SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2019; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

An Exploration of Effective Fuzzing for Side-channel Cache Leakage

Tiyash Basu[†], Kartik Aggarwal[‡], Chundong Wang[*§], and Sudipta Chattopadhyay[§]

[†] *Saarland University, Saarland, Germany*
[‡] *Birla Institute of Technology and Science, Pilani, Rajasthan, India*
[§] *Singapore University of Technology and Design, Singapore*

SUMMARY

Adversaries can compute the secret information of a program, such as the key for encryption routines, from *side channels* in the light of timing- and access-based CPU cache behaviours. As a result, it is crucial to understand whether a program is vulnerable to side-channel cache leakage or not. Yet how we can find out such a vulnerability in a program remains a problem. In this paper, we revisit this problem and contemplate a test generation methodology, which, in both timing- and access-based dimensions, systematically discovers the cache side-channel leakage of an arbitrary software program. At the core of our test generation framework is an algorithm that explores the program's input space and adapts at runtime according to observed cache performance in the executed tests. We have implemented our test generator for timing- and access-based attack tests and evaluated it with open-source subject programs, including ones from OpenSSL and Linux GDK libraries. Our extensive evaluation effectively discloses the vulnerabilities of these real-world software to both timing- and access-based cache attacks. We also empirically show that our test generator achieves higher and comparable effectiveness, respectively, in simulations and real hardware platforms with regard to revealing cache side-channel leakage compared to state-of-the-art fuzz testing tools.

Received . . .

KEY WORDS: Side-channel Attacks, Cache Timing Attacks, Cache Access Attacks, Fuzzing, Testing

1. INTRODUCTION

Side-channel attacks based on cache timing and access [6, 41, 26] have emerged to be a serious security breach in real-world software systems. Leveraging side-channel attacks, adversaries manage to figure out sensitive information of a program execution, e.g., a secret key, without any knowledge of the functional input or output of the program. The key intuition behind a cache timing attack is to observe the timing of cache hits and misses in executing a program, and subsequently use this timing to determine the secret features of the respective program. Similarly, cache lines accessed due to different input values in running the program also yield meaningful observations that can be interpreted [30, 26]. The disclosure of such timing- and access-based observations to an untrusted party may have disastrous consequences, resulting in a possible leakage of classified data and in turn a breakdown of the overall system. Therefore, it is crucial to examine software systems against potential timing- and access-based attacks with CPU cache.

[*]Correspondence to: C. Wang (cd_wang@outlook.com).
[‡]This work was done when Kartik Aggarwal was an intern in Singapore University of Technology and Design.

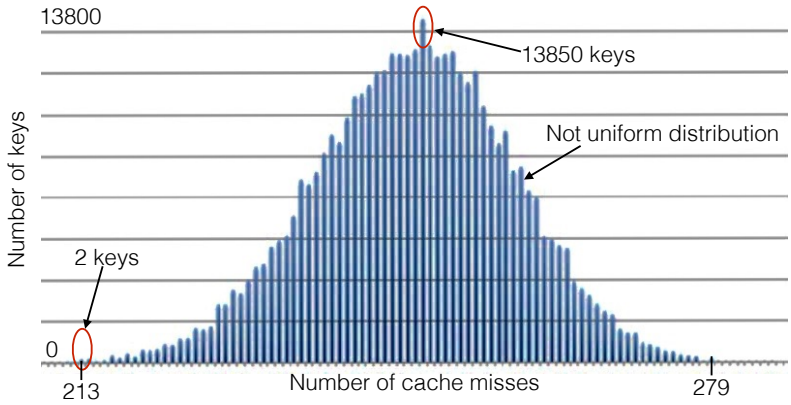

Figure 1. For a fixed input message, the plot shows the distribution of the number of keys with respect to a given number of cache misses. The experiment was performed for an implementation of AES-128 [14] for 256,000 different keys (picture taken from [11])

Table I. Cache Sets Accessed with Different Keys

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}	Cache Line Access Mark (Hexadecimal)
63	189	126	2	133	79	66	188	93	198	189	28	184	92	115	11	0200 FF00 0200 0040
49	26	222	36	133	117	181	195	29	18	167	177	81	58	30	11	0000 FF00 0000 0000
57	0	126	65	133	79	44	19	160	17	71	137	45	248	50	85	0000 FF00 0800 0010
49	133	0	36	158	142	44	188	93	249	249	137	45	224	30	11	0800 FF00 0000 0002
203	0	222	2	158	79	44	99	187	249	84	94	153	248	2	43	4001 FF00 1000 0000
203	133	0	2	116	142	113	195	187	17	84	137	184	224	115	43	4000 FF00 0008 0000

CPU cache is one of the prime components of a computing system for program execution. Given a program, its vulnerability to cache timing- and access-based attacks is entailed by the amount of information that can leak through the program's cache performance and behaviour. The cache performance and behaviour of a program, in fact, are significantly influenced by the underlying execution platform. Unfortunately, the state-of-the-art in software testing to validate platform dependent properties (e.g., performance) is far from being matured. Cache side-channel leakage, being dependent on the cache and thus, the underlying execution platform, is therefore an area that deserves attention.

In this paper, we take a step forward to test side-channel leakage with a focus on the cache. Specifically, given a program and a cache configuration, we formulate the test generation problem to validate software systems against both cache timing- and access-based attacks. In the light of this formulation, we show an appropriate coverage metric for the test generation problem and design a directed testing strategy to expose the cache side-channel leakage of an arbitrary program.

In order to understand the challenges involved in testing cache side-channel leakage via timing-based attack, let us consider the illustration in Figure 1. Figure 1 demonstrates the execution of an implementation of Advanced Encryption Standard (AES) [14] for a fixed plaintext message and 256,000 different keys. AES is an encryption algorithm that uses one single private key (i.e., being *symmetric* in contrast to asymmetric encryption that uses a public key and a private key) with a length of 128, 192, or 256 bits, to encrypt and decrypt data. Being fast and efficient, AES encryption has been widely used in many applications [17], but most software implementations of it are prone to side-channel leakage regarding cache timing and access records [52, 26, 29]. In Figure 1, the horizontal axis shows the number of cache misses exhibited (i.e., in a range between 213 and 279) and the vertical axis captures the number of 128-bit keys that induce them. Figure 1 clearly shows that the distribution of cache misses is essentially a *Gaussian* distribution. There exists only two keys which induce extreme cache performance (i.e., maximum or minimum), whereas there exists 13,850 keys which induce the modal cache performance. From the perspective of software testing, Figure 1 reveals the following challenges. Firstly, all 256,000 executions in Figure 1 exercise the *same program path*; therefore, merely exploring program paths is not sufficient to explore different cache behaviours of the respective program. This is because different executions of the

DOI: 10.1002/stvr

same path can yield different cache behaviours, as these executions may access different data even when executing the same path (e.g., for pointer-based accesses). Since it is critical to explore the different cache behaviours to expose cache side-channel leakage, optimizing a test generation towards path coverage may not reveal cache side-channel leakage. Secondly, only a few keys may exhibit certain cache behaviour, such as the leftmost and rightmost cache behaviours shown by two ends of Figure 1. As a result, we need directed test generation strategies that can maximize the width of the Gaussian curve as illustrated in Figure 1.

To investigate the influence of secret inputs on cache line accesses, we have monitored cache lines accessed at runtime with the AES program. We picked six different keys which yet share values in between. In Table I, b_i ($i \in [1, 16]$) captures the i-th byte of the AES secret key. Without loss of generality, we have simulated a 2KB fully associative L1 data cache with 64 cache lines (32B per cache line). We employed a Cache Line Access Mark, which is a bitmap for all 64 cache lines, to present whether a cache line has been accessed ('1') or not ('0') at runtime when the respective keys were processed by the AES program. As shown by the rightmost column of Table I, we present the Cache Line Access Mark in hexadecimal (one hexadecimal digit for four cache lines) for a concise representation. For all the experiments, we used the same plaintext message. As evidently demonstrated by Table I, different keys, in spite of sharing values in between and being processed through the same execution path, cause different cache lines to be accessed while some of the cache lines are accessed more than others. For example, there are eight cache lines (cf. 'FF' in Table I) that have been accessed for every one of the six keys while there are other 24 cache lines that have not been touched at all.

To summarize, the observations at the CPU performance and behaviour for a program, such as the aforementioned cache misses/hits and accessed cache lines, may vary significantly for different inputs, with some observations being more *frequent* than others. This requires systematically searching the input space of the program. Such a search process should ensure that the testing process not only exhibits the frequent observations (hence, common cache line hits and misses as well as frequently accessed cache lines), but also the infrequent cache behaviour (hence, exceptional behaviours).

In this paper, we design and evaluate a test generation framework, based on simulated annealing, to address the challenges mentioned in the preceding paragraphs. The output of our framework is a test suite, where each test in it witnesses a unique observation of cache timing and access for the program. In other words, each of the tests generated by our framework for an arbitrary program exhibits a distinctively specific cache timing or access behaviour. We show that the number of tests in our test suite is directly correlated with the amount of information that may leak through cache timing- and access-based attacks, respectively. Our work significantly differs from the work in static analysis of cache side channels [16, 33]. In particular, our test generation process does not exhibit any false positive, meaning that each test case in the test suite serves as a witness of a cache behaviour in real executions. Such witnesses with unique observations of cache timing and access, if thoroughly analyzed, are sufficiently usable to leak the secret information (e.g., the keys for AES encryption) for a software program [52, 26, 38]. Besides, our work has a significant flavour of testing and debugging. This means the tests generated by our framework can further be used to investigate a program and potentially reduce its cache side-channel leakage.

Not only searching input space is non-trivial for testing cache side-channel leakage, but the test execution itself also makes the testing problem challenging. For each generated test, we need to measure the cache performance. Unfortunately, such a measurement is extremely noisy in real hardware due to the presence of multiprocessing hardware and supervisory software (e.g., operating systems). For an effective test generation, it is crucial to minimize such noise, as potential attackers may employ several noise reduction techniques by themselves to mount a successful attack. In order to reduce the noise in test execution, we firstly use a simulator as a controlled environment where the execution statistics (e.g., the number of cache misses) is deterministic. By doing so, the aforementioned disturbing noise is ruled out. Secondly, in real hardware, we leverage performance counters and statistical methods, as well as explicitly introduce instructions to isolate execution in a single core. These methods, in turn, reduce the noise when measurements are taken from

DOI: 10.1002/stvr

real hardware. Finally, we employ state-of-the-art side-channel attacks (e.g., Prime+Probe [38]) to validate the effectiveness of our test generation strategy.

The remainder of the paper is organized as follows. We first provide a primer on CPU caches as well as cache-related side-channel information leakages in Section 2. We give an overview of the targeted problem in Section 4 and make the following contributions in this paper:

1. We formulated the test generation problem and an appropriate test coverage criterion for validating cache side-channel leakage of an arbitrary program with regard to timing- and access-based attacks (Section 3).
2. We designed a test generation algorithm that aims to search the program input space to explore different cache behaviours and subsequently, reveal the cache side-channel leakage of a program (Section 5).
3. We implemented our test generation algorithm in an open platform. Our implementation and all the experimental data is publicly available[†] to facilitate research.
4. We evaluated our test generator with real-world programs from OpenSSL library [40] and Linux GDK library [43] in a controlled environment (using Simplescalar [3] simulator) as well as in real hardware. We also compared our test generator with state-of-the-art fuzz testing tools Radamsa [28] and AFL [54]. Our evaluation effectively reveals cache side-channel leaks in all the chosen subject programs. Our directed approach in test generation outperforms (in terms of revealing cache side-channel leakage) both Radamsa and AFL (Section 6) for timing-based attacks. As to access-based attacks, our test generator achieves much better and comparable effectiveness compared to Radamsa and AFL in simulation and real hardware, respectively.

We conclude this paper with threats to validity (Section 8) and consequences (Section 9).

2. PRELIMINARIES

CPU cache is a critical component of memory hierarchy in computer systems. In general, the memory hierarchy is composed of disk, main memory, CPU cache(s), and CPU registers. Data is persistently stored in a disk. When a software program executes, the required data is loaded from the disk to main memory. This loaded data is then processed by the CPU for running the program. However, due to the limited number of CPU registers, they can only hold a limited number of variables in the program. As a result, CPU caches are employed to temporarily store and retrieve data. Modern CPUs typically embrace multiple levels of CPU caches, e.g., L1, L2, and L3 caches. L1 cache is the closest to the CPU. It is smaller in size as compared to L2 and L3 caches, however, it has very short access latency. For example, our evaluation uses an Intel® Core™ i5-3337U CPU. This has 64KB, 1MB, and 4MB, respectively, for its L1, L2, and L3 caches.

CPU caches have much smaller capacity than main memory. A CPU cache is uniformly partitioned into cache lines. Due to the smaller size of the CPU cache as compared to the main memory, each cache line might be shared by several memory locations. If the data needed by a software program is always in the CPU cache (i.e. a cache hit), then the performance of the program is superior. When the required data is not found in the cache, cache misses occur. Cache misses can occur for multiple reasons. Firstly, when a memory location is accessed for the first time, it needs to be brought into the cache, resulting in a cache miss. Such cache misses are often referred to as cold cache misses. Secondly, a memory block might be replaced from the cache by other memory blocks. The exact nature of this replacement is hardware controlled and depends on the associativity of the cache. For example, with a direct-mapped cache, each memory location can be mapped to only one cache line. With a fully-associative cache, each memory location can freely be mapped to any cache line. Thus, for fully-associative caches, cache misses occur when a memory location is accessed for the first time or when the cache is full. With an N-way set-associative cache (N is an integer, e.g., $N = 2$) that partitions the cache into N sets with m cache lines in each set, a memory

[†] https://github.com/tiyashbasu/Cache_Side_Channel_Tester

DOI: 10.1002/stvr

location can be mapped to any one of m cache lines in one set. To formalize the cache architecture for software testing, a model of CPU cache will be presented in Section 3.1.

CPU cache employs one of the most complex mechanisms in computer systems. All software programs rely on CPU caches for execution. Therefore, running a software program must incur interactions with the CPU cache. However, CPU caches are hardware controlled and their management is non-transparent to software developers. Moreover, state-of-the-art CPU management techniques are undisclosed by the manufacturers. Despite such non-transparency, attackers have managed to find *side channels* to uncover secret information of programs regardless of Intel or ARM processors [31, 24, 36]. The performance gap between a CPU cache and main memory, with different inputs fed during the execution of a program, entails different observations, such as the numbers of cache misses/hits and cache lines accessed. This, in turn makes cache side-channel attacks practical to discover sensitive information from a program. Such sensitive information includes secret keys of encryption routines, keystrokes in password checkers or any other private information such as the contact list of users. For example, by analyzing the timing differences of cache misses and hits in running an implementation of AES encryption, the private key could be figured out [41, 52]. This is one of the typical timing-based attacks targeting the CPU cache. Furthermore, by learning about the access records to CPU cache lines, attackers are also able to retrieve the keys used for AES encryption [26, 9]. Such an attack is categorized as access-based side-channel leakage.

Cache side-channel attacks are often non-invasive and they can even be easily mounted over the network [6]. We note that, side channels do not only exist for the CPU cache, and there are also other side channels besides cache timing and access, such as measuring the power consumption or even the vibration sound of electronic devices during the execution of a program [34, 30, 46, 23, 29, 47]. These side channel attacks are beyond the scope of this paper. Readers may refer to other literature for details.

3. PROBLEM FORMULATION OF TEST GENERATION

In this section, we we first provide brief models on caches and cache side channels before formulating the test generation problem to quantify the cache side-channel leakage.

3.1. A Model of CPU Cache

A typical cache architecture can be defined via four parameters – number of cache sets, cache line size, associativity and replacement policy. For example, consider an M-bit byte-addressable memory address space. Data from the main memory (DRAM) is fetched at the granularity of cache line size. Let us assume the cache line size is $2^{\mathcal{B}}$ bytes and the number of cache sets is $2^{\mathcal{S}}$. An arbitrary memory address X is mapped to the cache set $\lfloor \frac{X}{2^{\mathcal{B}}} \rfloor \ mod\ 2^{\mathcal{S}}$. Thus, within M-bit memory address, $\mathcal{S}$ bits can uniquely identify the cache set where an address is mapped to and $\mathcal{B}$ bits can be used to identify the individual bytes within a cache line. If the cache associativity is $\mathcal{A}$, then the cache can hold a total of $\left(2^{\mathcal{S}} \cdot \mathcal{B} \cdot \mathcal{A}\right)$ bytes of memory. When a cache set holds more than $\mathcal{A}$ cache lines and a new memory address is mapped to the same cache set, then a replacement policy is applied to evict a cache line from the set. The replacement policy is often a proprietary information and is unknown to the users of the system. Since our test generation methodology is oblivious to the replacement policy, for the rest of the paper we will simply capture the cache architecture via a triplet $\langle 2^{\mathcal{S}}, 2^{\mathcal{B}}, \mathcal{A} \rangle$. When the cache associativity $\mathcal{A} = 1$, the cache holds exactly one cache line for each cache set and it is a direct-mapped cache.

3.2. Cache side channel leakage

In this paper, we consider two common types of side-channel attacks, i.e., timing- and access-based attacks. In such side-channel attacks, the attackers monitor the number of cache misses and the trace

DOI: 10.1002/stvr

of cache lines accessed, respectively, when running the victim program. Subsequently, they employ statistical techniques to unveil the secret information [6] from the observed traces.

We assume the cache side channel to be a function, i.e.,

$$C : \mathbb{I} \rightarrow \mathbb{O}. \tag{1}$$

The function C maps a finite set of sensitive inputs to a finite set of observations. Given that the attackers monitor the number of cache misses for timing-based attacks, an observation $o \in \mathbb{O}$ captures the number of cache misses in an execution. As to access-based attacks, an observation o means a vector of cache lines accessed, as shown in the rightmost columns of Table I. If we model the choice of a secret input via a random variable X and the respective observation by a random variable Y, the leakage through the channel C is the reduction in uncertainty about X when Y is observed. In particular, the following result holds for quantifying the cache side-channel leakage [33]:

$$ML(C) \leq \log_2 |C(\mathbb{I})| \tag{2}$$

where $ML(C)$ captures the maximal leakage through the channel C. Here the cardinality of set $C(\mathbb{I})$ captures the number of output observations $\mathbb{O}$, where $C : \mathbb{I} \rightarrow \mathbb{O}$. In other words, $|C(\mathbb{I})|$ is the cardinality of the co-domain of function C. In Formula 2, equality holds when X is uniformly distributed.

3.3. Implication to test generation

Since we aim for a software validation framework, we assume the presence of a strong attacker whose choice of secret input is uniformly distributed. Therefore, $ML(C)$ is maximized and $ML(C) = \log_2 |C(\mathbb{I})|$ holds (*cf.* Formula 2). As a result, the number of unique observations by the attacker (i.e., $|C(\mathbb{I})|$) resembles the side-channel leakage of the respective program.

The quantification $|C(\mathbb{I})|$ provides preliminary insights on testing an arbitrary software. On the one hand, $|C(\mathbb{I})|$ provides an appropriate coverage metric for a test-generation scheme targeted to discover side-channel leakage. On the other hand, $|C(\mathbb{I})|$ can be used to compute the number of bits leaked through side channels (*cf.* Formula 2).

Motivated by $|C(\mathbb{I})|$, we develop a test generation algorithm that aims to maximize the value of $|C(\mathbb{I})|$. This means we generate test inputs in order to explore as many unique observations as an attacker can make. For each unique observation explored by our framework, a witness is provided. These witnesses can further be investigated to discover the information leak for respective executions. Besides, the number of unique observations explored by our test generation is directly correlated with the side-channel leakage quantified in Formula 2. As we look into both timing- and access-based attacks, Formula 1 is rewritten as

$$\begin{cases} C_t : \mathbb{I} \rightarrow \mathbb{O}_t, \\ C_a : \mathbb{I} \rightarrow \mathbb{O}_a, \end{cases} \tag{3}$$

where C_t and $\mathbb{O}_t$ are the channel and set of observations for timing-based attacks, respectively, while C_a and $\mathbb{O}_a$ are for access-based attacks.

3.4. Practical consideration

For timing-based attacks (i.e. for channel C_t), it is evident that we need to record the number of cache misses induced for each test execution. This can be accomplished via accessing the hardware performance counters, e.g., L1 cache miss counter. While testing for access-based cache attacks (i.e. for channel C_a), we need to check the cache lines being accessed for each test execution (cf. Table I). Although it is possible to identify the exact set of cache lines accessed in a simulated and controlled environment, such an information may not be feasible to obtain in real-world systems. This is because identifying the exact set of cache lines often requires a detailed knowledge of the underlying cache architecture, e.g, the cache replacement policy. Such knowledge is proprietary for most real-world systems. Moreover, checking the access statistics of each cache line for a test execution may entail heavy overhead to the test generator.

DOI: 10.1002/stvr

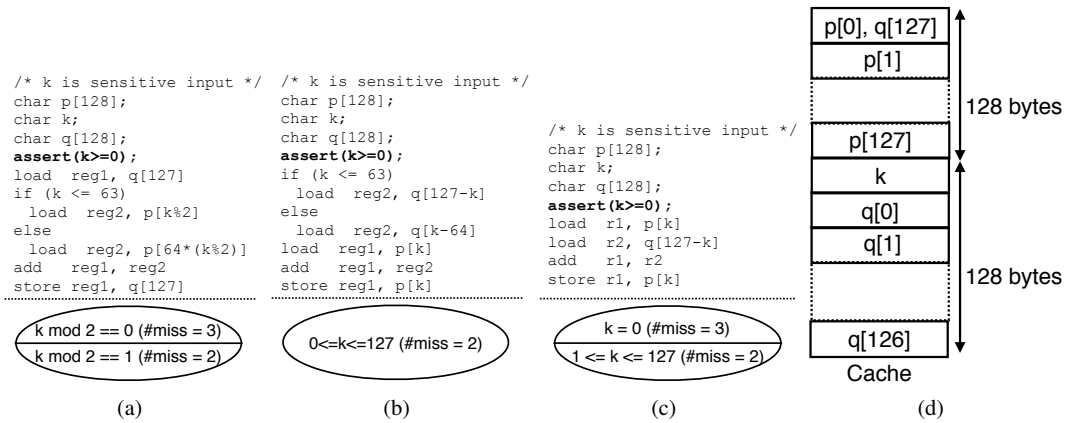

Figure 2. k is a sensitive input taking only positive values. (a)–(c) three code fragments and respective partitions of the input space with respect to the number of cache misses ($reg1$, $reg2$ represent registers), (d) mapping of program variables into a direct-mapped cache with 256B ($q[127]$ and $p[0]$ conflict in the cache)

To alleviate the challenges mentioned in the preceding paragraph, we approximate the maximal leakage computation via an efficient and scalable check for each test execution. Instead of checking whether a *cache line* is accessed during a test execution, we check whether a *cache set* is accessed. Recall that a set-associative cache, as captured by the triplet $\langle 2^{\mathcal{S}}, 2^{\mathcal{B}}, \mathcal{A} \rangle$, groups $\mathcal{A}$ different cache lines in a single cache set. Let us assume in test execution e, $\hat{O}_a^e$ and O_a^e capture the set of cache sets and the set of cache lines accessed, respectively. Over a set of arbitrary test executions E, the following relationships hold:

$$\left| \bigcup_{e \in E} \hat{O}_a^e \right| \leq \left| \bigcup_{e \in E} O_a^e \right| \leq |C_a(\mathbb{I})| \tag{4}$$

For access-based cache attacks, our test generation methodology computes $\left| \bigcup_{e \in E} \hat{O}_a^e \right|$ over a set of systematically generated test executions E. From Formulas 2–4, we have the following relationship when the choice of secret input is uniformly distributed:

$$\left| \bigcup_{e \in E} \hat{O}_a^e \right| = \overline{ML}(C_a) \leq ML(C_a) \tag{5}$$

where $\overline{ML}(C_a)$ captures the approximate leakage computed by our test generator for access-based channel C_a. Therefore, besides directing the test generator towards maximizing $\left| \bigcup_{e \in E} \hat{O}_a^e \right|$, the test generator also provides a lower bound on the amount of information leaked via access-based channel C_a.

It is worthwhile to note that in direct-mapped caches, associativity $\mathcal{A} = 1$. As a result, each cache set holds exactly one cache line. Thus, for direct-mapped caches, equality holds in Formulas 4–5.

4. MOTIVATIONAL STUDY

In this section, we motivate the challenges in test generation through examples. Since memory performance is accurately captured in the binary code, we directly test the binary code. However, for the sake of illustration, we use both assembly-level and source-level syntaxes in Figure 2. Figures 2(a)–(c) show three different code fragments. These code fragments execute on a platform employing a direct-mapped, 256 bytes cache. The mapping of variables $p[0 \ldots 127]$, $q[0 \ldots 127]$ and k into the cache is shown in Figure 2(d).

DOI: 10.1002/stvr

Consider the execution of the code in Figure 2(a), starting with an empty cache. We assume that k has been assigned to a register and for the sake of simplicity in the example, we ignore the cache performance of "assert" function call. Since k is assigned to a register, accessing k does not involve accessing the cache. When k is even, we get the following sequence of memory accesses: $q[127] \rightarrow p[0] \rightarrow q[127]$. The first two accesses to $q[127]$ and $p[0]$ would incur cache misses due to the initial empty state of the cache. Moreover, since $p[0]$ and $q[127]$ are mapped to the same location in the cache, the access to $p[0]$ will replace $q[127]$ from the cache, resulting in the second access of $q[127]$ to be a *cache miss*. A similar exercise would reveal that for odd values of k, the code in Figure 2(a) suffers two cache misses. To summarize, the code in Figure 2(a) exhibits two different cache behaviours, but these behaviours are not directly correlated with the program path. More specifically, each program path in Figure 2(a) exhibits all possible cache behaviour of the overall program.

The example in Figure 2(b) captures a program with exactly one cache behaviour, even in the presence of multiple program paths. In particular, the cache behaviour of the program in Figure 2(b) is independent of the program input. This happens primarily due to the fact that the `store` instruction of $p[k]$ will always find $p[k]$ in the cache, irrespective of the value of k. Moreover, the access to $q[127-k]$ or $q[k-64]$ will always be a cache miss due to the initial empty state of the cache. Similarly, the first access to $p[k]$ (i.e., `load reg1, p[k]`) will incur a cache miss. In summary, for all possible values of k, the code fragment of Figure 2(b) incurs two cache misses.

Finally, using the code fragment of Figure 2(c), we show that a single program path may lead to multiple different cache behaviours. In particular, consider the execution of the code in Figure 2(c) with $k = 0$. This leads to the following sequence of memory accesses: $p[0] \rightarrow q[127] \rightarrow p[0]$. Since $q[127]$ replaces $p[0]$ from the cache, the respective execution would incur three cache misses. It is worthwhile to note that for any $k \in [1, 127]$, access to $q[127-k]$ does not replace $p[k]$. As a result, for any $k \in [1, 127]$, the execution of the code in Figure 2(c) suffers two cache misses. This example demonstrates the variation of cache behaviour within a single program path and therefore, the importance of exploring the input partitions with respect to cache performance.

It is worthwhile to note the influence of access-based cache side channel in Figures 2(a)-(c). In Figure 2(a), for k that $k \leq 63$ and k is an even integer, only one cache line, i.e., the one holding $p[0]$ and $q[127]$, would be accessed, while for k that $k \leq 63$ and k is an odd integer, two cache lines, i.e., $p[0](q[127])$ and $p[1]$ would be accessed. For all other values of k, the program will access the first two cache lines, holding $q[127]$ and $p[1]$, respectively. In Figure 2(b), each possible value of $k \in [0, 127]$ manifest different cache-line access statistics. For instance, the cache line holding $p[0]$ and $q[127]$ was accessed with $k = 0$, while cache lines holding $p[1]$ and $q[126]$ were accessed with $k = 1$. In other words, by checking the cache-access statistics of the program in Figure 2(b), an access-based side-channel attacker can accurately determine the value of sensitive input k. A similar exercise on the program in Figure 2(c) will reveal that an access-based cache side channel completely reveals the value of sensitive input k.

The preceding examples demonstrate the non-trivial interaction between cache performance and programming patterns. In particular, a single program path may exhibit variation with respect to cache performance. Similarly, test inputs, that lead to the execution of different program paths, may exhibit the same cache performance. Moreover, depending on the exact nature of the attacker (i.e. timing- or access-based), the leakage of information may vary substantially. For instance, even though the program in Figure 2(b) does not leak any information with respect to a timing-based cache attack, it reveals complete information (i.e. the value of k) with respect to an accessed-based cache attack.

To summarize, it is important to design smart input generation techniques, which specifically focus on exploring different cache behaviours of a program. In this work, we accomplish this via a search-based test generation scheme.

5. METHODOLOGIES

In this section, we describe our test generation methodologies in detail.

DOI: 10.1002/stvr

5.1. Architecture and attack model

In this paper, we only focus on L1 caches, meaning that the attacker can distinguish memory accesses that are L1 cache misses with the memory accesses that are L1 cache hits. Broadly, side-channel attacks are classified into synchronous and asynchronous attacks [50]. In a synchronous attack, an attacker triggers the processing of known inputs (e.g., a plain-text or a cipher-text for encryption routines), whereas this phenomenon is not possible for asynchronous attacks. Synchronous attacks can be mounted easily, since the attacker does not need to compute the start and end of the victim routine. For instance, in a synchronous attack, the attacker can trigger encryption of known input messages and observe the encryption-timing [6]. Since we aim for a test generation tool with the aim of producing side-channel resistant implementations, we assume the presence of a strong attacker mounting synchronous attacks in this paper. Therefore, we assume the attacker can request and observe the execution (e.g., number of cache miss) of the targeted routine. We also assume that the attacker is capable to execute arbitrary user-level code in the same processor running the targeted routine. As a result, the attacker can flush the cache before the targeted routine starts execution and therefore, reduce the external noise in her observations. The attacker, however, is incapable to access the address space of the target routine.

In addition, as illustrated in Section 3 and Section 4, timing- and access-based attacks have similar characteristics in the problem formulation but such attacks exhibit different attacker observations for a specific program (cf. Figure 2(a)-(c)). We hence develop a coverage-directed test generation algorithm that 1) targets both the timing- and access-based cache attacks and 2) it is run separately for each type of side-channel attacks with isolated settings and outcomes. With such an algorithm, we aim to answer the following research questions:

- **RQ1**: How effective is our coverage-directed strategy, based on the formulation of maximal leakage (cf. Formula (2) and Formula (4)), in revealing cache side-channel vulnerabilities in a controlled environment (such as in a processor simulator)?
- **RQ2**: How effective is our coverage-directed strategy, based on the formulation of maximal leakage, in revealing cache side-channel vulnerabilities on real hardware?
- **RQ3**: How efficient is our coverage-directed strategy to generate tests?

5.2. Overview of the algorithm

In this section, we will use the timing-based attacks with the observation of cache hits and misses to illustrate the algorithm we have developed for the test generation. Algorithm 1 provides an outline of it. The central idea of our test generation revolves around a simulated annealing algorithm. Given a program P, let us assume $\{b_1, b_2, \ldots, b_n\}$ capture different bytes for an arbitrary input of the program. We first set an initial solution sol_{init} to initiate our test generation process. Such an initial solution comprises of N_i random values for each input byte b_i. In our case, N_i can be set as a configuration parameter in the test generation. Such a representation of the search space enables us to generate different sets of tests from the same solution, but with different objective values. We note that the objective value, for a set of tests, is defined as the number of unique cache misses exhibited by these tests; as different sets of tests may exhibit different numbers of unique observations (i.e., the objective values), the same solution may generate different sets of tests with different objective values. With this representation of the solution space, the annealing process explores solutions with the higher probability of increasing the unique observed cache misses, i.e., the objective value. If each solution were to represent only one test case, then each solution would always produce an objective value of *one*. Indeed, our evaluation observed a significant improvement by capturing multiple possible tests within one solution representation.

Given the initial solution, we iteratively generate test cases to maximize the unique number of observed cache misses. For any solution sol' generated by Algorithm 1, we randomly select a set of test inputs from sol'. The number of tests, to be selected from sol', is set prior to the test generation process. We run each selected test and record the set of observed cache misses. The cardinality of this set forms the objective value of solution sol'. In Algorithm 1, the objective function is computed and the test-suite $\mathcal{T}'$ is augmented via the procedure `computeTest`. In order to explore the

DOI: 10.1002/stvr

input space, we mutate each solution via the procedure `selectNeighbour`. The probability of selecting a mutated solution depends on the computed objective value of the respective solution and the temperature set for the annealing process. Finally, when searching the input space is completed, the procedure `postProcess` is used to remove test cases having duplicate observations of cache misses in $\mathcal{T}'$ and save the resultant test suite in $\mathcal{T}$. Upon termination of Algorithm 1, the test suite $\mathcal{T}$ is presented to the designer. Each test in $\mathcal{T}$ witnesses a unique cache performance of the program under test.

In the following, we describe some crucial components of out test generation process with timing-based attacks for illustration.

5.3. Initialization of the configuration parameters

The performance of a simulated annealing algorithm crucially depends on the configuration parameters such as t_{init}, t_{final}, α and $trials$ (*cf.* lines 9-12 in Algorithm 1). In our experiments, we first generated a few random executions to systematically set values of these parameters. We set t_{init} to a value where we observed a considerable amount of suboptimal solutions are accepted by our test generator. Such a value of t_{init} is desirable to avoid that the optimization process does not get stuck in a local maxima. In a similar fashion, we set t_{final} to a value where suboptimal solutions are rarely accepted, hence mimicking the exploitation phase in the simulated annealing process. In general, there is a lack of scientific approach to choose the value of α. The value of α should be set in a fashion that the temperature decays slowly and our test generation can explore a significant (but not exhaustive) portion of the input space to converge towards an optimal solution. In our experiments, we set $\alpha = 0.9$. Finally, we compute the value of $trials$ (*i.e.* the number of iterations for a given temperature) in such a fashion that a reasonable number of suboptimal solutions are accepted without slowing down the test generation process dramatically. In the future, we plan to develop a generic approach for systematically obtaining the values of t_{init}, t_{final}, α and $trials$.

5.4. Procedure `setInitialSolution`

In this procedure, we obtain an initial random solution from the input space (*cf.* line 14 in Algorithm 1). This is accomplished by generating N_i random values for each input byte b_i. These values set up the initial solution sol_{init} in our test generation process.

For instance, let us consider a scenario where the program under test is an implementation of AES-128. In AES-128, the length of the secret key is 128 bits or 16 bytes. When testing AES-128 for different secret keys, we first generate a set of random values for each of the sixteen key bytes. Figure 3, for example, captures a scenario where five random values are generated initially for each of the sixteen bytes of AES key.

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}
120	106	81	147	70	222	198	29	134	54	50	132	27	69	150	173
96	215	217	64	201	101	174	139	59	4	61	50	47	101	50	232
154	18	93	228	125	78	21	177	82	236	165	199	89	195	219	163
211	101	180	146	24	86	2	167	195	169	142	237	49	155	49	30
45	235	135	124	144	3	1	102	156	98	160	215	144	95	192	138

Figure 3. An example of a random initial solution generated by the procedure `setInitialSolution`

5.5. Procedure `computeTest`

In this procedure, we randomly generate M tests from a solution, append these tests to a test suite $\mathcal{T}'$ and execute these tests to compute the objective value (*cf.* line 19 in Algorithm 1) for the respective solution. The number of tests, as selected from a solution (*i.e.* M), is pre-configured by the designer. To generate a test from a solution, we select one value at random for each input byte b_i. Since an

DOI: 10.1002/stvr

Algorithm 1 Directed Test Generation for Covering Observations by an Attacker

Input:
P : Program under test
I : Input space of the program under test
C : A cache configuration
Output:
$\mathcal{T}$: A test suite where each $t \in \mathcal{T}$ exhibits a unique cache performance (i.e., the number of cache miss)
/* initialize relevant parameters (see Section 5.3) */
set intermediate test suite $\mathcal{T}' := \emptyset$
set initial temperature $t_{init} > 0$
set final temperature $t_{final} \in (0, t_{init})$
set temperature decay rate $\alpha \in [0, 1)$
set number of trials $trials$ per temperature round
/* set initial solution (see Section 5.4) */
let sol_{init} := `setInitialSolution`(I)
/* iterative test generation */
let $t := t_{init}$
let $sol_{cur} := sol_{init}$
/* compute tests and objective from initial solution (see Section 5.5) */
let $\langle obj, \mathcal{T}' \rangle$:= `computeTest`$(sol_{init}, P, \mathcal{T}')$
while $(t > t_{final})$ **do**
    Let $count := 0$
    while $(count < trials)$ **do**
        /* mutate solution (see Section 5.6) */
        let sol' := `selectNeighbour`(sol_{cur})
        /* compute tests and objective from sol' */
        let $\langle obj', \mathcal{T}' \rangle$:= `computeTest`$(sol', P, \mathcal{T}')$
        if $(obj' > obj)$ **then**
            $sol_{cur} := sol'$
            $obj := obj'$
        else
            select a random value $r \in [0, 1]$
            if $r < e^{\frac{obj' - obj}{t}}$ **then**
                $sol_{cur} := sol'$
                $obj := obj'$
            end if
        end if
        $count := count + 1$
    end while
    $t := t \cdot \alpha$
end while
/* Remove duplicate observations from $\mathcal{T}'$, and keep unique observations in $\mathcal{T}$ */
let $\mathcal{T}$:= `postProcess`$(\mathcal{T}')$
Report $\mathcal{T}$ to the designer

input byte b_i may hold up to N_i values (*cf.* Figure 3) in a solution, this selection is performed from a pool of N_i values for byte b_i.

For the sake of demonstration, let us assume that the designer has set M to be seven. To generate the first test from the solution given in Figure 3, we select one random value from each of the sixteen sets of five values (*cf.* Figure 4(a)). We repeat the same procedure to generate the remaining six test cases. We augment our test suite $\mathcal{T}$ with these seven test cases. We also execute these test cases

DOI: 10.1002/stvr

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}
120	106	81	147	70	(222)	198	(29)	134	54	50	132	27	69	(150)	173
(96)	215	217	(64)	201	101	174	139	59	4	(61)	50	47	101	50	(232)
154	(18)	93	228	125	78	(21)	177	(82)	236	165	199	(89)	195	219	163
211	101	180	146	(24)	86	2	167	195	169	142	(237)	49	155	49	30
45	235	(135)	124	144	3	1	102	156	(98)	160	215	144	(95)	192	138

(a)

$\mathcal{T}'$

166	34	254	68	91	158	79	0	56	233	196	72	102	54	223	154	43
219	156	248	148	97	59	164	175	232	99	101	79	188	54	21	28	47
⋮																
96	18	135	64	24	222	21	29	82	98	61	237	89	95	150	232	48
211	106	180	146	125	78	174	167	59	4	165	50	47	195	219	30	43
154	101	273	228	144	3	2	102	195	169	160	199	27	155	49	163	45
120	18	81	64	201	86	198	167	82	236	61	215	49	69	50	138	43
96	101	81	228	70	101	2	177	134	54	50	237	144	101	192	173	45
45	235	93	147	144	101	1	139	156	4	142	132	49	95	219	232	46
120	215	180	124	201	3	174	102	134	98	165	50	27	195	50	30	48

(b)

Figure 4. (a) Selection of a random input from the solution (the selected values are marked in circles). (b) The test suite $\mathcal{T}'$, augmented with the newly executed seven test cases. The number of cache misses observed, for each test case, is captured via the last column.

120		178		202
96		48		80
154	⊕	9	=	147
211		62		237
45		43		6

(a)

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}
202	139	5	168	86	184	129	16	235	131	26	169	248	80	250	125
80	222	84	21	14	70	120	180	46	207	95	228	98	126	151	153
147	62	40	12	14	116	213	67	161	26	147	154	75	75	186	94
237	55	182	21	81	90	235	227	141	16	46	152	216	35	48	50
6	17	16	179	114	167	120	60	147	28	25	43	17	7	78	47

(b)

Figure 5. (a) A depiction of the bitwise XOR operation flipping the bits for b_1 from the solution stated in Figure 3, (b) The neighboring solution obtained by flipping the bits of all values in the solution stated in Figure 3.

and record the number of cache misses suffered for each of the test case. In Figure 4(b), each row indicates a test case in our test suite, whereas the last column captures the number of cache misses observed by executing the respective test. In order to compute the objective value for the selected seven test cases (as indicated by the last seven rows in Figure 4(b)), we compute the number of unique cache misses observed by these tests. As shown in Figure 4(b), this can be captured by the set $\{43, 45, 46, 48\}$. Therefore, we set the objective value to be $four$ for the selected test cases.

5.6. *Procedure* `selectNeighbour`

In order to obtain a neighboring solution sol' from an arbitrary solution sol_{cur}, we mutate the current solution by flipping the bits of all the values it contains (*cf.* line 24 in Algorithm 1). Recall that each input byte b_i may contain up to N_i values. Therefore, we flip the bits of each of these N_i values to obtain a neighboring solution.

DOI: 10.1002/stvr

We revisit the example in Figure 3. Let us consider the input byte b_1. The five values for input byte b_1 undergo a bitwise exclusive-or (XOR) operation with systematically generated flipping masks, resulting in a new set of 5 values. These flipping masks are generated with the following condition: for each of the values for a given input byte, we gradually reduce the number of bits that are likely to be flipped from eight (when the temperature for the annealing process is t_{init}) to one (when the temperature for the annealing process is t_{final}). This is to facilitate the exploration of a larger neighbourhood during the start of the annealing process while gradually reducing the size of the neighbourhood to explore, as the annealing process progresses. The operation is demonstrated in Figure 5(a). We repeat such bit-flipping procedure for all other input bytes to obtain a neighboring solution sol'. The final solution, after flipping all the input bytes, is captured via Figure 5(b).

5.7. *Applying the algorithm with access-based attacks*

In the preceding sections, we have presented our test generation methodology with timing-based attacks. As to the access-based attacks, we need to replace the objective with a measurable value. We quantify the cache sets accessed as a form of bit vector in the execution of a program, as shown in Table I. Concretely, for access-based attacks, the objective function is modified to maximize $\left| \bigcup_{e \in E} \hat{O}_a^e \right|$ for a set of generated test executions E (cf. Formula 5). Recall that $\hat{O}_a^e$ captures the set of cache sets accessed in test execution e. In Algorithm 1, the number of unique bit vectors, each bitvector capturing the set of cache sets accessed in a test execution, is used as the objective value. The objective value is computed within the function `computeTest`. Upon exiting Algorithm 1, a test suite $\mathcal{T}$ is presented to the designer. Each test in $\mathcal{T}$ witnesses a unique cache access behaviour of the program under test.

5.8. *Notes on the choice of Simulated Annealing*

We have chosen simulated annealing because of its efficiency and feasibility in our experimental setting. The hill-climbing algorithm performs a local search to maximize our chosen objective function (i.e., the number of cache misses or accessed cache sets). Given the complex nature of the interaction between program features and cache behaviours, we envisioned that such a local search may not lead to a good coverage of cache behaviours. Indeed, for our preliminary evaluation, we observed that discarding bad solutions in the neighbourhood causes our test generators to be less effective. Consequently, we decided to use relatively more involved, yet inexpensive input space exploration strategy based on simulated annealing. By using such a strategy, we keep our testing framework efficient for both general-purpose computing and embedded systems. Moreover, this strategy also helps us to take advantage of the annealing process to generate test cases. For example, during the annealing process, our test generator gradually reduces the accessible neighbourhood space as the temperature decreases. A genetic algorithm, on the other hand, would generate a population of candidate solutions and require an objective function to evaluate the population. Since our target applications run both on general-purpose and embedded systems (i.e. Raspberry Pi), we realized that running a population of tests, in each iteration, would become quite expensive for the targeted Raspberry Pi device. Nonetheless, we note the possibility of simulating such device in massively parallel systems, e.g., in the cloud. In such a scenario, it might be feasible to run a parallelized version of genetic algorithm and this can be considered in a future extension of our proposed methodology.

6. EVALUATION

Experimental setup In our evaluation, we have chosen real-world subject programs, including programs from the OpenSSL [40] library and the Linux GDK [43] library. These programs consist of implementations of cryptographic algorithms and key mapping routines. Table II outlines some of the salient features of these programs. The "Size of binary" column indicates the binary size when the respective program is compiled via `gcc 2.7.2.3` with `-O2` optimization and targeting

DOI: 10.1002/stvr

the PISA architecture simulated by Simplescalar [3]. Simplescalar [3] is used as the controlled environment for our experiments. The choice of our subject programs is driven by the fact these programs are widely used in security-critical applications, thus making it essential to validate their security-related properties. We have also selected a basic implementation of AES [14] to stress test our framework against potentially insecure implementations.

We have implemented our test generator as an off-the-shelf tool written in C++. Being *off-the-shelf*, our test generator provides a generic tool that is not constrained to some specific software programs, but can be applied to fuzz various applications. We performed our experiments for all timing-based experiments and access-based experiments in a controlled environment to accurately evaluate the efficacy of our test generation methodologies. A controlled environment is a contrast to a real platform (e.g., a Raspberry Pi or a PC) where multiple programs run concurrently to compete for the CPU cache. In a controlled environment, there is no such disturbance and the execution statistics (e.g., the number of cache misses) is deterministic. As a result, for a given input, a target (sequential) program exhibits the same number of cache misses in a controlled environment. We have also evaluated our test generation scheme for both general-purpose and embedded systems. In particular, we have evaluated the effectiveness of our test generation for timing-based attacks on a 64-bit Intel® Core™ i5-3337U CPU having 4GB memory and running the Debian operating system. As to access-based experiments on real hardware, we used a Raspberry Pi 3 [19]. For access-based attacks, we chose an embedded systems (i.e., Raspberry Pi 3) due to the recent threat of access-based cache attacks on ARM-based embedded systems [36, 24]. We conducted the test generation process as long as our test results (i.e., the number of observations such as cache misses and accessed cache sets) change. For some cases, the convergence of test results took 3,000 iterations and for some other cases the convergence took place after 30,000 iterations. Concretely, we run any of the test generation methodologies until they converge. In order to reproduce results and facilitate research in this direction, we have made our tool and all data publicly available [‡].

Table II. Salient features of the subject programs

Program name	Input size (bytes)	Lines of C code	Size of binary (KB)
`Basic AES [14]`	16	773	29.9
`OpenSSL AES [40]`	16	1,382	64.5
`OpenSSL DES [40]`	8	551	33.5
`OpenSSL RC4 [40]`	10	158	13.0
`GDK-key from name [44]`	4	1,351	45.2
`GDK-key to unicode [45]`	4	1,686	14.9

6.1. *Evaluations on timing-based attacks*

RQ1: Effectiveness in a controlled environment As to timing-based attacks, we set up a controlled environment using Simplescalar [8], which simulates PISA architecture (an MIPS-like architecture). To evaluate our test generator, we compile each subject program into PISA compliant binary. Using the inputs generated by our test generator, we execute these PISA compliant binaries within Simplescalar simulator (using an in-order processor and 2KB L1 cache) and record the number of cache misses.

We compare our test generator with two state-of-the-art fuzz testing tools Radamsa [28] and AFL [54]. Radamsa is a black-box fuzzer and it does not need target software code for test generation. Therefore, we simply let it generate random test inputs by mutating a sample input for a subject program. AFL is a greybox fuzzing tool that generates inputs for a program while executing an instrumented version of the program binary. We note that AFL and Radamsa are not

[‡] https://github.com/tiyashbasu/Cache_Side_Channel_Tester

DOI: 10.1002/stvr

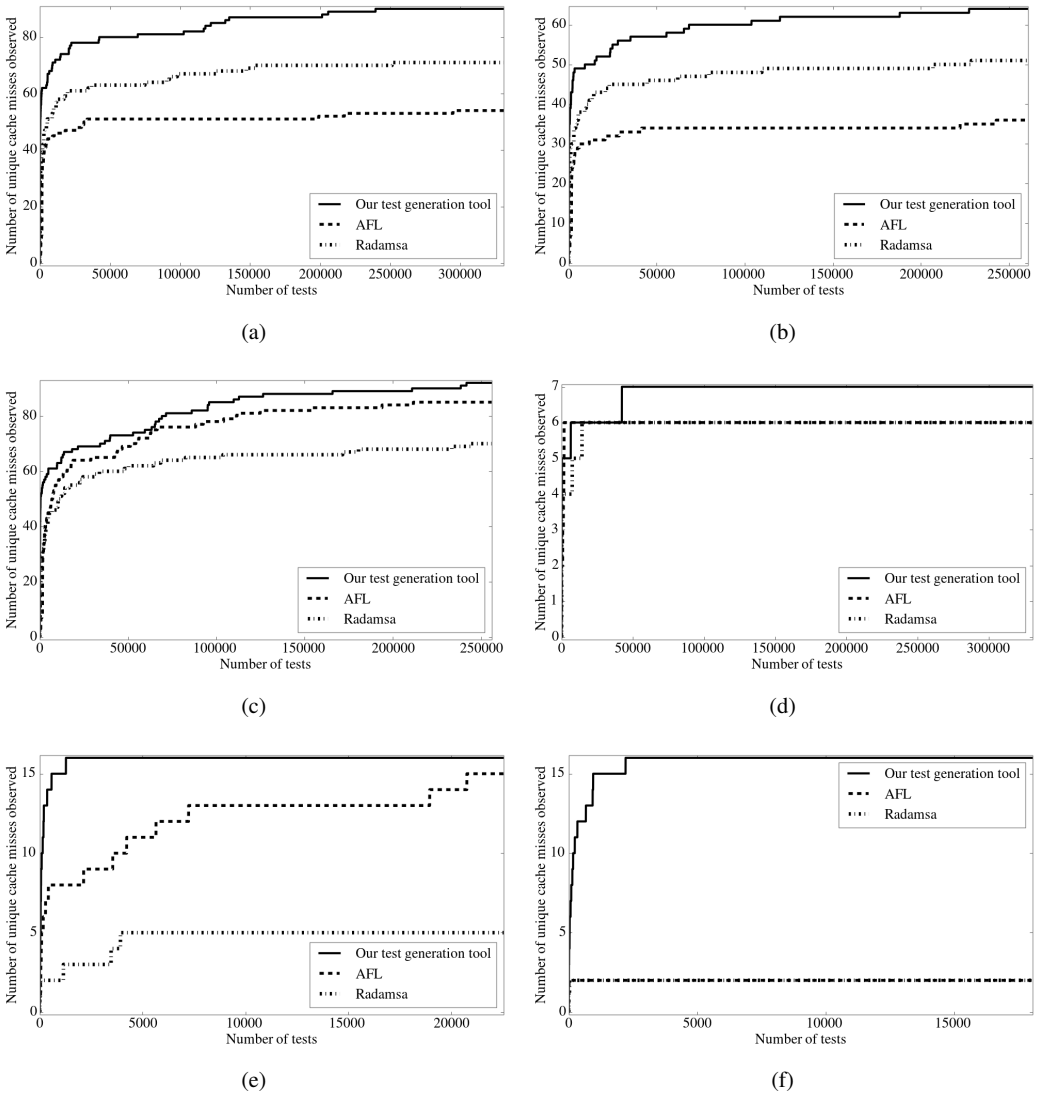

Figure 6. Comparative analyses of different test generators in a controlled environment: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

used for side-channel testing, yet they are used extensively to fuzz security critical programs. Our purpose of using these baseline tools was to demonstrate the need for designing new test generation schemes for side channels, which we propose for the first time with this paper. We do not intend to show that the performance of AFL and Radamsa is suboptimal along the line of their original purpose. In order to compare different test generation schemes, we compare the number of unique cache misses observed with respect to the number of tests generated by each scheme.

Figure 6 presents the results of testing six programs in the controlled environment of Simplescalar. The primary purpose of our test generation is to highlight cache side-channel leakage in arbitrary software binaries. Figure 6 clearly highlights the higher cache side-channel leakage in basic AES and OpenSSL DES, as compared to the OpenSSL version of AES. This is due to the higher number of unique observations reported in basic AES and OpenSSL DES, as compared to the OpenSSL implementation of AES. From Figure 6, we can also observe that, in several scenarios, our approach outperforms the fuzz testing tools by a significant margin. This is expected, as our approach is customized and directed, in order to expose cache side-channel leakage of a program.

DOI: 10.1002/stvr

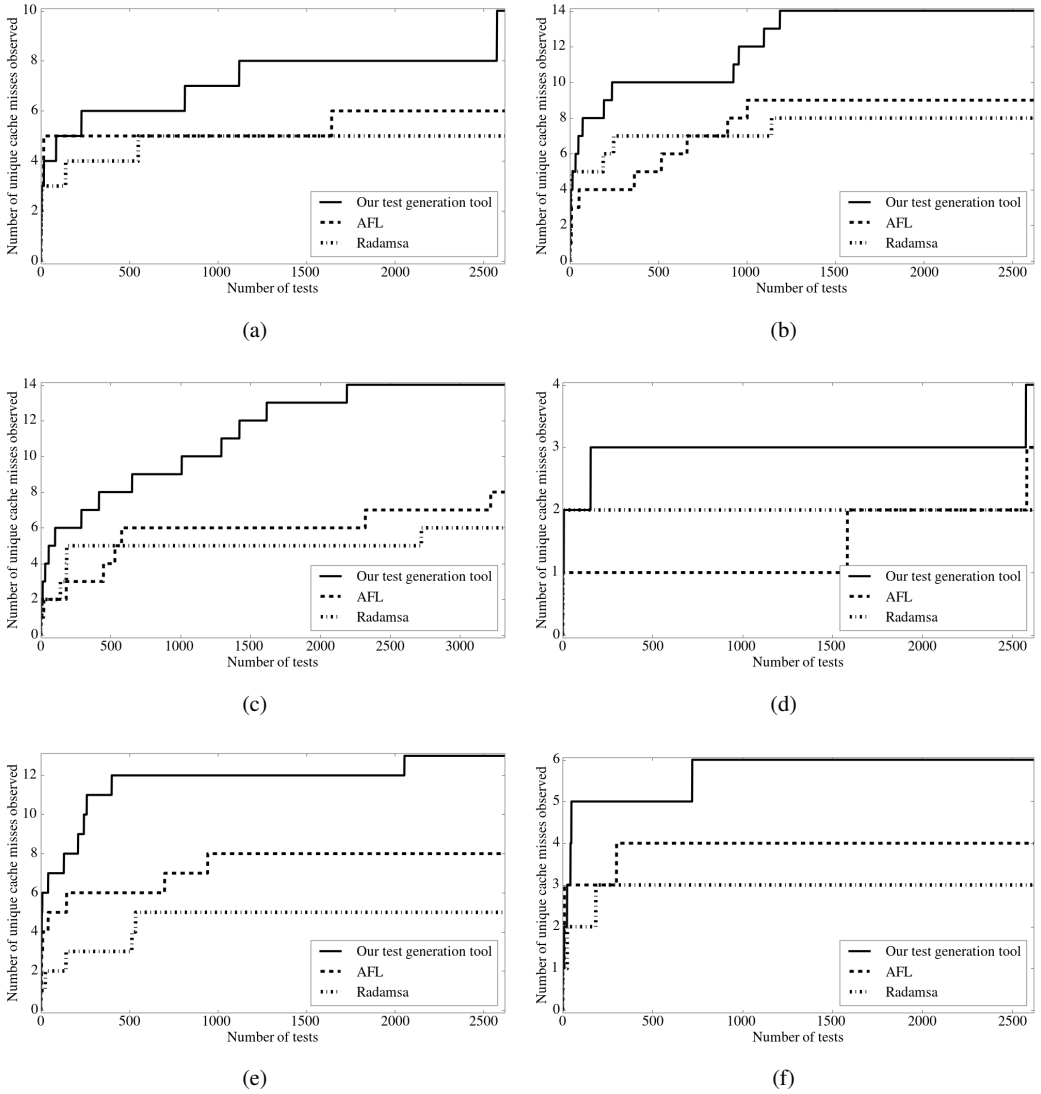

Figure 7. Comparative analyses of different test generators in real hardware: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

This also indicates the requirement of better test generation methodologies that focus on testing cache side-channel leakage.

RQ2: Effectiveness in a real hardware Measuring cache performance in a real hardware is challenging as compared to the same in a controlled environment. This is because, observing cache performance in a real hardware is extremely noisy. Such a noisy behaviour appears due to the following reasons:

1. Binaries compiled for real hardware have additional code introduced by the linker, during the final stages of compilation. These extra code causes cache misses by themselves, thus causing interference in our readings.
2. Current-generation CPUs are multiprocessing, i.e., every CPU core executes multiple kernel threads and user threads in an interleaved fashion. Besides, the existence of both software and hardware interrupts may disrupt the measurement of cache performance.

DOI: 10.1002/stvr

```
/*setting up performance monitoring*/
struct perf_event_attr pe;
pe.type = PERF_TYPE_HW_CACHE;
pe.config = PERF_COUNT_HW_CACHE_L1D
     | (PERF_COUNT_HW_CACHE_OP_READ << 8)
     | (PERF_COUNT_HW_CACHE_RESULT_ACCESS << 16);
//a few more settings done here
int fd = perf_event_open(&pe, 0, -1, -1, 0);

/*enabling performance measurement*/
ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

/*start of the code to test*/
AES_set_encrypt_key(key, 128, &e_key);
AES_encrypt(in, out, &e_key);
/*end of the code to test*/

/*disabling performance measurement*/
ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
```

Figure 8. Instrumenting source code for OpenSSL AES using `perf`.

It is worthwhile to mention that an attacker, who observes cache misses to break an implementation, might employ several algorithms to reduce the noise in her measurements. Therefore, from a software validation perspective, it is critical to understand that the attacker is capable in extracting the number of cache misses suffered by the victim routine. As a result, a software testing tool, in order to expose cache side-channel leakage, should also take appropriate measures in reducing the noise introduced in the observed cache performance.

In order to reduce the noise in measuring cache performance, we perform the following steps. First, we instrument the source code of each subject program to monitor cache misses only for the routines that might be subjected to a cache attack (e.g., an encryption routine). This is accomplished by using Linux utilities `perf_event_open` [21] to set up the cache performance monitoring and `ioctl` [20] to enable and disable the cache performance monitoring. An example of such an instrumentation is depicted in Figure 8. Secondly, we configure the underlying execution platform to isolate a CPU core for running our tests. Of course, such an isolation is only partial. This is because, in spite of a CPU core being isolated, most of the critical kernel threads will still run on it. Therefore, some interrupts will still be directed to the isolated CPU core, resulting interference in the measurement. However, in our evaluation, we observed the noise for such interference is minimal. Finally, we implement a wrapper which runs a subject program with each test input 3,000 times and subsequently, reports the median of all observed cache misses as the final observation. For instance, let us assume that we generate 5 tests and the recorded median values are 100, 200, 300, 200, and 200, respectively. In order to compare the effectiveness of different test generators, we compare the number of unique medians recorded for the generated tests. In this example, therefore, we quantify the effectiveness of the respective test generator as $|\{100, 200, 300\}| = 3$. Note that counting all possible cache misses is not an appropriate metric for quantifying cache side-channel leakage. This is because, the variation of cache misses, for a single program input, only makes a side-channel attack difficult to mount.

Figure 7 demonstrates results obtained in real hardware. The effectiveness of the simulated annealing approach remains better compared to both Radamsa and AFL in all scenarios. However, the absolute effectiveness for the simulated annealing approach (compared to fuzz testing) is less when compared to the results in a controlled environment. This is attributed to the large caches in desktop machines. Due to the large caches, the dependency between cache performance and test input is reduced, resulting in a few unique observations by the attacker. Nevertheless, it is worthwhile to mention that large caches do not eliminate the dependency between cache performance and program inputs. This is because different program paths are likely to exhibit different cache performance. For instance, even though GDK library routines exhibit a small number

DOI: 10.1002/stvr

of cache misses, the variation in the observed cache misses appear due to the varying cache behaviour along different program paths.

RQ3: Efficiency of our test generator Our test generation is directed towards maximizing certain objectives, which is the number of unique cache misses being observed. In particular, we generate different solutions by analyzing the past executions. Therefore, the generation of each test takes much longer on average as compared to fuzz testing. However, a directed approach has the advantage to potentially converge quickly and expose more cache side-channel leakage as compared to a random approach, as observed from Figure 6 and Figure 7. In our evaluation, all experiments for a given subject program, using the simulated annealing, took a maximum of four hours on real hardware. We note that the test generation time includes both the generation time of test inputs and the test execution time (i.e., 3,000 times on real hardware), as our test generation is driven by the measurements obtained from hardware. Indeed, the process of test execution dominates this time. On contrary to our test generator, in the other fuzzing tools (i.e., AFL and Radamsa), the test generation time and test execution time can be easily decoupled. Thus, fuzz testing for these fuzzing tools only took a few minutes to generate the respective number of test inputs. This time does not include the test execution time on real hardware. We believe four hours testing time is acceptable to expose security-related risks for the chosen subject programs in Table II.

6.2. Evaluations on access-based attacks

As to access-based attacks, we still used the Simplescalar to simulate a 2KB L1 data cache and recorded the bit vector of cache lines accessed. As for the real hardware, we used a Raspberry Pi 3 which contains an ARMv7 processor with 16KB L1 data cache (64 cache sets and 256B per set). Raspian Linux was installed in the Raspberry Pi. As a result, the aforementioned `perf_event_open` and `ioctl` for monitoring cache behaviours could still be leveraged for measuring cache behaviour. Isolating executions in one single ARM core was also necessitated since multiple processes might be running concurrently in Raspian. As a relatively small cache is used in a controlled environment (i.e. Simplescalar), we recorded the set of cache lines being accessed for each test execution. For Raspberry Pi, we monitored the set of cache sets being accessed. Therefore, in the controlled environment, our test generator quantifies $\left|\bigcup_{e\in E} O_a^e\right|$, whereas for Raspberry Pi, we revert to our approximation $\left|\bigcup_{e\in E} \hat{O}_a^e\right|$ (cf. Formula 4) due to the closed cache architecture of Raspberry Pi. We have compared our test generation framework to AFL and Radamsa using the same routines as used for testing timing-based attacks.

RQ1: Effectiveness in a controlled environment Figure 9 presents the effectiveness of our test generation for access-based cache attacks. The diagrams in Figure 9 clearly demonstrate that our generation tool outperforms AFL and Radamsa with much more observations generated. Moreover, these diagrams clearly indicate that access-based cache attacks are more powerful than timing-based attacks in terms of leaking secret information from AES and DES. This is due to the significantly larger number of observations generated as compared to the timing-based attacks (cf. Figure 6). It is also worthwhile to note that for programs OpenSSL RC4 and GDK - key to unicode, all three test generation algorithms just generated one observation (i.e. one unique set of accessed cache lines) at runtime, as shown by Figure 9(d) and Figure 9(f). This is because, the simulated cache in Simplescalar always made all generated inputs access the same set of cache lines for OpenSSL RC4 and GDK - key to unicode.

RQ2: Effectiveness in a real hardware In this section, we first discuss the relevant challenges to test access-based cache attacks on real hardware. Subsequently, we discuss a simple yet effective measure to deal with the noise in Raspberry Pi. Finally, we discuss the evaluation of our test generation methodology in Raspberry Pi.

DOI: 10.1002/stvr

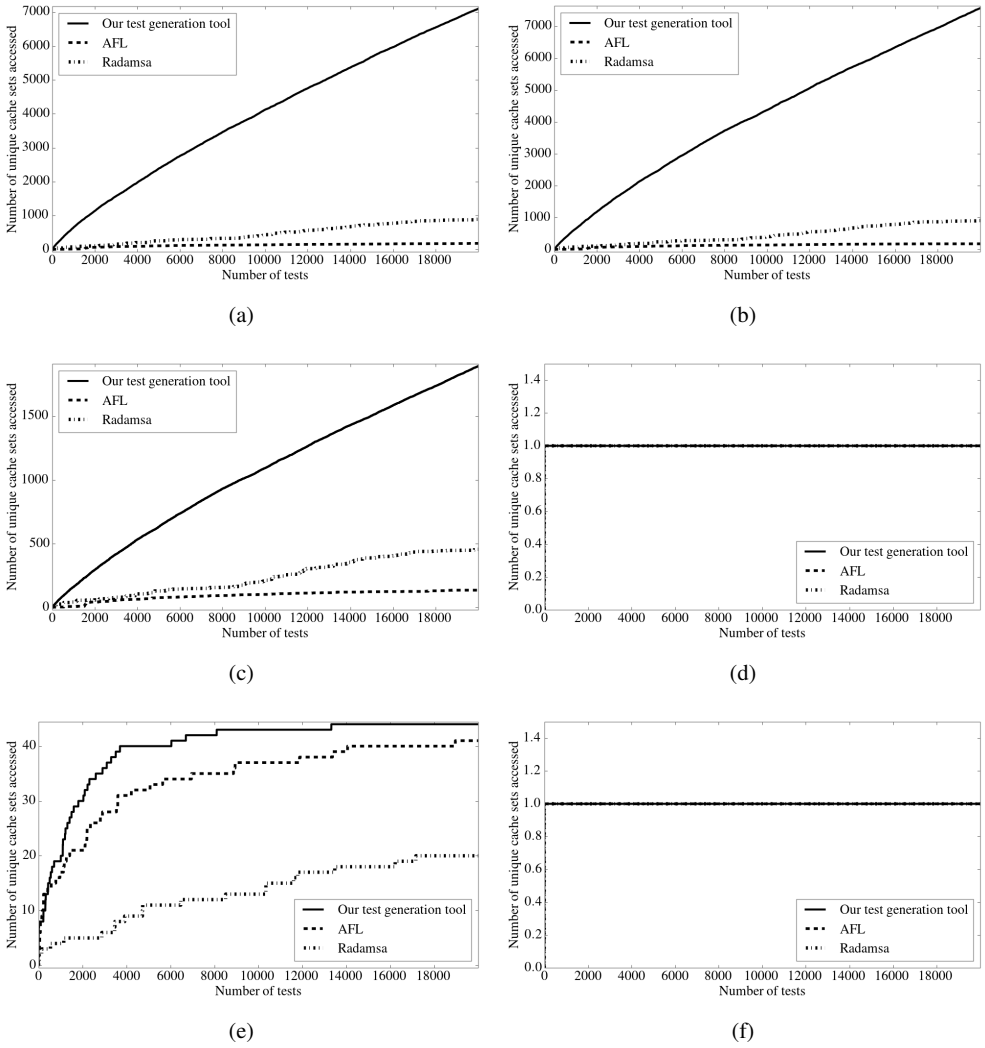

Figure 9. Comparative analyses of different test generators in the controlled environment for access-based attacks: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

Prime + Probe to learn cache sets accessed Unlike Simplescalar, in which we could freely track any cache line, a real-world processor, like the one we were using with Raspberry Pi 3, demands a method to know what cache sets were accessed by a program. To do so we employed the PRIME + PROBE attack [38], which includes two steps. In the first step, i.e., priming the cache, as an attacker, we filled the cache by copying a large amount of our own data (via `memcpy`) so that the cache would become full. Then we ran the victim program (e.g., the encryption routine). The process of priming is shown in Figure 10(a). At this time cache sets of the attacker's program would be replaced in executing encryption program. Hence, in the second step, i.e., probing the cache as illustrated in Figure 10(b), we called `memcpy` again in our attacker program to access each cache set. If the attacker content in a cache set had been replaced (via the encryption routine), then accessing such a cache set will entail cache misses. Concretely, if accessing a cache set incurs a delay equivalent to the cache miss penalty (typically much longer than the cache hit), then we conclude that the respective cache set had been accessed by the encryption program.

We note that we still relied on `perf_event_open` and `ioctl` to determine whether a cache set was hit or not. We repeatedly called the program in Figure 10(b) for 64 cache sets and in the end

DOI: 10.1002/stvr

we could arrive at the bit vector capturing the cache set access statistics. Then, our test generation algorithm based on simulated annealing could proceed with the bit vector as it did in the controlled environment.

```
/* Priming the cache */

memcpy(src, dst, 16384); // Cache size is 16KB

/* Start of Encryption */
aes_key_setup(key, e_key, 128);
aes_encrypt(in, out, e_key, 128);
```

(a)

```
/* Probe the Cache */

/* Enable and start perf */
ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

/* Start of Section - Activity to Measure */
memcpy(src, dst, 256); // Cache set size is 256B
/* End of Section - Activity to Measure */

/* Stop perf */
ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
```

(b)

Figure 10. An Illustration of Prime+Probe for Testing: (a) Priming the Cache (b) Probing the Cache

Rectifying the problem of noise We need to reduce the noise when executing programs in a real hardware platform on which multiple processes are running. However, we can model the noise and, using the probability distribution graph, remove the noise from our observations at a fairly high probability. For modeling the noise in Raspberry Pi environment, we removed all the code which was being executed as part of `perf_event_open` and ran the `ioctl` command. We ran this code 30,000 times to model the noise in measuring the number of cache misses and then plot its frequency distribution curve. This frequency distribution resembled a normal distribution. Thus, we can apply a confidence interval (95 percent or above) from this distribution graph. Let us assume that the mean and the standard deviation for this distribution are μ_0 and σ_0, respectively. Using the standard statistical computation, we have the lower and upper end points of 95% confidence interval as $(\mu_0 - 1.96 \cdot \frac{\sigma_0}{N})$ and $(\mu_0 + 1.96 \cdot \frac{\sigma_0}{N})$, respectively, where N is the size of the test sample. While probing the cache (cf. Figure 10(b)), we measure the number of cache misses to access each cache set for multiple times and get the frequency distribution graph of the number of cache misses incurred. Let us assume that the mean for this distribution is μ and of course, it includes the noise incurred. We then subtract the noise to check the presence of cache misses. Concretely, if we probe the cache for N times, then we check the value of $m = (\mu - \mu_0) - 1.96 \cdot \frac{\sigma_0}{N}$. If $m > 0$, then we conclude that the cache miss occurred. Finally, we note that the aforementioned process is carried out independently for each cache set, as memory accesses do not interfere across cache sets. Thus, after concluding the probe phase, we know the cache sets suffering cache misses. This, in turn, helps us to determine the accessed cache sets by the victim program (e.g., AES). By removing the noise, the end results also closely resemble the actual cache sets accessed for a test execution.

Results Leveraging the technique of Prime+Probe and noise modeling, we performed experiments with our test generation tool, AFL, and Radamsa on the six subject programs. The observation monitored at runtime was the set of unique cache sets accessed. Figure 11 captures results obtained within the Raspberry Pi 3, and confirms the effectiveness of the simulated annealing approach with regards to access-based attacks. In brief, even though for the program GDK-key to unicode our test generator tool yielded suboptimal effectiveness compared to Radamsa, for other five programs it achieved comparable or better observations in terms of the number of unique cache sets accessed. Nevertheless, in a controlled environment, our proposed test generation algorithm proves its effectiveness as compared to AFL and Radamsa. In Raspberry Pi, such effectiveness is impaired for two main reasons. Firstly, in Raspberry Pi, we only measure the set of cache sets being accessed, which is an approximation of the set of accessed cache lines. Secondly, despite our effort in reducing the timing-related noise in Raspberry Pi, we still cannot remove all sources of timing noise in real hardware. This, in turn, affects the effectiveness of all test generation tools.

DOI: 10.1002/stvr

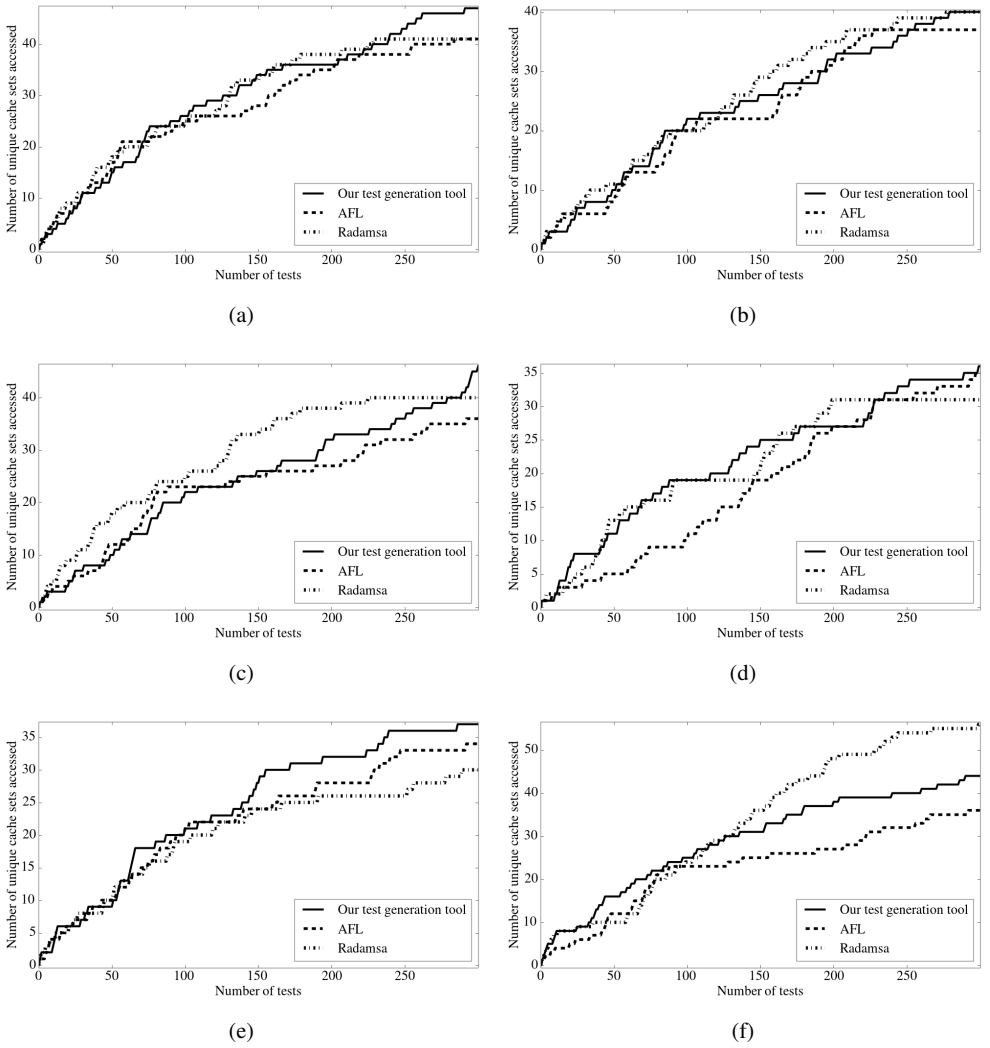

Figure 11. Comparative analyses of different test generators in the real hardware for access-based attacks: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

RQ3: Efficiency of our test generator We have validated the efficiency of our test generator for access-based attacks. Our test generator is directed towards maximizing the objective of the number of unique cache sets accessed by the program. A directed approach helped us resolve the problems that a random approach is unable to address. The time taken by our test generation program to execute 20,000 iterations was approximately 25 minutes, while the fuzz testing tools cost about 6 minutes for the same number of iterations. Note that the time cost of our algorithm for access-based test generation is less than that with timing-based test generation. The temporal overhead is nevertheless acceptable since our test generation tool manages to exhibit much more extensive cache side channel leakages than the state-of-the-art fuzz testing tools.

6.3. Discussion of Evaluation Results

We have performed experiments in line with the three research questions for both timing- and access-based attacks, respectively. As mentioned, the efficiency of testing access-based leakage is higher than that of doing with timing-based leakage. However, a comparison between results of timing- and access-based tests shows that the effectiveness of our test generator varies evidently with

DOI: 10.1002/stvr

respect to the type of attacks. For example, in running OpenSSL RC4 and GDK - key to unicode in the controlled environments, the effectiveness of our test generator, as captured in Figure 6 (d) and Figure 6 (f), is much more promising than the effectiveness shown in Figure 9 (d) and Figure 9 (f), respectively. The results in real hardware platforms for these two program differentiate even more. Such differences indicate that, 1) timing-based attack vulnerabilities are easier to be discovered by testing than access-based attack vulnerabilities, and 2) some programs may need further efforts with customized strategies to be tested. Finally, we can also learn from the evaluation results that, compared to timing-based attacks, the effectiveness for all the three fuzzing approaches is similar for access-based attacks on real hardware platforms. Factually, it is more sophisticated to mount access-based attacks in practice, as they require more knowledge of the underlying hardware and the victim software [27, 51].

7. RELATED WORK

In the last few decades, the research in software testing has made a significant progress. However, the validation of non-functional software properties (e.g., performance and energy) has gained attention only recently. In this paper, we target the validation of security-related software properties, which are critically dependent on the underlying execution platform.

Search and Model-based Testing Search-based testing has a long history in the software engineering community. The most common techniques are hill climbing, simulated annealing, and genetic algorithms [39]. These have been applied extensively to test a variety of software properties. A key contribution of our paper is to reduce the side-channel testing problem into a coverage-guided testing problem. We designed a search-based test generation methodology, based on simulated annealing, to optimize the coverage metric. Although we used simulated annealing for the sake of simplicity and efficiency, other search-based test generation strategies can also be adopted, e.g., genetic algorithms. However, to the best of our knowledge, we propose the first methodology to test the cache side-channel leakage of arbitrary programs in commodity systems. We note that, the CPU cache, as hardware controlled, is more difficult to be monitored during test execution.

Our test generation strategy does not use models for the software systems, in contrast to model-based security testing [18]. However, for our experiments conducted in controlled environments, the simulator uses a cycle-accurate model of the underlying hardware. This is to accurately evaluate the effectiveness of our proposed test methodologies. However, as compared to the existing model-based security testing approaches, we focus on side channels for CPU cache.

Static Analysis of Caches Static cache analysis [49, 12] is an active and challenging research topic. These works propose models of different caches, which in turn can be used to explore the cache behaviour of a program. However, these models are suitable for statically analyzing the timing behaviour of a program and cannot be directly used for side-channel testing. Moreover, these works need to model the cache replacement policy and are not appropriate for commodity systems where the cache architecture is unknown. By contrast, our presented approach does not rely on the model of caches; instead it executes the program and monitors the cache behaviour (e.g., the number of cache misses and the record of accessed cache set) for side-channel testing. As a result, we believe that our work can be leveraged to drive security-related optimizations.

Side-channel Attacks to CPU Cache Cache-based side-channel attacks have emerged to be serious threat for many systems, including but not limited to embedded systems [36]. A detailed account on side-channel attacks has recently been published in a survey [22]. Tools mounting cache side channel attacks employ techniques to guess/ex-filtrate secret information from the execution. This is in stark contrast to our proposed methodology. Our methodology focuses on exploring the input space of the program and quantifying the overall vulnerability of the program against cache side-channel attacks. While exploring the input space, our proposed methodology

DOI: 10.1002/stvr

also generates a test for each observation (e.g., the number of cache misses) explored, whereas the number of observations correlates with the vulnerability against cache side-channel attacks. Our methodology is not designed to be used to speculate about secret information from a test execution. In this paper, we leverage an attacker model that monitors the cache timing [6] and cache accesses [26] to discover sensitive information. However, we believe that the proposed architecture of our test generation is generic and it can be adapted easily to test against more advanced attacking scenarios [53, 25, 1, 7, 31, 37].

Static Analysis of Side Channels for CPU Cache Recently, a few approaches have been proposed to quantify the information leak through cache side channels [16, 33, 32]. These works are based on static analysis and therefore, they suffer from the presence of false positives. Since our approach is based on testing, it does not generate any false positive. Moreover, we generate witnesses for each observed cache behaviour. These witnesses can further be used for testing and debugging.

Testing Side Channels In the past year, research in software testing has focused on using symbolic execution and Max-SMT to quantify side-channel leakage [42]. In contrast to our test generator, this work does not take into account side-channel leaks through micro-architectural entities, such as caches. Moreover, we also evaluate our test generator to validate the cache side-channel leakage in real hardware.

Testing and Dynamic Analysis of Side Channels for CPU Cache A recent approach based on dynamic analysis [11] quantifies cache side-channel leakage from execution traces that record number of misses and cache access statistics. Specifically, such an analysis can be used to guess the secret input from execution traces. Our work is complementary in the sense that it can be used with the existing dynamic analysis [11] to guess the secret key for each generated test execution. However, in contrast to our work, the proposed dynamic analysis approach [11] requires a symbolic model of the cache, which is difficult to obtain without a complete information about the underlying cache architecture. In contrast to symbolic testing of cache side-channel [10], the approach proposed in this paper does not explicitly model hardware caches. Instead we learn cache behaviour on-the-fly and therefore, we can show the application of the approach also on real hardware. In our previous approach [5], we proposed a search-based test generation methodology to validate the cache side-channel leakage of arbitrary programs. In this work, we extend the approach for access-based cache attacks and show the effectiveness of our approach in both the controlled environment as well as in a real embedded system.

Verification of Side Channels for CPU Cache Our approach is orthogonal to works related to the verification of constant-time cryptographic software [4, 2]. In particular, our approach targets arbitrary binary code and is not limited to the verification of constant-time cryptographic software. Besides, our proposal has a significant flavour of testing and debugging, as we generate witnesses for observed cache behaviour through directed test generation. In contrast to recent works on verifying cache side-channel freedom [13], our approach on testing is oblivious to cache replacement policies and its effectiveness is evaluated on real-world general-purpose as well as embedded systems.

Countermeasures to Throttle Side Channels for CPU Cache Our work is orthogonal to approaches that propose countermeasures against side-channel attacks [52, 48, 15]. Of course, we believe that the open platform provided by our work can be utilized as a valuable tool to validate existing and new countermeasures. In particular, as we target arbitrary binary code, we can use our test generator to discover potential flaws in countermeasures proposed to mitigate cache side channels.

In summary, we have proposed a test generation framework to validate arbitrary software against cache-based side-channel attacks. To the best of our knowledge, this is the first search-based approach that systematically discovers witnesses to validate cache side-channel leaks of a program.

DOI: 10.1002/stvr

8. THREATS TO VALIDITY

Our framework does not exhibit false positives, therefore the computed cache side-channel leak indeed appears in real execution. However, we conducted our test generation process until they do not reveal any new observation (e.g., cache miss or accessed cache set) for sufficient time. However, this does not guarantee the absence of any new attacker observation. Thus, the number of observations exposed by our test generator imposes a lower bound on the information leakage. This means our framework should not be used to prove the absence of cache side-channel leak. Software systems, that must adhere to zero leakage, can leverage our test generator to discover implementation flaws early during the design.

In this paper, we have targeted cache timing attacks [6] and access-based cache attacks [26]. There exists other cache attacks [1, 53, 31, 37] not covered in this paper. Therefore, our test generator cannot be used directly to validate software systems against such cache attacks. However, we believe that our test generation strategy is quite generic and it can be adapted easily to account for other cache attacks by reformulating the objective function of a solution. Besides, the open platform of the test generator facilitate research in this direction and improve the state-of-the-practice in testing software non-functional properties.

As discussed in the evaluation, it is virtually impossible to reduce all noise in measuring cache performance for complex execution platforms. We have reduced the impact of noise in the evaluation via running a single test multiple times, using statistical metrics, isolating executions in a single core, modeling the noise in measurements and using hardware performance counters. Finally, although we repeated measurements for a single test input in the order of thousands, it might be insufficient for certain hardware platforms.

Finally, we note that each test generated by our methodology corresponds to a unique attacker observation (i.e., the number of cache misses or accessed cache set). Thus the number of unique observations, as explored by our test suite, captures the channel capacity of the *overall program*. This metric, by no means, captures the amount of information being leaked for *a given execution*. In our prior work [11], we show that quantifying the information leakage, for a given execution, is quite involved and requires different machineries to solve the problem. Thus, even though our test generation process quantifies the overall side-channel vulnerability (i.e., the channel capacity) of a program, we may not conclude the side-channel vulnerability of a given execution of the same program from our test generation process.

9. CONCLUSION AND FUTURE WORK

In this paper, we have introduced the test generation problem to validate cache side-channel leakage of an arbitrary software with regards to both timing- and access-based attacks. We have shown that such a problem differs from classic program-path exploration problems. The key insight behind the test generation problem is to systematically explore the cache behaviour. Since cache behaviour critically depends on the underlying execution platform, it is crucial for such a test generator to understand the influence of execution platforms on the generated tests. Following this insight, we have designed a simulated-annealing-based test generation algorithm in order to expose the cache side-channel leakage of a program. Our evaluation highlights cache side-channel leakage in real-world programs from OpenSSL and Linux GDK libraries, both on a simulated environment and on a real hardware. We also show that our directed approach is more effective in revealing cache side-channel leakage than state-of-the-art fuzz testing tools. Following this result, we believe that other search-based testing approaches, such as genetic programming, will also be effective in exposing cache side-channel leakage of software.

The key intuition in this paper reflects on the importance of exploring the interaction between software systems and the underlying execution platform. This is critical to understand several other non-functional properties, such as performance, energy and robustness among others. Therefore, we believe that we can extend our work in several directions to validate software non-functional properties. We plan to use several optimizations to improve the annealing process. In this fashion,

DOI: 10.1002/stvr

we can generate a more efficient test generation tool, which is also directed to expose side-channel leakage of arbitrary programs. In particular, we also plan to leverage machine learning to understand the behaviour of execution platform and design better test generation methodologies that target software non-functional properties. We further plan to investigate appropriate synergies between symbolic execution and search-based methods in this direction. Finally, we aim to use the power of our test generator to detect timing and access covert channels that attackers can leverage to stealthily fetch data for analysis and speculation when a program is running [35, 38]. We believe this is possible, as our test generator explores timing and access behaviours of a program and any significant deviation from such timing and access can be detected efficiently at runtime. For reproducibility and further research, the experimental data and our tool are made publicly available:

https://github.com/tiyashbasu/Cache_Side_Channel_Tester

Acknowledgement We thank the anonymous reviewers for their constructive feedback in an earlier version of this paper. This work is partially supported by the Ministry of Education of Singapore under the grant MOE2018-T2-1-098.

REFERENCES

1. Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 112–121, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
2. Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, 2016. USENIX Association.
3. Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.
4. Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM.
5. Tiyash Basu and Sudipta Chattopadhyay. Testing cache side-channel leakage. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 51–60, 2017.
6. Daniel J Bernstein. Cache-timing attacks on AES, 2005.
7. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
8. Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.
9. Ashokkumar C, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 261–275, March 2016.
10. Sudipta Chattopadhyay. Directed automated memory performance testing. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, pages 38–55, Berlin, Heidelberg, 2017. Springer-Verlag.
11. Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leak in cache attacks via symbolic execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 25–35, 2017.
12. Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 49(4):517–562, 2013.
13. Sudipta Chattopadhyay and Abhik Roychoudhury. Symbolic verification of cache side-channel freedom. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(11):2812–2823, 2018.
14. Brad Conte. Advanced Encryption Standard Implementation. https://github.com/B-Con/crypto-algorithms. Last accessed on 04/16/2015.
15. Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.
16. Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C., 2013. USENIX.
17. Diaa Salama Abd Elminaam, Hatem Mohamed Abdual-Kader, and Mohiy Mohamed Hadhoud. Evaluating the performance of symmetric encryption algorithms. *International Journal of Network Security*, 10(3):213 – 219, 2010.
18. Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: a taxonomy and systematic classification. *Softw. Test., Verif. Reliab.*, 26(2):119–148, 2016.

DOI: 10.1002/stvr

19. Raspberry Pi Foundation. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Last Accessed on 02/17/2016.
20. The Linux Foundation. ioctl. http://man7.org/linux/man-pages/man2/ioctl.2.html. Last Accessed on 05/19/2017.
21. The Linux Foundation. perf_event_open. http://man7.org/linux/man-pages/man2/perf_event_open.2.html. Last Accessed on 06/11/2017.
22. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. In *Cryptology ePrint Archive*, 2016.
23. Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 444–461, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
24. Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 1075–1091, 2017.
25. Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., 2015. USENIX Association.
26. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.
27. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.
28. Aki Helin. Radamsa. https://github.com/aoh/radamsa. Last accessed on 05/02/2017.
29. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 591–604, Washington, DC, USA, 2015. IEEE Computer Society.
30. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 189–204, Bellevue, WA, 2012. USENIX.
31. Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
32. Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 286–296, 2007.
33. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 564–580, Berlin, Heidelberg, 2012. Springer-Verlag.
34. Armin Krieg, Johannes Grinschgl, Christian Steger, Reinhold Weiss, and Josef Haid. A side channel attack countermeasure using system-on-chip power profile scrambling. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 222–227, July 2011.
35. Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.
36. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 549–564, 2016.
37. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
38. Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.
39. Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
40. OpenSSL. OpenSSL Library. https://github.com/openssl/openssl/tree/master/crypto. Last Accessed on 05/13/2018.
41. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.
42. Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 387–400, June 2016.
43. The GNOME Project. GDK keyboard handling library. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html. Last Accessed on 02/05/2017.
44. The GNOME Project. gdk_keyval_from_name. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html#gdk-keyval-from-name. Last Accessed on 02/05/2017.
45. The GNOME Project. gdk_keyval_to_unicode. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html#gdk-keyval-to-unicode. Last Accessed on 02/05/2017.
46. Xin Qiu, Eric J Sprunk, Daniel Z Simon, Lawrence Tang, and Lawrence R Cook. Countermeasure to power attack and timing attack on cryptographic operations, October 12 2004. US Patent 6,804,782.

47. Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys Tutorials*, 20(1):465–488, Firstquarter 2018.
48. Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*, pages 718–735. Springer, 2013.
49. Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3), 2000.
50. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
51. Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, Vancouver, BC, 2017. USENIX Association.
52. Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.
53. Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time RSA. *J. Cryptographic Engineering*, 7(2):99–112, 2017.
54. Michal Zalewski. American fuzzy lop (AFL). `http://lcamtuf.coredump.cx/afl`. Last Accessed on 11/02/2017.

DOI: 10.1002/stvr