on the functions that are reported to have leakages.

Table 8 presents various files from openSSL 3.0.0 and the number of leaky code locations reported by Data (**LKGSL**) and ENCIDER (**SB** and **CSC**). Column **COMMON** lists the number of code locations that are found to be leaky by both DATA and ENCIDER. A close inspection of the leakage locations reveals that DATA seems to report leakages deeper in the code while ENCIDER seems to cover more cases of leakages due to using symbolic inputs.

To detect the leakages reported in Table 8, DATA ran for a total of 83,315.14 secs and used a 4.90 GB of peak memory whereas ENCIDER ran for a total of 1,464.13 secs and used a 0.82 GB of peak memory.

7.6 False Positives

We have determined true positives using two methods: 1) Contacting the developers and getting confirmation for the new vulnerabilities, 2) Checking whether the vulnerabilities have been previously known either through published CVEs or being reported as true positives in previous studies. False positives were determined by studying the code and manually checking whether the leakage could also be detected via public return values or whether the data at the leakage location is actually tainted with secret data. The false positives reported by ENCIDER include three timing and one data cache side channel. So, overall ENCIDER reported four false positives out of 106 side channel reports, achieving a precision of 96%. ENCIDER reported a false timing side channel in the `sgxsd_enclave_server_call` function of the ContactDiscovery Enclave of the Signal App due to an unconstrained public return value of a modeled API function in one of the paths. It turns out that the two paths return two different public outputs and, hence, the leak could be determined by the public outputs. This type of false positives can be eliminated by providing a more detailed model of the API function, e.g., by constraining the return value. Another source of false positives is about security labeling of APIs that get called inside a loop. If the same memory cell is used for such a parameter in all iterations, branching conditions on that memory cell before the API function gets called are falsely detected as secret dependent branches. ENCIDER reported three false positive side channels of this type in openSSL 1.0.1c; two timing side channels in the `ssl3_get_record` function and one data cache side channel in the `ssl3_enc` function. This type of false positives can be eliminated by providing an intrinsic function for ENCIDER that can declassify the labels of such memory locations at the appropriate locations, e.g., at the beginning of the loop body.

7.7 Exploitability of Discovered Vulnerabilities

The feasibility of the local timing side channel in mbedTLS-SGX that was detected by ENCIDER follows from the feasibility of the code cache side channel in an earlier version of the code [47], which shows that the callsite for the `mbedtls_md_process` function can be exploited as a timing observation point. In fact, mbedTLS developers agreed with ENCIDER and fixed the vulnerabilities we reported in a recent version. We think that the other timing observation points found by ENCIDER are also possibly exploitable. However, openSSL developers have previously fixed the Lucky13 related vulnerabilities using a constant-time approach. s2n's developer do not consider local attackers as a serious threat due to the configuration of their servers. The timing side channel in the `secp256k1_ec_seckey_verify` function can be exploitable when it is called from the `moxie_bls_bip32_derive_secp256k1_private` function in the Ledger enclave as the failure case skips the subsequent computations. The secret dependent branch in the `BN_lebin2bn` function that gets called by several SGX SSL API, including `sgx_ecc256_compute_shared_dhkey`, and the timing side channel with multiple observation points inside the `sgx_rsa3072_sign` function can be exploited by a local attacker to facilitate a RACOON attack [64]. In fact, Intel confirmed both side channels and is planning issuing of a CVE. Finally, the data and code cache side channels in Signal's ContactDiscovery Service can be exploited together to recover part of the secret ticket value, which is employed as a "defense in depth" as indicated by the developers. Like any side channel analysis tool, ENCIDER can be utilized both for improving security of software as well as attacking software. We hope that developers will incorporate ENCIDER to the development process to detect and remove all types of leakages before deployment.

7.8 Limitations

Although ENCIDER can detect software side channels by working at the IR level, the differences between the IR and the binary executable and the details of the cache placement may lead to false positives as well as false negatives. Another source of false negatives may be due to the path explosion problem, especially when few opportunities exist for API modeling. Users can leverage the detailed coverage information provided by KLEE to decide if increasing the timeout may help with the side channel analysis.

8 Related Work

SGX Side Channel Analysis: Table 9 provides a comparison of ENCIDER with the related work on SGX side channel analysis. Stacco [17] detects secret dependent control-flows in SSL/TSL implementations running inside an SGX enclave by comparing dynamically collected traces. DATA [31] uses differential dynamic analysis to identify differences on execution traces on a byte-address granularity and reports those addresses that yield statistically significant differences. MicroWalk [21] uses mutual information analysis over a set of traces extracted during dynamic analysis to detect secret dependent branches and memory accesses in binaries including the SGX IPP library. ANABLEPS [22]

TABLE 9
Comparison of ENCIDER with other works that detect side channels in SGX enclaves.

Approach	Analysis Type	Side Channel Attack Type					Input Generation	Memory Address	Taint Tracking
		Cache-level		Page	Branch	Timing			
		ICache	DCache	Level	Level				
Stacco [17]	Dynamic	✓	✗	✓	✓	✗	Prot. Know.	Virtual	✗
DATA [31]	Dynamic	✓	✓	✓	✓	✗	Random	Virtual	✗
MicroWalk [21]	Dynamic	✓	✓	✓	✓	✗	Random	Virtual	✗
ANABLEPS [22]	Dynamic	✓	✗	✓	✓	✓	Fuzzing	Virtual	✗
ENCIDER (this work)	Static	✓	✓	✓	✓	✓	Not needed	Virtual	✓

TABLE 10
Comparison of ENCIDER with other state-of-the-art side channel analysis tools. ST: secret dependent time difference detection, PO: public output, IR: IR-based resource usage, TO: infers timing observation points (detects local timing side channels), SBR: reports secret dependent branches, CCL: code cache lines, DCL: data cache lines, DP: decision procedure, $BSTR$: byte-level specification and tracking of sensitive data, API: incorporating sensitivity of the API parameters, BF: bug finding, VR: verification, LQ: leakage quantification.

Approach	Timing SCA			IR	SBR	Cache SCA		DP	Info. Flow		St. Leak	Type
	ST	PO	TO	IR		CCL	DCL		BSTR	API		(BF,VR,LQ)
Almeida et al. [29] (ct-verif)	✓	✓	✗	LLVM	✗	✗	✗	SMT	✗	✗	✗	VR
Pasareanu et al. [48]	✓	✗	✗	Bytecode	✓	✗	✗	MaxSMT	✓	✗	✗	BF, LQ
Antonopoulos et al. [49] (Blazer)	✓	✗	✗	Bytecode	✓	✗	✗	SMT	✓	✗	✗	VR
Chen et al. [50] (Themis)	✓	✗	✗	Bytecode	✓	✗	✗	SMT	✓	✗	✗	VR
CoCo-CHANNEL [61]	✓	✗	✗	Bytecode	✓	✗	✗	SMT	✓	✗	✗	VR
Wang et al. [62] (CacheD)	✗	✗	✗	x86	✗	✗	✓	SMT	✓	✗	✗	BF
Brotzman et al. [63] (CaSym)	✗	✗	✗	x86	✓	✗	✓	SMT	✓	✗	✗	BF
Wang et al. [30] (CacheS)	✗	✗	✗	REIL	✓	✗	✓	SMT	✓	✗	✗	BF
ENCIDER (this work)	✓	✓	✓	LLVM	✓	✓	✓	SMT	✓	✓	✓	BF

detects secret dependent control-flows in enclave binaries using concolic execution and fuzzing and has been applied to legacy applications running on a library OS. To our knowledge, ENCIDER is the first IR-level side channel analysis tool that incorporates the strong attacker and programming model of SGX to detect side channels that can be exploited by all of the five types of attacks that are mentioned in Table 9. Unlike these approaches, ENCIDER reports the leakage sites as it employs taint tracking whereas the approaches in [17], [21], [22], [31] may require additional manual processing of the traces with differences in resource usage to identify the leaky branch instructions. ENCIDER complements the approach of Moat [65], which formally verifies SGX enclaves for explicit leaks.

Side-channel Analysis: Table 10 provides a detailed comparison of ENCIDER with state-of-the-art side channel analysis approaches. Previous works on timing side channel analysis [29], [48], [49], [50], [61] do not consider local attackers and can only detect timing differences observed at predefined code locations, e.g., termination points. ENCIDER, on the other hand, can automatically infer timing observation points that can be exploited by local attackers (see Section 4 for a motivating example). Also, the approaches in [29], [48], [49], [50], [61] focus on resource types for which the amount of resource usage can be computed independent of the interfering paths, e.g., execution time, whereas ENCIDER can also handle resource types that require the details of the resource usage in the interfering paths, e.g., instruction cache lines. Another aspect in which our work differs from [48], which also uses symbolic execution, is that we decompose the secret dependent path constraints and minimize the number of SMT queries instead of offloading all such constraints to a MaxSMT solver, as our goal is to detect side channels rather than quantify the leakage. Also, MaxSMT solving is known not to scale to large input sizes [48]. Similar to Blazer [49], ENCIDER follows a decomposition approach to detecting timing side channels. However, ENCIDER differs from Blazer in two aspects: 1) ENCIDER's partitions are performed with respect to secret dependent branches and the partitions are organized into a hierarchy to optimize constraint solving; 2) ENCIDER can also generate attack inputs.

Our work differs from other work on cache side channel analysis [30], [62], [63] as ENCIDER detects code cache side channels by leveraging the metadata extracted from the binaries. While the approaches in [30], [63] also report secret dependent branches like ENCIDER, they can lead to false positives. The risk of a code cache side channel depends on the memory locations of the basic blocks on the paths of the secret dependent branch.

Our work differs from the work in [29], [30], [48], [49], [50], [62], [63] by incorporating precise information flow modeling of APIs. Finally, to our knowledge, our work is the first to unify timing and cache side channel analyses within the same non-interference analysis and ENCIDER is the first tool to provide multiple types of secret leakage detection: timing side channels (remote and local), and cache side channels (instruction and data).

9 Conclusion

To address side-channel threats against confidential computing, we have designed and implemented ENCIDER detecting both timing and cache side-channel attacks automatically. ENCIDER uses dynamic symbolic execution and supports SGX program modeling. We have applied

ENCIDER to 4 real-world SGX enclave implementations, 2 SGX SDK crypto libraries, 3 TLS implementations, and 3 common crypto libraries. In total, we have found 29 timing side channels and 73 cache side channels.

Acknowledgments

This work was partially funded by the National Science Foundation under awards CNS-1815883 and CNS-1942235, by the Semiconductor Research Corporation, and by an Intel gift. We would like to thank Thomas Shrimpton and the anonymous reviewers for their feedback.

References

[1] Intel Corporation, "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx, 2016.

[2] Linux Foundation Projects, "Confidential Computing Consortium," https://confidentialcomputing.io/, last accessed 15.10.2019.

[3] Signal, "Private Contact Discovery Service (Beta)," https://github.com/signalapp/ContactDiscoveryService, last accessed 05.01.2019.

[4] "Ledger enhances Blockchain applications security using Intel technology," https://www.ledger.com/ledger-enhances-blockchain-applications-security-using-intel-technology/, 2017, Last accessed 09.10.2019.

[5] "Intel® Software Guard Extensions SSL," https://github.com/intel/intel-sgx-ssl, last accessed on 07.07.2020.

[6] "mbedtls-SGX: a TLS stack in SGX," https://github.com/bl4ck5un/mbedtls-SGX, last accessed on 07.07.2020.

[7] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[8] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th USENIX Security Symposium*, 2003.

[9] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password interception in a SSL/TLS channel," in *Advances in Cryptology - CRYPTO 2003*, Berlin, Heidelberg, 2003, pp. 583–599.

[10] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013, pp. 526–540.

[11] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-vm attack on AES," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham: Springer International Publishing, 2014, pp. 299–319.

[12] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[14] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre: Exploiting Speculative Execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[15] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[16] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time RSA," *J. Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

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

[18] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Proceedings of LATINCRYPT*, 2012, pp. 159–176.

[19] Intel Corporation, "Intel Integrated Performance Primitives," https://software.intel.com/en-us/ipp, last accessed 15.10.2019.

[20] ——, "Intel Software Guard Extensions for Linux OS." https://01.org/intel-softwareguard-extensions, Jun. 2016.

[21] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 161–173.

[22] W. Wang, Y. Zhang, and Z. Lin, "Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019.*, 2019, pp. 443–457.

[23] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[24] "CVE-2021-0001," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0001, last accessed on 02/28/2022.

[25] "INTEL-SA-00477," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00477.html, last accessed on 02/28/2022.

[26] "Intel 2021 Product Security Report," https://www.intel.com/content/www/us/en/security/intel-2021-product-security-report.html, last accessed on 02/28/2022.

[27] "CVE-2020-16150," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-16150, last accessed on 02/28/2022.

[28] "Local side channel attack on classical CBC decryption in (D)TLS," https://tls.mbed.org/tech-updates/security-advisories/mbedtls-security-advisory-2020-09-1, last accessed on 02/28/2022.

[29] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[30] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 657–674.

[31] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA – differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 603–620.

[32] X. Ruan, *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[33] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® Software Guard Extensions: EPID Provisioning and Attestation Services," *White Paper*, Mar. 2016.

[34] Intel Corporation, "Intel® Software Guard Extensions SDK for Linux* OS Developer Reference ," https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.7_Open_Source.pdf, 2016.

[35] S. Johnson, https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, Feb. 2018.

[36] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[37] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[38] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[39] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *ACM CCS*, 2017.

[40] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Confer-*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3160346, IEEE Transactions on Dependable and Secure Computing

ence on Cryptographic Hardware and Embedded Systems. Springer, 2017, pp. 69–90.

[41] Intel Corporation, "Intel Analysis of Speculative Execution Side Channels," Jan. 2018, white Paper Revision 1.0.

[42] C. Percival, "Cache missing for fun and profit," in *In Proc. of BSDCan 2005*, 2005.

[43] ARM, "ARM TrustZone," https://developer.arm.com/ip-products/security-ip/trustzone, last accessed 15.10.2019.

[44] AMD, "AMD Secure Encrypted Virtualization," https://developer.amd.com/sev/, last accessed 15.10.2019.

[45] "mbedtls 2.6.0," https://tls.mbed.org/code/releases/mbedtls-2.6.0-gpl.tgz, last accessed on 07.07.2020.

[46] CVE-2013-0169, "Lucky thirteen - timing side channel during decryption," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0169, last accessed on 07.07.2020.

[47] CVE-2018-0498, "Plaintext recovery on use of CBC based ciphersuites through a cache based side-channel," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0498, last accessed on 07.07.2020.

[48] C. S. Pasareanu, Q. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, 2016, pp. 387–400.

[49] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, 2017.

[50] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 875–890.

[51] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, 2015, pp. 49–64.

[52] Intel, "Intrinsics guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, last accessed 11.13.19.

[53] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, 2004, pp. 75–88.

[54] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[55] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, 2005, pp. 352–367.

[56] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

[57] ct verif, "Verifying Constant-Time Implementations," https://github.com/imdea-software/verifying-constant-time/tree/master/examples, last accessed on 02.04.2020.

[58] B. Laurie, "Make CBC decoding constant time," https://git.openssl.org/?p=openssl.git;a=commit;h=2acc020b770920657a169bf6be4ff12b254255e6, last accessed on 07.07.2020.

[59] libsec256k1, "Optimized C library for ECDSA signatures and secret/public key operations on curve secp256k1," https://github.com/bitcoin-core/secp256k1, last accessed on 09.12.2019.

[60] A. Langley, "Lucky Thirteen attack on TLS CBC," https://www.imperialviolet.org/2013/02/04/luckythirteen.html, last accessed on 07.07.2020.

[61] T. Brennan, S. Saha, T. Bultan, and C. S. Pasareanu, "Symbolic path cost analysis for side-channel detection," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 27–37.

[62] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 235–252.

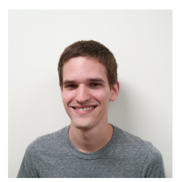
[63] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019, pp. 505–521.

[64] R. Merget, M. Brinkmann, N. Aviram, J. Somorovsky, J. Mittmann, and J. Schwenk, "Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E)," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1151, 2020.

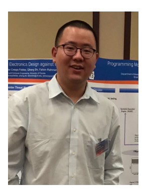
[65] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

Tuba Yavuz Tuba Yavuz is an Assistant Professor at the Electrical and Computer Engineering Department of University of Florida. Her research interests include formal methods, program analysis, and system security. Yavuz received a Ph.D. in computer science from the University of California of Santa Barbara. Contact her at tuba@ece.ufl.edu address.

Farhaan Fowze received his B.Sc. from Bangladesh University of Engineering and Technology(BUET) in 2012. He is currently a Postdoc at the Florida Institute of Cybersecurity Research (FICS). He received his Ph.D. from the Electrical and Computer Engineering Department at the University of Florida. His research interests include model extraction, binary analysis, and program analysis. Contact at farhaan104@ufl.edu .

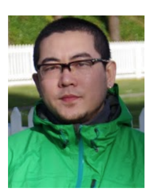


Grant Hernandez is a security engineer at Qualcomm. He receieved his Ph.D. from the Computer and Information Science and Engineering department at the University of Florida. His research focuses on automated embedded binary firmware analysis to discover vulnerabilities at scale. Contact him via email grant.hernandez@ufl.edu.



Ken (Yihang) Bai is a Ph.D. student at the Electrical and Computer Engineering Department at the University of Florida His research focuses on model guided binary analysis. Contact him via email baiyihang@ufl.edu.

Kevin Butler is an Associate Professor of Computer and Information Science and Engineering at the University of Florida, and Associate Director of the Florida Institute for Cybersecurity Research. His research focuses on establishing the trustworthiness of computer systems and embedded devices. Butler received at Ph.D. in computer science and engineering from the Pennsylvania State University in 2010. He is a Senior Member of IEEE and ACM.



Dave (Jing) Tian is an Assistant Professor in the Department of Computer Science at Purdue University. His research involves operating system security, embedded system security, and trusted computing. He received his Ph.D. in computer science from the University of Florida in 2019. For more information, please access https://davejingtian.org.