TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/18 pp47–58
DOI: 10.26599/TST.2021.9010078
Volume 28, Number 1, February 2023

SPIDER: Speeding up Side-Channel Vulnerability Detection via Test Suite Reduction

Fei Yan, Rushan Wu, Liqiang Zhang*, and Yue Cao

Abstract: Side-channel attacks allow adversaries to infer sensitive information, such as cryptographic keys or private user data, by monitoring unintentional information leaks of running programs. Prior side-channel detection methods can identify numerous potential vulnerabilities in cryptographic implementations with a small amount of execution traces due to the high diffusion of secret inputs in crypto primitives. However, because non-cryptographic programs cover different paths under various sensitive inputs, extending existing tools for identifying information leaks to non-cryptographic applications suffers from either insufficient path coverage or redundant testing. To address these limitations, we propose a new dynamic analysis framework named SPIDER that uses fuzzing, execution profiling, and clustering for a high path coverage and test suite reduction, and then speeds up the dynamic analysis of side-channel vulnerability detection in non-cryptographic programs. We analyze eight non-cryptographic programs and ten cryptographic algorithms under SPIDER in a fully automated way, and our results confirm the effectiveness of test suite reduction and the vulnerability detection accuracy of the whole framework.

Key words: side-channel detection; test suite reduction; dynamic analysis

1 Introduction

Side-channel attacks infer sensitive information, such as cryptographic keys or private user data, by monitoring non-functional information during program execution. A variety of side-channel attacks target cryptographic implementations[1–3] owing to the valuable information they contain. Particularly, software-based micro-architectural side-channel attacks (e.g., cache attacks[4–6], Dynamic Random Access Memory (DRAM) attacks[7], and controlled-channel attacks[8]) have received extensive attention from researchers and developers because they can be launched from the software without the need of physical access. Currently, a lot of these software-based attacks exploit side-channel information leakage to recover secret keys of cryptographic primitives[9, 10].

To address this issue, leakage detection tools that allow developers to identify side-channel vulnerabilities have been proposed. Such tools can be classified into static and dynamic approaches. Most static analysis tools target cache attacks and use abstract interpretation[11–13]. Although these tools try to accurately quantify information leakages, they provide an over-approximation of leakage[12]. By contrast, dynamic approaches[14, 15] focus on concrete program executions to reduce false positives. Various dynamic approaches in detecting side-channel leakages have been proposed[16, 17]. DATA[16] collects and cross-compares execution traces of various inputs of target cryptographic algorithms. Abacus[18] utilizes one execution trace and applies symbolic execution to generate constraints, and then estimates the number of leaked bits for each leakage site. The two techniques perform well on cryptographic implementations with just a subset of inputs or even one input. This is because, in the case of cryptographic algorithms, crypto primitives heavily diffuse secret

• Fei Yan, Rushan Wu, Liqiang Zhang, and Yue Cao are with Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: yanfei@whu.edu.cn; rushanwu@whu.edu.cn; Zhanglq@whu.edu.cn; yue.cao@whu.edu.cn.

∗ To whom correspondence should be addressed.

Manuscript received: 2021-10-01; accepted: 2021-10-13

inputs during processing, so testing a subset of inputs is enough to encounter information leaks with a high probability[16]. However, previous techniques cannot be easily applied to non-cryptographic applications, which have multiple program paths, because testing just a subset of inputs may not reach a perfect path coverage and may lose detection accuracy. Although a few tools target side-channel vulnerabilities in non-cryptographic applications[19], they mainly focus on timing side-channels, which are quite different from our target of address-based side-channels.

Based on this observation, we aim to explore principles and techniques that automatically identify side-channel vulnerabilities in general programs that allow an adversary who can observe execution traces of the control flow of a program to infer sensitive information. To achieve our goal, we designed tool that can tackle the following challenges: The first challenge is how to generate enough valid sensitive inputs of target programs to reach a high path coverage. Current static approaches, such as symbolic execution, are easily trapped into path explosion and require source codes sometimes. The second challenge is how to reduce the size of the input set while maintaining the vulnerability detection capability. Several redundant test cases exist, and testing all inputs and generating all execution traces under these inputs are time consuming and cause little enhancement on the results of the side-channel vulnerability detection. The third challenge is how to refine detection objects that possibly have side-channel vulnerabilities. Previous tools hardly focus on the size of traces generated, leading to a waste of memory space for the storage of unimportant data. In this paper, we address these challenges and identify side-channel vulnerabilities in non-cryptographic programs. More specifically, we built a tool named SPIDER by leveraging fuzzing techniques to generate input sets for an arbitrary program, integrating execution profiling of basic blocks and clustering techniques to realize test suite reduction, extracting target functions according to cluster results finely, and identifying side-channel vulnerabilities using trace diffing. We tested SPIDER with eight non-cryptographic programs and ten cryptographic algorithms and successfully discovered numerous side-channel leakage points for these programs with reduced consumption of time and memory space. To summarize, the contributions of this study are as follows:

- We target non-cryptographic programs containing sensitive information and introduce fuzzing, execution profiling, and clustering for speeding up the dynamic analysis of side-channel vulnerabilities.
- We propose a novel approach reducing the test suite while maintaining the accuracy of side-channel vulnerability detection. We model the execution profiles as program features and use the hierarchical clustering method for input classification.
- We perform a comprehensive analysis of sensitive side-channel vulnerabilities for real-world programs more than only cryptographic algorithms. The detailed report on leakages including corresponding trigger inputs, source locations, and call stacks, can help developers locate and fix vulnerabilities.

2 Background

2.1 Address-based side channels

Side channels leak sensitive information through nonfunctional behaviors caused by shared hardware resources in modern computer systems, such as the CPU cache[4, 20, 21], page table[8], and DRAM[7]. If a program jumps to different target addresses in branches or accesses different memory addresses when it processes different sensitive inputs, then it may be vulnerable to address-based side-channel attacks. As shown in Fig. 1, function dA_get_corrupted_input( ) has a "for" loop that enumerates every element of array x and calls function binomial( ) if the element is not 0. The program executes different patterns of control transfers when it processes different sensitive inputs, indicating that it may be vulnerable to side channel attacks.

Different side channels can be exploited to retrieve information at various granularities. Generally, cache side channels can extract secret information at the granularity of the cache line, whereas controlled side channels[8, 22] can observe sensitive information at the granularity of the memory page.

2.2 Test suite reduction

Test suite reduction is one of the most important techniques in the software testing domain. The goal

```
void dA_get_corrupted_input (dA* this, int* x, int* tilde_x, double p)
{
    int i;
      for (i=0; i<this–> n_visible; i++){
     if (x[i] == 0) {
         tilde_x[i] = 0;
       }  else {
         tilde_x[i] = binomial(1, p);
        }
     }
}
```

Fig. 1 Sensitive input-dependent control-flow transfers.

of test suite reduction is to find a reduced test suite by permanently eliminating redundant test cases according to certain criteria while keeping their fault detection capability similar to the original test suite. Several reduction techniques have been proposed, such as requirement based, coverage based, genetic algorithm, clustering, and fuzzy logic. We focus on the clustering technique used in test suite reduction. In Ref. [23], the authors used the data mining approach of the clustering technique in software testing to reduce the test suite. In Ref. [24], previous research was enhanced, and the number of function execution and the sequence of functions were taken into consideration. Clustering techniques select test cases using coverage- and distribution-based techniques. They produce small representative sets of test cases, saving time and cost of software testing.

3 Overview

3.1 Threat model

We consider a powerful adversary who attempts to retrieve secret information from side-channel observations. We assume that an adversary shares the same hardware platform with the target and side-channel observations collected by the adversary are noise free. The target program is deterministic, and the adversary has access to the binary executable of the target program. The adversary has no direct access to the target's memory or cache, but it can probe its memory or cache at each program point. Moreover, the adversary does not only learn the sequence of the addresses of instructions but also the address of the operands that are accessed by each instruction. Such a threat model can cover most memory-based and cache-based side-channel attacks.

Not all of the side channels are of our focus in this study. In particular, we focus on side-channel vulnerabilities due to secret-dependent control flows. Side channels caused by different data access patterns are currently out of scope.

3.2 Methodology

The key objective of this work is to automatically identify side-channel vulnerabilities in non-cryptographic programs processing sensitive information. Prior studies in side-channel detection mainly consider cryptographic implementations, and because of the high diffusion of secret keys in the program, a few execution traces have already enabled them to reach their goal. On the one hand, unlike cryptographic programs, perfect detection accuracy cannot be easily achieved by just testing a subset of inputs in non-cryptographic but sensitive programs. On the other hand, testing the whole input space containing many redundant test cases is impossible and not much use for detection accuracy improvement. For example, DATA[16] only takes three inputs in the case of testing symmetric algorithms, whereas in practice, we need to use fuzzing to generate 366 inputs of the Hunspell program of our benchmark to reach a high path coverage. In addition, trace generation and comparison without test suite reduction would take up more than 33 hours in the Hunspell program, which is 3x slower than our tool. Therefore, we aim to speed up the dynamic analysis of side-channel vulnerabilities while maintaining vulnerability detection capabilities through test suite reduction.

As depicted in Fig. 2, we first fed the target program into the fuzzing engine for high path coverage. Next, we collected execution profiles on the top of Dynamic Binary Instrumentation (DBI), modeled the execution profile of basic blocks as feature vectors, and utilized the clustering technique for the test suite reduction. Based on the clustering results, we extracted sensitive functions for subsequent monitoring. Finally, we conducted trace generation and leakage detection under the reduced input set and target functions.

4 Design

In this section, we illustrate how to detect side-channel vulnerabilities in non-cryptographic programs through execution profiling and the clustering technique. We use the code snippet shown in Fig. 3 as a running example. This program takes three types of sensitive inputs: $a < 5$, $5 \leqslant a < 8$, or $a \geqslant 8$. Accordingly, the program outputs

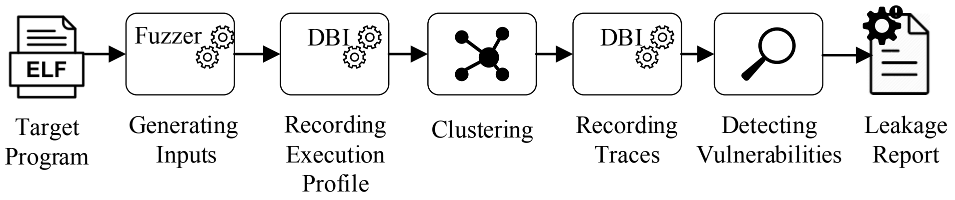


Fig. 2 Overview of SPIDER.

```
int func (int a)
{
   int r;
   if (a < 5), r = 0;
   else if (a < 8), r = 1;
   else  r = 2;
   return r;
}
int main (int argc, char** argv){
    int a, res;
    a = atoi (argv[1]);
     res = func (a);
    printf (“Result is %d\n”, res);
   return 0;
}
```

Fig. 3 Code snippet of our running example.

three types of results: 0, 1, or 2.

4.1 Input generation

Because our approach uses dynamic analysis, it is important to generate various concrete inputs that cover as many input spaces as possible for high path coverage. Sometimes, the exact type of input cannot be easily determined. Such a problem is a concerning issue in software testing, and many existing vulnerability tools contribute to solving such a challenge. The American Fuzzy Lop (AFL)[25] tool generates various inputs through diverse mutation strategies and gray-box evolutionary search algorithms for path coverage enhancement. We used the AFL tool for our purpose in the first step when designing our approach. We point out that a user-constructed input set with high path coverage is also acceptable.

In particular, we used the AFL to execute target programs multiple times before starting our analysis. The input generated by the AFL is called I_{fuzz}. When fuzzing cannot further explore the program path or reach the predefined timeout, we terminated the input generation stage. During this execution stage, we collected as many program inputs as possible, which form a sufficiently large input set. In our simple example, I_{fuzz} includes the entire set of integers.

4.2 Test suite reduction

There are many redundant inputs with no use for result enhancement in the last phase, and we eliminated such inputs to save the time and cost of testing. This goal coincides with the idea of the test suite reduction in software testing. We first define our problem below:

Given a test suite named I_{fuzz}, and a set of test requirements, R_1, $R_2, \ldots, R_n$, that must be satisfied to provide the desired test coverage of the program, can we find a reduced suite called $I_{reduced}$ containing minimal test cases from I_{fuzz} that satisfies all test requirements R_i at least once?

The problem can be abstracted as a classification problem or an equivalence class division problem. We intend to apply machine learning techniques for classification. We first need to construct vectors representing program features under different inputs and classify them reasonably. A program is composed of multiple basic blocks, namely, the single-entrance, single-exit sequences of instructions. As shown in Fig. 4, in our example program, the basic block information and execution profiles are quite different under different types of inputs. When two inputs belong to different groups, they jump to different branch target addresses and therefore execute different basic blocks. This condition indicates that the program may be vulnerable to side-channel attacks. Hence, we consider the execution sequence of basic blocks as the feature of each input in I_{fuzz} and collect execution profiles r_i on the top of the DBI framework, i.e., Intel Pin[26]. We executed the target program on each input in I_{fuzz} and logged the start address and end address of basic blocks, execution sequence of basic blocks, and function calls and returns. The execution profiles of our example program are shown in Column 2 of Table 1. As Tabel 1 shows, inputs of the same type have the same execution profiles. We just display six inputs of the example program for a clear explanation. Essentially, during our experience, there are a lot of inputs, not just the six inputs we show in Table 1. For example, in our benchmark, there are 185 inputs in the case of the dA program of just 81 KB size after fuzzing.

To map observations to vectors with the same dimensions, we first traversed all execution profiles and

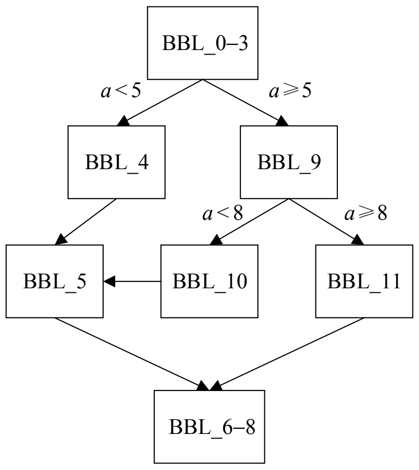


Fig. 4 Execution profile of three types of input.

Table 1 Test suite reduction of example program.

Input under test	Basic block sequence	Vector representation	Clustering result
$a = 0$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_4, BBL_5, BBL_6, BBL_7, BBL_8}	{1,1,1,1,1,1,1,1,1,0,0,0}	0
$a = 1$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_4, BBL_5, BBL_6, BBL_7, BBL_8}	{1,1,1,1,1,1,1,1,1,0,0,0}	0
$a = 6$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_9, BBL_10, BBL_5, BBL_6, BBL_7, BBL_8}	{1,1,1,1,0,1,1,1,1,1,1,0}	1
$a = 7$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_9, BBL_10, BBL_5, BBL_6, BBL_7, BBL_8}	{1,1,1,1,0,1,1,1,1,1,1,0}	1
$a = 9$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_9, BBL_11, BBL_6, BBL_7, BBL_8}	{1,1,1,1,0,0,1,1,1,1,0,1}	2
$a = 10$	{BBL_0, BBL_1, BBL_2, BBL_3, BBL_9, BBL_11, BBL_6, BBL_7, BBL_8}	{1,1,1,1,0,0,1,1,1,1,0,1}	2

constructed a complete list of basic blocks. Then, we represented observations collected under each input as a vector v_i, where each element indicates the number of times the corresponding basic block executed. The vector representation of our example program is shown in Column 3 of Table 1.

After we successfully constructed vectors representing the execution profile under every input, we correctly classified them using a clustering algorithm. Because we could not decide how many clusters would be generated, we utilized the hierarchical clustering algorithm[27]. We considered the L1 norm distance with the tolerance $\epsilon = 1$.

We designed an algorithm (Algorithm 1) to illustrate the above process. Specifically, we first obtained all executed basic blocks by traversing all execution profile r_i and constructed a complete basic block set s. We use the function $insert()$ to insert the new record vector of basic blocks into the set s and the function $unique()$ to remove duplicate elements from the set. Then, we traversed r_i again to construct v_i for every input by counting the number of executions of basic blocks. We use the function $find()$ to determine the index of record vector in the set s. Finally, we passed the vectors to the clustering algorithm and derived the label of every input. The clustering result of our example program is shown in Column 4 of Table 1. Inputs in each class have the same execution profiles, so we formed the minimal test suite $I_{reduced}$ by gathering one of the inputs in each class.

Algorithm 1 Clustering based on execution profile feature

```
Input: the record vector of basic blocks executed under every
    input in $|I_{fuzz}|$, $r_0, r_1, \ldots, r_{n-1}$; cluster bound $K$;
    distanced $d$; and tolerance $\epsilon$
Output: the label of every input in $|I_{fuzz}|$, $l_0, l_1, \ldots, l_{n-1}$
1: $s = \{0\}, r = \{0\}, i = 0, j = 0$;
2: while $i < n$ do
3:     $r = r_i$;
4:     $s = s.insert(s.end(), r.begin(), r.end())$;
5:     $i++$;
6: end while
7: $s = unique(s)$;
8: $m = |s|$;
9: $v_0[m], v_1[m], \ldots, v_{n-1}[m] = \{0\}$;
10: while $j < n$ do
11:     $k = 0$;
12:     while $k < |r_j|$ do
13:         $idx = find(s, r_j[k])$;
14:         $v_j[idx]++$;
15:         $k++$;
16:     end while
17:     $j++$;
18: end while
19: $V = (v_0, v_1, \ldots, v_{n-1})$;
20: $(l_0, l_1, \ldots, l_{n-1}) = clustering(V, K, d, \epsilon)$;
21: return $l_0, l_1, \ldots, l_{n-1}$.
```

4.3 Target function extraction

In this section, we describe how unnecessary information can be filtered during trace generation and also describe how the size of traces with each given input in $I_{reduced}$ which can be reduced.

Based on the clustering results in Section 4.2, we only focused on specific sensitive functions that explain different cluster results. We cross-compared $v_i s$ between clusters and found basic blocks that differ in $v_i s$. The target program executes different basic blocks when it jumps to different branch target addresses under different types of inputs. Hence, the functions containing these basic blocks are susceptible to side-channel vulnerabilities.

Because we logged the information of basic blocks and function calls during the feature vector construction presented in Section 4.2, we can easily track the functions that the basic blocks belong to and construct a function list of possible side-channel vulnerabilities. Given that not only one basic block points to a function, we only recorded the function name once. Moreover, in the case of function nesting, we traced back to the initial call function, providing accurate information on the control flow transfers of the target program. In our example, the program executes different basic blocks under the input of labels 0 and 1. It executes BBL_4 under the input of label 0, while it executes BBL_9 and BBL_10 under the input of label 1. Based on the difference of basic blocks executed, we can narrow our detection range to a function named func( ) in Fig. 3. The algorithm is illustrated in Algorithm 2. The set C is a set that contains the call stack of the target sensitive functions. The variable cs temporarily stores the call stack. We obtain a list of target sensitive functions by simulating the process of function calls.

Algorithm 2 Extracting list of target sensitive functions

```
Input: the execution trace under one input in $|I_{reduced}|$, $t$;
    the address of basic blocks that may exist vulnerabilities, $b_0, b_1, \ldots, b_{n-1}$
Output: the list of sensitive functions $fl$
1: $C = \varnothing, cs = \{0\}, i = 0, fl = 0$;
2: while $i < |t|$ do
3:   if $isCall(t[i])$ then
4:     $cs = cs.push\_back(t[i].addr)$;
5:   else if $isRet(t[i])$ then
6:     $cs = cs.pop\_back()$;
7:   else
8:     $j = 0$;
9:     while $j < n$ do
10:      if $t[i].addr == b_j \wedge find(cs, C) == false$ then
11:        $C = C \cup cs$;
12:      end if
13:      $j++$;
14:    end while
15:  end if
16:  $i++$;
17: end while
18: $k = 0$;
19: while $k < |C|$ do
20:   $fl[k] = C[k].back()$;
21:   $k++$;
22: end while
23: $fl = unique(fl)$;
24: return $fl$.
```

4.4 Trace generation

We executed the program under every input in $I_{reduced}$ on Intel Pin and stored the necessary information in a trace file. We recorded the start address and the end address of basic blocks, instruction and target address of branches, and address of function calls and returns for control flow transfers construction. Then, we collected traces of two inputs with the same label to verify whether inputs also influence the number of loops.

To execute the program in a noise-free environment, we first disabled Address Space Layout Randomization (ASLR) and kept public inputs to the program fixed. We then passed inputs in $I_{reduced}$ generated in Section 4.2 and target function name extracted in Section 4.3 to the target program, and executed the program in instrumented mode, recording data that we need for later analysis. We ignored the deviation caused by the recording time because it is too small compared to the trace analysis.

We considered the func( ) function in Fig. 3, assuming that line numbers are equal to the code addresses. The execution with two different inputs, $a_0 = 0$ and $a_1 = 6$, yields two traces t_0 and t_1,

$t_0 = [(1,4), \mathbf{(4,4)}, \mathbf{(4,5)}, (7,8)]$,

$t_1 = [(1,4), \mathbf{(4,5)}, \mathbf{(5,5)}, \mathbf{(5,6)}, (7,8)]$.

There are two differences in the traces; both are marked bold. As shown in Fig. 3, the differences occur as if in Line 4 branches to Line 4 or 5, depending on the input a, and causes assignment operations in Lines 4 and 5 to be performed either on the variable r.

In Fig. 3, a control flow leak is characterized by its branch point, where the control flow diverges, and its merge point, where branches coalesce again. In this example, the branch point is at Line 4 and the merge point at Line 7, when the function returns.

4.5 Leakage detection

Clearly, traces do not always have the same length. We need to align traces for more precision and determine control flow differences. We cross-compared every element in every two traces. Basic blocks are the same when their start address and end address are identical, such as $(1,4)$ in t_0 and t_1 in Section 4.4. In this case, we moved to the next element of two traces. A branch point was located when the target address differs, and the correct merge point was determined by the first identical address in the intersection of the following address sequences between two traces. When a branch point appeared, we continuously moved to the next element in both traces until we successfully found its merge point. The branch point in the sample program is $(4,4)$ or $(4,5)$ in t_0 and t_1, and the merge point is $(7,8)$ in t_0 and t_1. We constantly re-aligned traces in the same way until the end of traces.

There is a problem when applying the trace alignment algorithm of DATA[16] for solving context sensitivity. They determine function returns using $INS\_IsRet()$ API provided by Intel Pin, which also contains the interrupt return (iret). The calling depth drops below zero when the interrupt return happens, and then the process of finding the merge point will trap into a dead loop. Due to the differences in the characteristics of programs, this problem is not apparent in the case of cryptographic programs, but it will have a significant impact on non-cryptographic programs. We solved this problem by pushing the loop forward when the calling depth of two traces dropped below zero.

All discovered branch points are considered possible side-channel vulnerabilities. Apart from each pair of branch point and merge point, function calls in the context, source code location, and input pair, which can trigger the vulnerabilities, are also reported. Consequently, developers can fix their program selectively according to the report.

5 Implementation

SPIDER consists of 2512 lines of code in C++, 352 lines of code in Python, and 270 lines of code in a shell script. We implemented our input generation through the AFL tool and greatly reduced the size of the test suite with the help of the algorithm we designed and the clustering algorithm. We implemented trace collection on the top of the Intel Pin framework for analyzing x86 binaries. To reduce their size, we utilized cluster results to extract target functions that may have side-channel vulnerabilities and only monitored these functions for detecting control flow leakages. We implemented leakage detection, which condenses all findings into leakage reports, including pairs of inputs that can reproduce the leakage, source location of leakage, and call stacks. We wrote a shell script to connect every step.

6 Evaluation

In this section, we present our evaluation results. We built the source program into a 64-bit x86 Linux executable with GCC 7.5. All our evaluations were performed in Ubuntu 18.04, running on the top of Intel i5-1135G7 CPU, with 16 GB RAM.

6.1 Experimental setup

Benchmark selection. The core idea of SPIDER is to speed up the dynamic analysis of side channel vulnerabilities in non-cryptographic programs without loss of accuracy. To verify the effectiveness of our approach, we used the same benchmark with ANABLEPS[28], which contains various non-cryptographic programs with sensitive information. In addition, we checked our approach on 10 finalists of the NIST lightweight cryptography standardization project[29] for further confirmation.

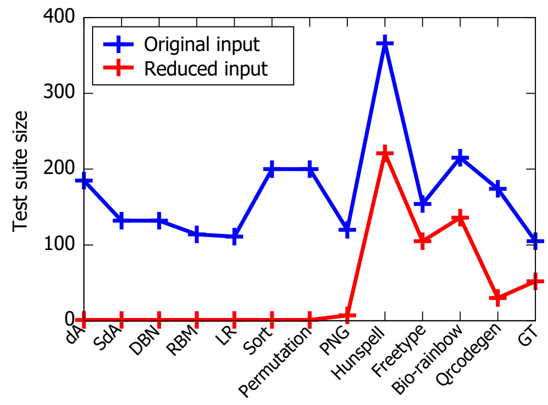
6.2 Experimental results

6.2.1 Effectiveness of the test suite reduction

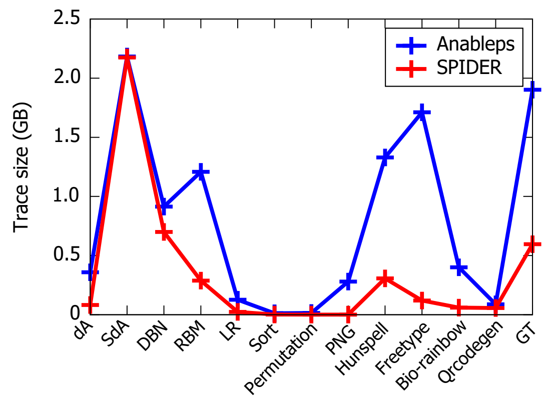
We generated I_{fuzz}, the original input set of the testing program, after running the AFL tool for 24 h. Then, we combined execution profiling and clustering techniques to reduce the test suite and obtain a small input set named $I_{reduced}$. To verify the effectiveness of the test suite reduction in our approach, we compared our work with ANABLEPS[28]. ANABLEPS generates inputs with the help of Driller[30], another fuzzing tool. The choice of fuzzing tool has little influence on the generation of original inputs, and we focus on how much our approach can influence the reduction of the test suite.

We extracted the data of the input size and trace size in ANABLEPS. As the size of the original input set is quite different between SPIDER and ANABLEPS and only the total size of all traces is given in ANABLEPS, we calculated the average trace size for precise comparison.

Specifically, we compared the test suite size and averaged the trace size of each testing program, and the results are shown in Fig. 5. As shown in Fig. 5a, SPIDER can drastically reduce the size of the input set, ranging from 31.8% to 98.8%. This finding proves the effectiveness of the test suite reduction. In addition, we compared the average trace size of the two approaches and summarized our results in Fig. 5b. Clearly, the trace size of SPIDER is smaller than that of ANABLEPS, and is reduced by 3.6% to 92.9%, which proves the effectiveness of the test suite reduction and target function extraction. They both help filter out several unnecessary pieces of information and hence reduce the size of the trace. Hence, our approach reduces the size of the test suite and traces successfully and



(a) Comparison of test suite size



(b) Comparison of trace size

Fig. 5 Comparison of the test suite size and trace size.

efficiently, which speeds up the trace generation and leakage detection.

6.2.2 Capabilities of side-channel vulnerability detection

In this section, we try to confirm that although the size of the input set is reduced through our design, there is no drop in the accuracy of detecting side channel vulnerabilities. We first tested our approach on NIST finalists[29], which are all lightweight cryptographic algorithms designed for constrained environments.

Because test vector generators were provided, we utilized their design to generate the original input set of 100 inputs for every algorithm and conduct our experience. As a result, except Grain128-AHEAD, all inputs of the nine finalists of the NIST lightweight cryptography standardization project were clustered into one class after the test suite reduction. Moreover, except Grain128-AHEAD, nine finalists are resistant to side channels, in line with results in two tools tested in Ref. [31]. We then presented a detailed explanation of side-channel vulnerabilities in Grain128-AHEAD found by SPIDER.

The vulnerable code snippet of Grain128-AHEAD is shown in Fig. 6. During the generation of the

```
int crypto_aead_encrypt ( )
{
   …
   for (unsigned long long i = 0; i < mlen; i++) {
         …
         for (int j = 0; j < 16; j++) {
             …
             if ( j % 2 == 0) {…}
             else {
                     if (data.message[ac_cnt++] == 1)
                             {accumulate(&grain);  }
                     …
   }}}
   …
}
```

Fig. 6 Vulnerable code snippet of Grain128-AHEAD.

keystream for the message (i.e., plaintext), the function accumulate( ) is called when the bit of "message" is equal to 1. Hence, if an adversary can observe the execution profile of accumulate( ), then it could recover every bit of "message". The result of accumulate( ) is later used for appending MAC to a ciphertext.

We then tested our approach on ANABLEPS's benchmark and presented the results in Table 2. Because the detection level of ANABLEPS differs from ours, we confirm the details of the leakages reported in our approach with a manual check. The results in Table 2 show the possibility of side-channel vulnerabilities, and the exploitability of vulnerabilities remains for future works.

6.2.3 Performance overhead

We also measured the performance of our approach and reported the execution time for each of the key components of our approach in Table 3. Moreover, we configured the AFL tool to run 24 h for all of the benchmarks in the input generation phase for high path coverage. As shown in Column 4 of Table 3, the test suite reduction phase takes little time and plays a great

Table 2 Evaluation results.

Benchmark program	Functionality under test	Number of leaks
Deep learning	dA	6
	SdA	3
	DBN	3
	RBM	2
	LR	1
gsl	Sort	2
	Permutation	1
Hunspell	Spell checking	148
PNG	Image render	2
Freetype	Character render	463
Bio-rainbow	Bioinfo clustering	94
Qrcodegen	Generate QR Code	165
Genometools	bed to gff3 convertion	265

Table 3 Performance overhead.

Benchmark program	Functionality under test	Input generation (h)	Test suite reduction (min)	Trace gneration (min)	Leakage detection (min)
Deep learning	dA	24	1.35	0.07	0.03
	SdA	24	18.78	1.17	3.85
	DBN	24	1.02	0.15	0.25
	RBM	24	0.28	0.08	0.10
	LR	24	0.13	0.03	0.02
gsl	Sort	24	2.90	0.05	0.02
	Permutation	24	3.03	0.05	0.02
Hunspell	Spell checking	24	16.70	167.43	650.77
PNG	Image render	24	0.42	1.10	0.05
Freetype	Character render	24	2.05	225.15	295.30
Bio-rainbow	Bioinfo clustering	24	3.33	167.18	185.97
Qrcodegen	Generate QR Code	24	3.07	3.02	15.53
Genometools	bed to gff3 convertion	24	5.60	656.67	306.72

role in saving the cost and time of the later analysis.

7 Related Work

Currently, approaches in detecting side-channel vulnerabilities consist of static and dynamic approaches.

(1) Static approach

CacheAudit[12] uses abstract domains to compute an over-approximation of cache side channel information leakage upper bound. While a zero leakage bound reveals the absence of address-based side channels, a non-zero leakage bound could introduce false positives due to abstractions made on the data of the program. CacheS[13] improves CacheAudit with new abstract domains that only track secret-related code. CaSym[11] introduces a static cache-aware symbolic reasoning technique to cover multiple paths in a target program. All approaches mentioned above only work on small code snippets, making it difficult to be widely applied in larger applications, such as the benchmark programs used in our work.

(2) Dynamic approach

Diffuzz[19] modifies the fuzzing engine to find the maximum timing difference in non-cryptographic applications during program execution, but it only detects timing side channels instead of address-based side channels we focus on. CacheD[32] combines dynamic trace recording with a static analysis for avoiding false positives. However, it only tracks one execution trace, missing leakage in other execution paths. Moreover, CacheD cannot detect secret-dependent control flows. Stacco[14], MicroWalk[17], and DATA[16] detect side-channel vulnerabilities in a similar way, but their focus of detection is different. Stacco manually generates various inputs to the SSL libraries and uses Intel Pin tools to detect vulnerabilities in SSL/TSL implementations. MicroWalk introduces mutual information between sensitive inputs and execution traces for side-channel leakage quantification. DATA detects address-based side-channel vulnerabilities by comparing different execution traces under various inputs. ANABLEPS[28] uses Intel PT to generate execution traces with huge sizes for detecting side-channel vulnerabilities in enclave binaries. Abacus[18] utilizes symbolic execution and Monte Carlo sampling to estimate the number of leaked bits for each leakage site, but it only relies on one trace for modeling constraints. They both set up their own vulnerability judging rules, which are one of the necessary stages in other vulnerability detection frameworks[33, 34]. Obviously, current dynamic approaches mainly focus on address-based side channels in cryptography algorithms, and a few execution traces are enough for vulnerability detection. Existing dynamic approaches do not extend the side-channel analysis of crypto libraries into non-crypto software well yet.

Hardware and software side-channel mitigations have been proposed. Hardware countermeasures, including partitioning hardware resources[35], randomizing cache access[36–38], and modifying micro-architectural components[39], require changes to complex processors and are complex to adopt. On the contrary, software approaches are usually easy to implement, and they modify key-dependent control flow at the source code level[40, 41], at the compiler level[42], and at runtime[43]. Our side-channel detection method can locate possible vulnerabilities and help subsequent side-channel mitigation.

8 Conclusion

In conclusion, we designed and implemented a software tool for automatically detecting side-channel vulnerabilities in non-cryptographic programs with sensitive information. Our tool is the first side-channel vulnerability analysis tool that introduces fuzzing, execution profiling, and clustering technique for test suite reduction. With our tool, we have discovered numerous side-channel leaks in our test programs. Our tool can be used by software developers to check for side-channel vulnerabilities in the program they write.

There are still some potentially promising directions for future work. The current design only considers side-channel vulnerabilities due to sensitive input-dependent control flows. Leakages due to sensitive input-dependent data accesses are currently out of scope. One of the future works is to extend SPIDER in the handling of these vulnerabilities. Although the combination of execution profiling and clustering performs well in our tool, there are still some kinds of techniques for test suite reduction applied in software engineering. Moreover, comparing the impact of various approaches for test suite reduction on our tool and discovering better algorithms remains for our future work. Another direction is to extend the fuzzer with clustering techniques that can directly reduce test cases during the input generation.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (Nos. 61272452 and 61872430), the National Key Basic Research and Development (973) Program of China (No. 2014CB340601), the Key R&D Program of Hubei Province (No. 2020BAA003), and the Prospective Applied Research Program of Suzhou City (No. SYG201845).

References

[1] D. J. Bernstein, Cache-timing attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[2] P. C. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems, in *Proc. of the 16th Annu. Int. Cryptology Conf.*, Santa Barbara, CA, USA, 1996, pp. 104–113.

[3] E. Tromer, D. A. Osvik, and A. Shamir, Efficient cache attacks on AES, and countermeasures, *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, 2010.

[4] Y. Yarom and K. Falkner, FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack, in *Proc. 23rd USENIX Conf. Security Symp.*, San Diego, CA, USA, 2014, pp. 719–732.

[5] D. Gullasch, E. Bangerter, and S. Krenn, Cache games–bringing access-based cache attacks on AES to practice, *in Proc. of 2011 IEEE Symp. Security and Privacy*, Oakland, CA, USA, 2011, pp. 490–505.

[6] D. Gruss, R. Spreitzer, and S. Mangard, Cache template attacks: Automating attacks on inclusive last-level caches, in *Proc. 24th USENIX Conf. Security Symp.*, Washington, DC, USA, 2015, pp. 897–912.

[7] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, DRAMA: Exploiting DRAM addressing for cross–CPU attacks, in *Proc. 25th USENIX Conf. Security Symp.*, Austin, TX, USA, 2016, pp. 565–581.

[8] Y. Z. Xu, W. D. Cui, and M. Peinado, Controlled-channel attacks: Deterministic side channels for untrusted operating systems, in *Proc. of 2015 IEEE Symp. Security and Privacy*, San Jose, CA, USA, 2015, pp. 640–656.

[9] G. Irazoqui, T. Eisenbarth, and B. Sunar, S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES, in *Proc. of 2015 IEEE Symp. Security and Privacy*, San Jose, CA, USA, 2015, pp. 591–604.

[10] C. P. García, B. B. Brumley, and Y. Yarom, Make sure DSA signing exponentiations really are constant-time, in *Proc. 2016 ACM SIGSAC Conf. Computer and Communications Security*, Vienna, Austria, 2016, pp. 1639–1650.

[11] R. Brotzman, S. Liu, D. F. Zhang, G. Tan, and M. Kandemir, CaSym: Cache aware symbolic execution for side channel detection and mitigation, in *Proc. of 2019 IEEE Symp. Security and Privacy*, San Francisco, CA, USA, 2019, pp. 505–521.

[12] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, Cacheaudit: A tool for the static analysis of cache side channels, *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, p. 4, 2015.

[13] S. Wang, Y. Y. Bao, X. Liu, P. Wang, D. F. Zhang, and D. H. Wu, Identifying cache-based side channels through secret-augmented abstract interpretation, in *Proc. 28th USENIX Security Symp.*, Santa Clara, CA, USA, 2019, pp. 657–674.

[14] Y. Xiao, M. Y. Li, S. C. Chen, and Y. Q. Zhang, STACCO: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves, in *Proc. 2017 ACM SIGSAC Conf. Computer and Communications Security*, Dallas, TX, USA, 2017, pp. 859–874.

[15] A. Zankl, J. Heyszl, and G. Sigl, Automated detection of instruction cache leaks in modular exponentiation software, in *Proc. of the 15th Int. Conf. Smart Card Research and Advanced Applications*, Cannes, France, 2016, pp. 228–244.

[16] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, DATA–differential address trace analysis: Finding address-based side-channels in binaries, in *Proc. 27th USENIX Conf. Security Symp.*, Baltimore, MD, USA, 2018, pp. 603–620.

[17] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, MicroWalk: A framework for finding side channels in binaries, in *Proc. 34th Annu. Computer Security Applications Conf.*, San Juan, PR, USA, 2018, pp. 161–173.

[18] Q. K. Bao, Z. H. Wang, X. T. Li, J. R. Larus, and D. H. Wu,

Abacus: Precise side-channel analysis, in *Proc. of 2021 IEEE/ACM 43rd Int. Conf. Software Engineering* (*ICSE*), Madrid, Spain, 2021, pp. 797–809.

[19] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, DiffFuzz: Differential fuzzing for side-channel analysis, in *Proc. of 2019 IEEE/ACM 41st Int. Conf. Software Engineering* (*ICSE*), Montreal, Canada, 2019, pp. 176–187.

[20] D. A. Osvik, A. Shamir, and E. Tromer, Cache attacks and countermeasures: The case of AES, in *Proc. of Cryptographers' Track at the RSA Conf.*, San Jose, CA, USA, 2006, pp. 1–20.

[21] C. Percival, Cache missing for fun and profit, https://papers.freebsd.org/2005/cperciva-cache_missing/, 2005.

[22] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, Preventing page faults from telling your secrets, in *Proc. 11th ACM on Asia Conf. Computer and Communications Security*, Xi'an, China, 2016, pp. 317–328.

[23] B. Subashini and D. JeyaMala, Reduction of test cases using clustering technique, *Int. J. Innov. Res. Sci. Eng. Technol.*, vol. 3, no. 3, pp. 1992–1996, 2014.

[24] R. C. Wang, B. B. Qu, and Y. S. Lu, Empirical study of the effects of different profiles on regression test case reduction, *IET Softw.*, vol. 9, no. 2, pp. 29–38, 2015.

[25] American fuzzy lop, https://lcamtuf.coredump.cx/afl, 2013.

[26] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[27] S. C. Johnson, Hierarchical clustering schemes, *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[28] W. B. Wang, Y. Q. Zhang, and Z. Q. Lin, Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries, in *Proc. of the 22nd Int. Symp. Research in Attacks, Intrusions and Defenses* (*RAID 2019*), Beijing, China, 2019, pp. 443–457.

[29] NIST, Lightweight cryptography, https://csrc.nist.gov/projects/lightweight-cryptography, 2021.

[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Y. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution, in *Proc. of the 23rd Annu. Network and Distributed System Security Symp.*, San Diego, CA, USA, 2016, pp. 1–16.

[31] A. B. Hansen, E. H. Nielsen, and M. Eskildsen, Toolchain for timing leakage analysis of NIST lightweight crypto candidates, https://csrc.nist.gov/Presentations/2020/toolchain-for-timing-leakage-analysis-of-lwc, 2020.

[32] S. Wang, P. Wang, X. Liu, D. F. Zhang, and D. H. Wu, Cached: Identifying cache-based timing channels in production software, in *Proc. 26th USENIX Conf. Security Symp.*, Vancouver, Canada, 2017, pp. 235–252.

[33] J. C. Hu, J. F. Chen, L. Zhang, Y. S. Liu, Q. H. Bao, H. Ackah-Arthur, and C. Zhang, A memory-related vulnerability detection approach based on vulnerability features, *Tsinghua Science and Technology*, vol. 25, no. 5, pp. 604–613, 2020.

[34] J. W. Tang, R. X. Li, K. P. Wang, X. W. Gu, and Z. Y. Xu, A novel hybrid method to analyze security vulnerabilities in Android applications, *Tsinghua Science and Technology*, vol. 25, no. 5, pp. 589–603, 2020.

[35] D. Page, Partitioned cache architecture as a side-channel defence mechanism, http://eprint.iacr.org/2005/280, 2005. .

[36] G. Dessouky, T. Frassetto, and A. R. Sadeghi, HybCache: Hybrid side-channel-resilient caches for trusted execution environments, in *Proc. of the 29th USENIX Security Symp.*, Boston, MA, USA, 2020, pp. 451–468.

[37] Z. H. Wang and R. B. Lee, New cache designs for thwarting software cache-based side channel attacks, in *Proc. 34th Annu. Int. Symp. Computer Architecture*, New York, NY, USA, 2007, pp. 494–505.

[38] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, SCATTERCACHE: Thwarting cache attacks via cache set randomization, in *Proc. 28th USENIX Conf. Security Symp.*, Santa Clara, CA, USA, 2019, pp. 675–692.

[39] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security, in *Proc. 38th Annu. Int. Symp. Computer Architecture*, San Jose, CA, USA, 2011, pp. 189–199.

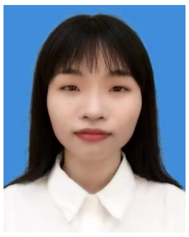
[40] R. Könighofer, A fast and cache-timing resistant implementation of the AES, in *Proc. of Cryptographers' Track at the RSA Conf.*, San Francisco, CA, USA, 2008, pp. 187–202.

[41] C. Rebeiro, D. Selvakumar, and A. S. L. Devi, Bitslice implementation of AES, in *Proc. of the 5th Int. Conf. Cryptology and Network Security*, Suzhou, China, 2006, pp. 203–212.

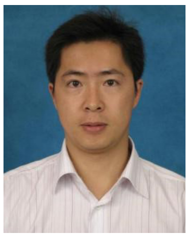
[42] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, Practical mitigations for timing-based side-channel attacks on modern x86 processors, in *Proc. of 2009 30th IEEE Symp. Security and Privacy*, Oakland, CA, USA, 2009, pp. 45–60.

[43] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, Thwarting cache side-channel attacks through dynamic software diversity, in *Proc. of 22nd Annu. Network and Distributed System Security Symp.*, San Diego, CA, USA, 2015, pp. 8–11.

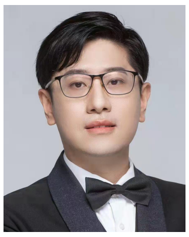
Fei Yan received the PhD degree from Wuhan University, China in 2007. He is currently an associate professor at the School of Cyber Science and Engineering, Wuhan University, China. He is a co-founder of ChinaSigTC (China special interest group on trusted cloud) and served as an associate chair of program committee of CTCIS (Chinse Trusted Computing and Information Security Conference) from 2017 to 2021. His current research interests include system security, trusted computing, side-channel security, and AI security.



Rushan Wu received the BS degree from Wuhan University, China in 2019. She is currently a master student at the School of Cyber Science and Engineering, Wuhan University, China. Her research interests include side-channel analysis and system security.



Liqiang Zhang received the PhD degree in information security from Wuhan University, Wuhan, China in 2008. He is currently an associate professor at the School of Cyber Science and Engineering, Wuhan University, China. His current research interests include trusted computing, software analysis, AI security, and system evaluation.



Yue Cao received the PhD degree from University of Surrey, Guildford, UK in 2013. Further to the PhD study, he was a research fellow at University of Surrey, and an academic faculty at Northumbria University, Lancaster University, UK, and Beihang University, China, and he is currently a professor at the School of Cyber Science and Engineering, Wuhan University, China. His research interests focus on intelligent transport systems, including E-mobility, V2X, and edge computing.