Testing Cache Side-channel Leakage

Tiyash Basu
Saarland University
tiyashbasu@gmail.com

Sudipta Chattopadhyay
Singapore University of Technology and Design (SUTD)
sudipta_chattopadhyay@sutd.edu.sg

***Abstract*—Cache timing attacks retrieve secret information (*e.g.* a secret key) about a program by analyzing the cache behaviour in program executions. It is, therefore, crucial to understand whether a program is vulnerable to cache timing attacks. But how can we test a program to discover its vulnerability against cache timing attacks? In this paper, we propose, design and evaluate a test generation methodology that systematically discovers the cache side-channel leakage of arbitrary software binaries. At the core of our test generation is a method that systematically explores the program input space and it adapts based on the observed cache performance in the executed tests. We have implemented our test generator and evaluated it with several open-source subject programs, including programs from OpenSSL and Linux GDK libraries. Our evaluation effectively reveals cache side-channel leakage in such real-world programs. We also empirically show that our test generator is more effective in revealing cache side-channel leakage than traditional fuzz testing tools Radamsa and AFL.**

I. Introduction

Side-channel attacks aim to retrieve secret features of a program execution (*e.g.* a secret key) without knowing its functional input or output. Among others, cache timing attacks [12] have emerged to be a serious security breach in real-world software systems. The key intuition behind a cache timing attack is to observe the timing of cache hits and misses in a program execution, and subsequently use this timing to determine the secret features of the respective program. The disclosure of such secret information to an untrusted party may have disastrous consequences, often resulting in a complete breakdown of the overall system. Therefore, it is crucial to validate software systems against potential cache timing attacks.

For a given program, its vulnerability to cache timing attacks depends on the amount of information that can leak through its cache performance. The cache performance of a program, in turn, is critically influenced by the underlying execution platform. Unfortunately, the state-of-the-art in software testing is far from being matured to validate software properties, such as cache performance, that critically depends on the execution platform. In this paper, we take a step forward to bridge this gap in software testing. Given a program and a cache configuration, we formulate the test generation problem to validate software systems against cache timing attacks. Based on 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.

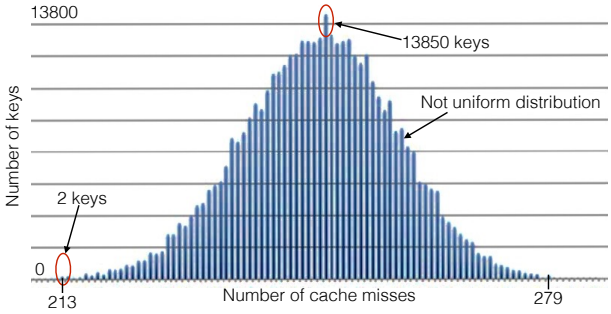


Fig. 1. For a fixed input message, the plot shows the distribution of the number of keys with respect to a given number of cache misses. The experiment was performed for an implementation of AES-128 [1] for 256,000 different keys (picture taken from [15])

In order to understand the challenges involved in testing cache side-channel leakage, consider the illustration in Figure I. Figure I demonstrates the execution of an implementation of Advanced Encryption Standard (AES) [1] for a fixed plaintext message and 256000 different keys. 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 keys that induce them. Figure I clearly shows that the distribution of cache misses is essentially a *gaussian*. 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 I reveals the following challenges: First, all 256,000 executions in Figure I exercise the *same program path*. Therefore, merely exploring program paths is not sufficient to explore different cache behaviour of the respective program. Since it is critical to explore the different cache behaviour 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 extreme cache behaviour shown in Figure I. In general, the cache behaviour of a program may vary significantly, with some cache behaviour 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 cache behaviour (hence, common timing behaviour), but also the infrequent cache behaviour (hence, exceptional timing behaviour).

In this paper, we design and evaluate a test generation scheme, based on simulated annealing, to address the chal-

lenges mentioned in the preceding paragraphs. The output of our framework is a test suite, where each test in it witnesses a unique cache timing of the program. 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 attacks. Our work significantly differs from the work in static analysis of cache side channels [17], [22]. In particular, our test generation process does not exhibit any false positives, meaning that each test in the test suite serves as a witness of a cache behaviour in real executions. 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 that searching input space is non-trivial to test cache side-channel leakage, it is also the test execution that makes the testing problem challenging. For each generated tests, 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 a potential attacker may employ several noise reduction techniques herself to mount an attack. In order to reduce the noise in test execution, we use the following techniques: Firstly, we use a controlled environment where the execution statistics (*e.g.* the number of cache misses) is deterministic. Secondly, in real hardware, we leverage performance counters, statistical methods and explicitly introduce instructions to isolate execution in a single core. This, in turn, reduces the noise when measurements are taken from real hardware.

The remainder of the paper is organized as follows. We give an overview of the problem in Section III and we make the following contributions in this paper:

1) We formulate the test generation problem and an appropriate test coverage criteria for validating cache side-channel leakage of an arbitrary program (Section II).
2) We design a test generation algorithm that aims to search the program input space to explore different cache behaviour and subsequently, reveal the cache side-channel leakage of a program (Section IV).
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 [6] and Linux GDK library [2] in a controlled environment (using simplescalar [10] simulator) as well as in real hardware. We also compared our test generator with state-of-the-art fuzz testing tools Radamsa [21] and AFL [28]. Our evaluation effectively reveals cache side-channel leaks in all the chosen subject programs and our directed approach in test generation outperforms (in terms of revealing cache side-channel leakage) both Radamsa and AFL (Section V).

We conclude this paper with threats to validity (Section VII) and consequences (Section VIII).

II. Test Generation Problem

Cache side-channel attacks exploit the performance gap between caches and main memory (DRAM) to discover sensitive information. 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. Cache side-channel attacks are often non-invasive and they can even be mounted easily over the network [12]. In this paper, we choose timing-related attacks. In such side-channel attacks, the attacker monitors the number of cache misses incurred in an execution and employs statistical techniques to discover secret information [12].

We assume the cache side channel to be a function $C : \mathbb{I} \rightarrow \mathbb{O}$. The function C maps a finite set of sensitive inputs to a finite set of observations. Since the attacker monitors the number of cache misses, in this scenario, an observation $o \in \mathbb{O}$ captures the number of cache misses in an execution. 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 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 [22].

$$ML(C) \leq \log_2 |C(\mathbb{I})| \tag{1}$$

where $ML(C)$ captures the maximal leakage of channel C. In Equation 1, equality holds when X is uniformly distributed.

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.* Equation 1). 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 and discover its potential side-channel leakage. 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.* Equation 1). Therefore, 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 Equation 1.

III. Overview

In this section, we motivate the challenges in test generation through examples. Since memory performance is accurately

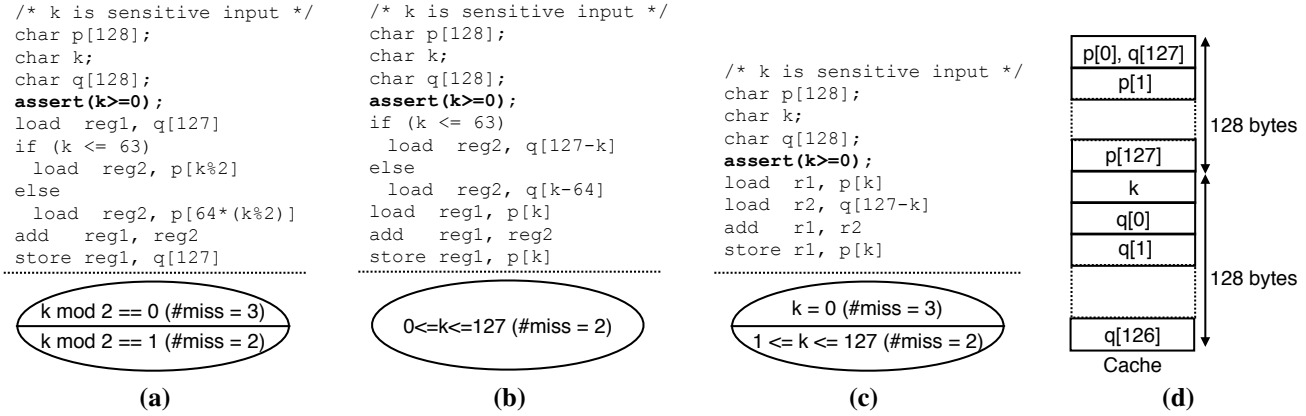


Fig. 2. k is a sensitive input taking only positive values. (a)-(c) three code fragments and respective partitions of the input space with respect to the number of cache misses ($reg1$, $reg2$ represent registers), (d) mapping of program variables into a direct-mapped cache sized 256 bytes ($q[127]$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

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).

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.

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. Therefore, 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.

IV. Methodologies

A. 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 [26]. In 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 synchronous attack, the attacker can trigger encryption of known input messages and observe the encryption-timing [12]. 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 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.

B. Overview of the algorithm

Algorithm 1 provides an outline of our test generation framework. 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 set of tests from the same solution, but with different objective values. This is because, the objective value, for a set of tests, is defined as the number of unique cache misses exhibited by these tests. With this representation of the solution space, the annealing process explores solutions with the higher probability of increasing the unique observed cache misses and hence, 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 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.

Algorithm 1 Directed Test Generation for Covering Observations by an Attacker

```
1:  Input:
2:  P : Program under test
3:  I : Input space of the program under test
4:  C : A cache configuration
5:  Output:
6:  T: A test suite where each t ∈ T exhibits a unique cache
7:  performance (i.e. the number of cache miss)
8:
9:  /* initialize relevant parameters */
10: /* (see Section IV-C) */
11: set intermediate test suite T' := φ
12: set initial temperature t_init > 0
13: set final temperature t_final ∈ (0, t_init)
14: set temperature decay rate α ∈ [0, 1)
15: set number of trials trials per temperature round
16: /* set initial solution */
17: /* see Section IV-D */
18:  let sol_init := setInitialSolution(I)
19:
20: /* iterative test generation */
21: let t := t_init
22: let sol_cur := sol_init
23: /* compute tests and objective from initial solution */
24: /* (see Section IV-E) */
25:  let ⟨obj, T'⟩ := computeTest(sol_init, P, T')
26: while (t > t_final) do
27:     Let count := 0
28:     while (count < trials) do
29:         /* mutate solution */
30:         /* (see Section IV-F) */
31:         let sol' := selectNeighbour(sol_cur)
32:         /* compute tests and objective from sol' */
33:         let ⟨obj', T'⟩ := computeTest(sol', P, T')
34:         if (obj' > obj) then
35:             sol_cur := sol'
36:             obj := obj'
37:         else
38:             select a random value r ∈ [0, 1]
39:             if r < e^((obj'−obj)/t) then
40:                 sol_cur := sol'
41:                 obj := obj'
42:             end if
43:         end if
44:         count := count + 1
45:     end while
46:     t := t · α
47: end while
48: /* remove duplicate observations from T' */
49: /* keep unique observations in T */
50: let T := postProcess(T')
51: Report T to the designer
```

C. 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 12-15 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 $trials$.

D. Procedure `setInitialSolution`

In this procedure, we obtain an initial random solution from the input space (*cf.* line 18 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

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

E. 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 25 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.

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)

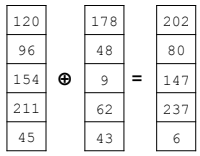
Fig. 4. (a) Selection of a random input from the solution (the selected values are marked in circles). (b) The test suite $\mathcal{T}'$, augmented with the newly executed seven test cases. The number of cache misses observed, for each test case, is captured via the last column.

To generate a test from a solution, we select one value at random for each input byte b_i. Since an 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 illustration, 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 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.

F. Procedure `selectNeighbour`

In order to obtain a neighbouring 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 31 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 neighbouring solution. Additionally, for each of these N_i values, we gradually reduce the number of bits that are likely to be flipped from *eight* (when the temperature is t_{init}) to *one* (when the temperature is t_{final}). This, in effect,



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)

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

gradually reduces the accessible neighbourhood space as the temperature decreases.

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. This operation is demonstrated in Figure 5(a). We repeat such bit-flipping procedure for all other input bytes to obtain a neighbouring solution sol'. The final solution, after flipping all the input bytes, is captured via Figure 5(b).

V. Evaluation

A. Experimental setup

In our evaluation, we have chosen real-world subject programs, including programs from OpenSSL [6] library and Linux GDK [2] library. These programs consist of implementations of cryptographic algorithms and key mapping routines. Table I outlines some of the salient features of these programs. The choice of our subject programs is driven by the fact these programs are widely used in security-critical applications, this making it essential to validate their security-related properties. We have also selected a basic implementation of AES [1] to stress test our framework against potentially insecure implementations.

We have implemented our test generator as an off-the-shelf tool written in C++. We performed all our experiments on a 64-bit Intel® Core™ i5-3337U CPU having 4GB memory and running Debian operating system. In order to reproduce results and facilitate research in this direction, we have made our tool and all data publicly available in the following URL:

https://github.com/tiyashbasu/Cache_Side_Channel_Tester

B. Research questions

Through our evaluation, we aim to investigate the following research questions:

1) **RQ1:** How effective is our test generator in revealing cache side-channel leaks in a controlled environment (such as in a processor simulator)?

Program name	Input size (bytes)	Lines of C code	Size of binary (KB)
Basic AES [1]	16	773	29.9
OpenSSL AES [6]	16	1382	64.5
OpenSSL DES [6]	8	551	33.5
OpenSSL RC4 [6]	10	158	13.0
GDK-key from name [3]	4	1351	45.2
GDK-key to unicode [4]	4	1686	14.9

TABLE I
Salient features of the subject programs

2) **RQ2:** How effective is our test generator in revealing cache side-channel leaks in real hardware?
3) **RQ3:** How efficient it is compared to state-of-the-art fuzz testing tools?

C. RQ1: Effectiveness in a controlled environment

We setup a controlled environment using simplescalar [10], 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 [21] and AFL [28]. Radamsa is a black-box fuzzer and it does not need target software 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. 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 demonstrates our observation. The primary purpose of our test generation is to highlight cache side-channel leakage in arbitrary software binaries. For instance, 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. We 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. This also indicates the requirement of better test generation methodologies that focus on software non-functional properties, such as cache side-channel leakage.

D. 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

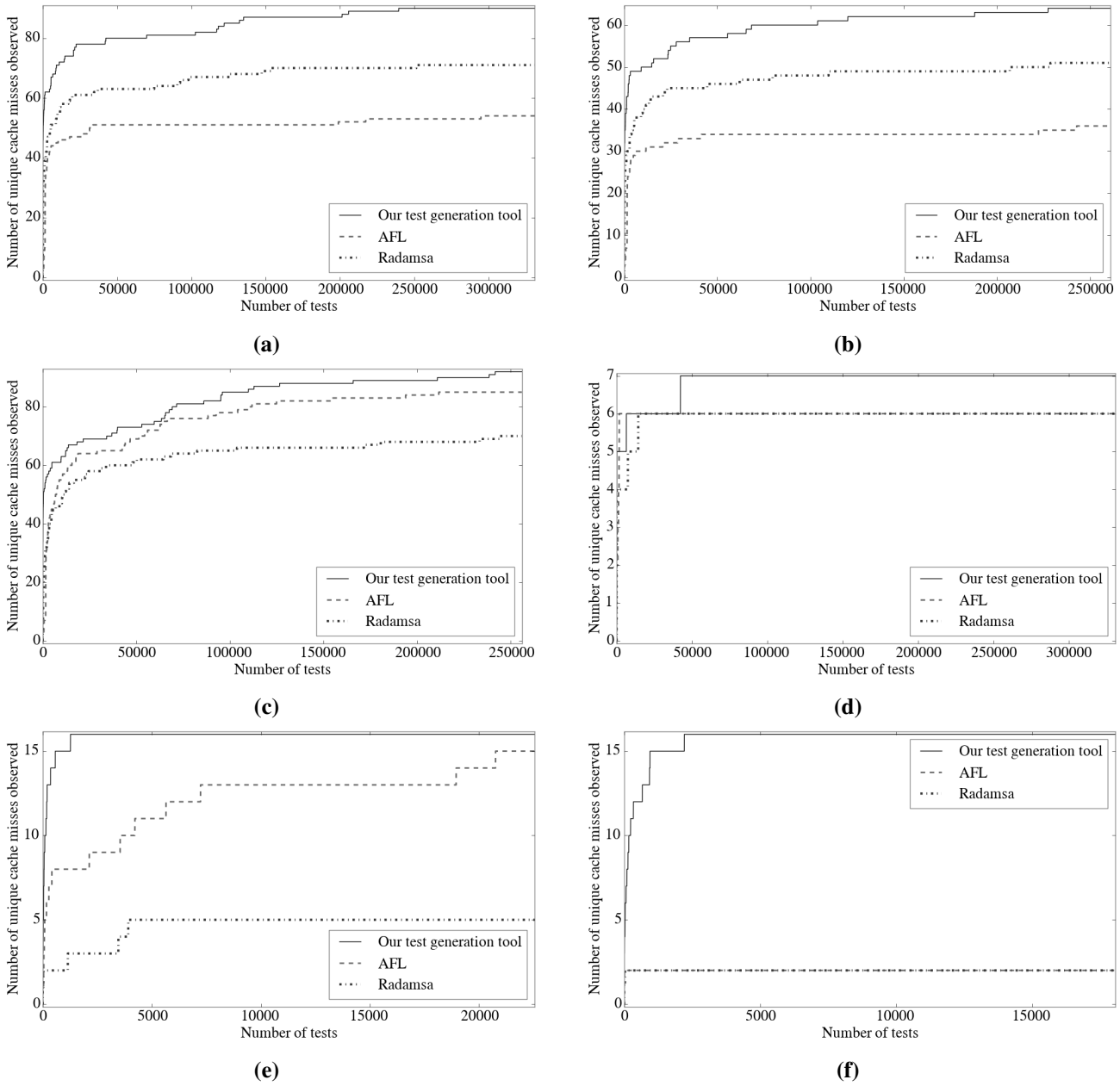


Fig. 6. Comparative analyses of different test generators in a controlled environment: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

of compilation. These extra code cause 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.

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` [7] to set up the cache performance monitoring and `ioctl` [5] to enable and disable the cache performance monitoring. An example

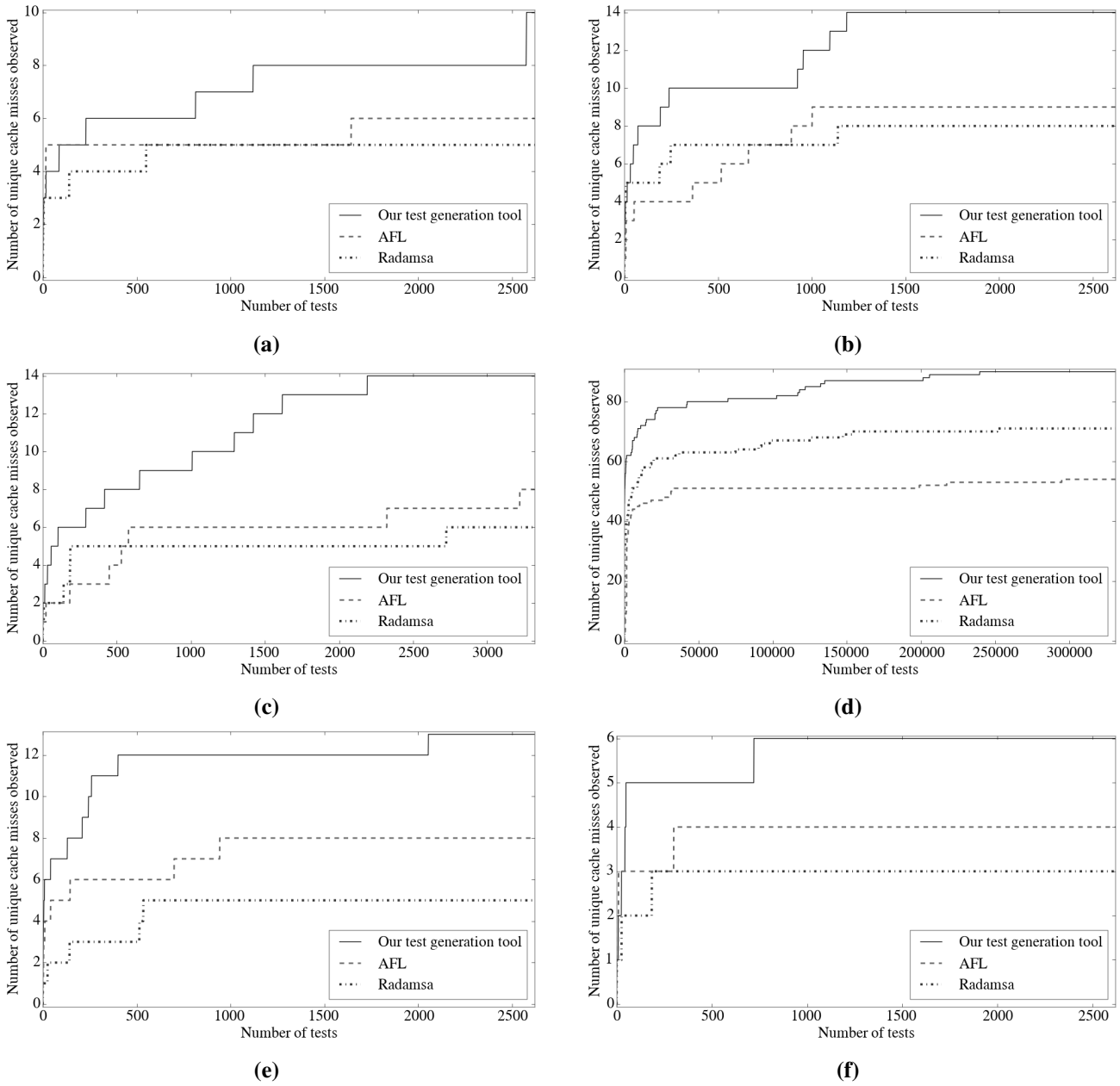


Fig. 7. Comparative analyses of different test generators in real hardware: (a) Basic AES (b) OpenSSL AES (c) OpenSSL DES (d) OpenSSL RC4 (e) GDK - key from name (f) GDK - key to unicode

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 3000 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

```
/*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);
```

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

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

E. 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. In contrast, fuzz testing only took a few minutes to generate all test inputs. We believe four hours testing time is acceptable to expose security-related risks for the chosen subject programs in Figure I. Besides, we plan to use several optimizations to improve the annealing process. In this fashion, we can generate a more efficient test generation tool, which is also directed to expose side-channel leakage of arbitrary programs.

VI. 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.

Cache-based side-channel attacks have emerged to be serious threat for many systems, including but not limited to embedded systems. A detailed account on side-channel attacks has recently been published in a survey [18]. In this paper, we leverage an attacker model that monitors the cache timing to discover sensitive information [12]. 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 [8], [13], [19], [20].

Recently, a few approaches have been proposed to quantify the information leak through cache side channels [17], [22]. 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. Our approach is orthogonal to works related to the verification of constant-time cryptographic software [9], [11]. In particular, our approach targets arbitrary binary code and it is not limited to the verification of constant-time cryptographic software. Besides, our proposal has a significant flavor of testing and debugging, as we generate witnesses for observed cache behaviour through directed test generation. In the past year, research in software testing has focused on using symbolic execution and Max-SMT to quantify side-channel leakage [23]. 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.

Our work is orthogonal to approaches that propose countermeasures against side-channel attacks [16], [24], [27]. 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 contrast to our recent approach [14], 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.

Finally, static cache analysis [25] is an active and challenging research topic. Compared to static cache analysis, our approach has flavors of testing and debugging. As a result, we believe that our work can be leveraged to drive security-related optimizations.

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.

VII. Threats to validity

Our framework does not exhibit false positives, therefore the computed cache side-channel leak indeed appears in real execution. However, 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. There exists other cache attacks [8] 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 and using performance counters provided by the operating system.

VIII. Conclusion and Future Work

In this paper, we have introduced the test generation problem to validate cache side-channel leakage of an arbitrary software. 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. In particular, we 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 also 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 covert channels. We believe this is possible, as our test generator explores timing behaviour of a program and any significant deviation from such timing can be detected efficiently at runtime.

References

[1] Advanced Encryption Standard Implementation. https://github.com/B-Con/crypto-algorithms.

[2] GDK keyboard handling library. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html.

[3] gdk_keyval_from_name. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html#gdk-keyval-from-name.

[4] gdk_keyval_to_unicode. https://developer.gnome.org/gdk3/stable/gdk3-Keyboard-Handling.html#gdk-keyval-to-unicode.

[5] ioctl. http://man7.org/linux/man-pages/man2/ioctl.2.html.

[6] OpenSSL Library. https://github.com/openssl/openssl/tree/master/crypto.

[7] perf_event_open. http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[8] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES. In *Information and Communications Security*. Springer, 2006.

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX*, pages 53–70, 2016.

[10] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.

[11] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014.

[12] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[13] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *ASIACRYPT*. Springer, 2009.

[14] Sudipta Chattopadhyay. Directed automated memory performance testing. In *TACAS*, 2017.

[15] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leak in cache attacks through symbolic execution. *CoRR*, abs/1611.04426, 2016.

[16] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: a tool for the static analysis of cache side channels. *TISSEC*, 18(1):4, 2015.

[18] 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. https://eprint.iacr.org/2016/613.pdf/.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[20] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*. IEEE, 2011.

[21] Aki Helin. Radamsa. https://github.com/aoh/radamsa.

[22] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *CAV*. Springer, 2012.

[23] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *CSF*, 2016.

[24] 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.

[25] 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.

[26] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[27] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, 2007.

[28] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl.