Privacy Leakage Vulnerability Detection for Privacy-Preserving Computation Services

Su Zhang
Key Lab of High-Confidence Software Technology, MoE
School of Computer Science
Peking University
Beijing, China
samsuzhang@pku.edu.cn

Ying Zhang*
Key Lab of High-Confidence Software Technology, MoE
National Engineering Research Center for Software Engineering
Peking University
Beijing, China
zhang.ying@pku.edu.cn

2022 IEEE International Conference on Web Services (ICWS) | 978-1-6654-8143-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICWS55610.2022.00043

***Abstract*—Privacy leakage is a forever critical issue for data sharing and cooperation. Therefore, many Privacy-Preserving-Computation-aimed services (PPCS) are published to provide a secure environment in which data can be processed in its encrypted or opaque state by specific programs (i.e. PPCS program). However, PPCS programs still face the risk of privacy leakage due to the intentionally or careless designed privacy leakage vulnerabilities (PLV) that may leak sensitive data in the returned result. Unfortunately, traditional PLV-detection approaches like quantitative estimation and taint analysis become inefficient for these PLVs due to the extremely large input domain and the complex data-processing logic of PPCS programs. In this paper, we propose a fuzzing-based approach named FuzzLeaks to detect PLVs. It uses coverage-oriented fuzz testing to generate test cases for checking PPCS programs and thus to carry out leakage estimation to detect PLVs. It effectively quantifies privacy leakage under the extremely large input domain via path-sensitive byte-level entropy analysis, and handles the complex data-processing logic via input mutation based on dynamic information flow analysis. We implement FuzzLeaks and validate it on the PLDA data set and LAVA-M data set. The experimental results show that FuzzLeaks outperforms traditional approaches in accuracy by 35.72% on the PLDA dataset, and the dynamic-analysis-based mutation guidance adopted by FuzzLeaks can even resulted in 50 more non-PLV bugs found on the LAVA-M data set.**

***Index Terms*—privacy leakage vulnerability, fuzz testing, privacy-preserving computation service**

I. Introduction

Privacy leakage is a forever critical issue for data sharing and cooperation, and always gains public attentions. For example, Cambridge Analytica illegally harvested 87 million Facebook profiles and used them for targeted political advertisements [1]. Such leakage will surely cause huge problems.

Therefore, Privacy-Preserving Computation (PPC) [2] has become a hot topic in recent years, and we can see that many clouds have published PPC-aimed services (PPCS) such as AWS Nitro Enclaves [3], Azure Confidential Computing [4] and AlibabaCloud DataTrust [5] to preserve data privacy during data sharing and cooperation. In general, a PPCS provides a secure environment in which data can be processed in its encrypted or opaque state by specific programs (i.e. PPCS program). However, although a PPCS program is strictly checked beforehand and is restricted to its claimed data privilege at runtime in such a secure environment, it may still expose sensitive data outside imperceptibly along with the returned result if being intentionally or carelessly designed.

```
01.  …
02.  for (int i = 0; i < peopleInfos.size(); i++) {
03.      salary_sum += stoi(peopleInfos[l][4]);
04.  }
05.  salary_avg = salary_sum / peopleInfos.size();
06.  return to_string(salary_avg);
```

Fig. 1. Example of a program snippet with privacy leakage vulnerabilities.

For example, Fig. 1 shows a snippet of a PPCS program that calculates the average salary of some certain people. We can see that the variable to iterate over *peopleInfos* is *l* instead of *i* (in line 3). As a result, the returned *salary_avg* will be the salary of a specific person instead of the average salary. This vulnerability will undoubtedly cause privacy leakage even if the program is executed with the protection of PPCS. We refer to such vulnerabilities as privacy leakage vulnerabilities (PLV), and an automated approach for PLV-detection is necessary considering that manual checking is high costly. However, unfortunately, traditional automated approaches like quantitative estimation [6]–[9] and taint analysis [10]–[13] will usually fail on the above PLV-detection task due to that for a given PPCS program that processes and analyzes data: 1) its potential input domain could be extremely large and hard to be dynamically analysed, 2) and its logic could be extremely complicated and hard to be statically analysed.

To solve the above problems, in this paper, we propose a fuzzing-based approach named FuzzLeaks to detect PLVs. It uses coverage-oriented fuzz testing to generate test cases for checking PPCS programs and thus to carry out leakage estimation to detect PLVs based on entropy analysis. In addition, to address the drawbacks of the above approaches, FuzzLeaks invents the path-sensitive byte-level entropy analysis to handle the extreme cases of large input domain, and leverages a dynamic information flow analysis to mutate fuzzing inputs to improve the testing coverage for the complex data-processing programs.

We implement FuzzLeaks and validate it on the PLDA data set [14] and LAVA-M data set [15]. The experimental results show that FuzzLeaks can detect PLVs more accurately

978-1-6654-8143-4/22/$31.00 ©2022 IEEE
DOI 10.1109/ICWS55610.2022.00043

than traditional approaches by 35.72% on the PLDA data set, and the dynamic-analysis-based mutation guidance adopted by FuzzLeaks can even resulted in 50 more non-PLV bugs found on the LAVA-M data set.

Our main contributions are:

- A novel fuzzing approach for detecting PLVs.
- A path-sensitive byte-level entropy analysis method to effectively quantify privacy leakage under the extremely large input domain.
- A mutation guidance method based on dynamic information flow analysis to improve the coverage of fuzz testing on complex data-processing programs.
- Implement the FuzzLeaks system and achieve promising results on the PLDA data set and LAVA-M data set.

The rest of the paper is organized as follows. Section 2 provides essential background. Section 3 introduces the definition of PLV. Section 4 introduces the design of FuzzLeaks. Section 5 introduces the implementation of FuzzLeaks and conducts validation. Section 6 introduces the related work. Section 7 summarizes the paper.

II. Background

A. Privacy-Preserving Computation

Privacy-Preserving Computation (PPC) [2] refers to the technologies that realize joint computation among mutually distrustful participants while preserving the privacy of the computed data.

PPC for general purposes is usually implemented based on Secure Multi-Party Computation (SMPC) [16] or Trusted Execution Environment (TEE) [17]. In SMPC-based PPC, a program will be converted into the form of a series of cryptographic protocols (e.g. Garbled Circuit, Secret Sharing, and Zero-Knowledge Proofs) which can perform equivalent calculations on the sensitive data in its encrypted state. In TEE-based PPC, a program will be executed inside a certain TEE to perform calculations on the sensitive data, which is always encrypted outside the TEE and can only be decrypted inside the TEE with its endorsement key.

In recent years, major clouds have successively released PPC-aimed services (PPCS), such as AWS Nitro Enclaves [3], Azure Confidential Computing [4] and AlibabaCloud Datatrust [5].

B. Entropy

Entropy could be used to describe the information amount or uncertainty of a random variable [18].

For a given random variable X, its Shannon entropy is $H(X) = -\sum_{x \in X} p(x) \log p(x)$. Among them, $p(x)$ is the probability of the occurrence of event x. The larger $H(X)$ is, the greater the uncertainty of X is. In addition, there are variants of Shannon entropy, such as min-entropy [19].

In the following of this paper, we try to quantify the privacy leakage of PPCS programs via entropy analysis.

C. Fuzz Testing

Fuzz testing (i.e. fuzzing) is a software testing technology that discovers software vulnerabilities by providing (semi-)automatically generated inputs to target software and monitoring crashes or assertion failures that occur during execution [20]. Fuzzing can be easily applied to large-scale programs due to its good scalability.

To better discover software vulnerabilities, fuzzers (i.e. fuzzing tools) usually focus on how to improve code coverage, that is, how to generate test cases that could trigger more different program execution paths. Such coverage-oriented fuzzers (e.g. AFL [21]) typically obtain code coverage information by instrumenting the code at compile time to decide whether the newly generated test cases should be retained. Then, they continues to mutate and generate new test cases based on the retained ones via strategies such as bit or byte flip, special bytes replacement, bytes increase or decrease, block deletion or insertion, and input splicing.

In the following of this paper, we try to combine coverage-oriented fuzzing and entropy analysis to detect PLVs in PPCS programs.

III. PLV Definition

The goal of FuzzLeaks is to find PLVs in PPCS programs.

The general form of a PPCS program is $F(x_1 \dots x_n) = (y_1 \dots y_n)$, in which F is the program logic, x_i is the sensitive input data of participant$_i$, and y_i is the observable output to participant$_i$. In this paper, we generalize the form of a PPCS program as $F(secret) = observation$.

A privacy leakage vulnerability (PLV) is a defect in software that could lead to the leakage of sensitive data. In a PPCS program $F(secret) = observation$, we assume that the output $observation$ and the program logic F could be obtained by data users, including the potential attackers. If there exists an $observation$ that could be used to infer some piece of the $secret$, it is considered that there exists a PLV in F.

Although PPCS provides plenty of privacy-preserve mechanisms, once there is a PLV in a PPCS program, it is possible for attackers to bypass these mechanisms and obtain sensitive data.

IV. FuzzLeaks Design

FuzzLeaks uses coverage-oriented fuzz testing to generate test cases triggering different execution paths of the target program and thus to carry out entropy analysis on the sampled input and output to detect PLVs. It must handle the following two challenges.

- The input of PPCS programs often contains a set of multidimensional data of indeterminate length, which means that the input domain may be extremely large. Test-based leakage estimation approaches that require test cases to traverse the entire input domain become less effective.
- PPCS programs extensively use conditional statements and loop statements to analyze input data, which will

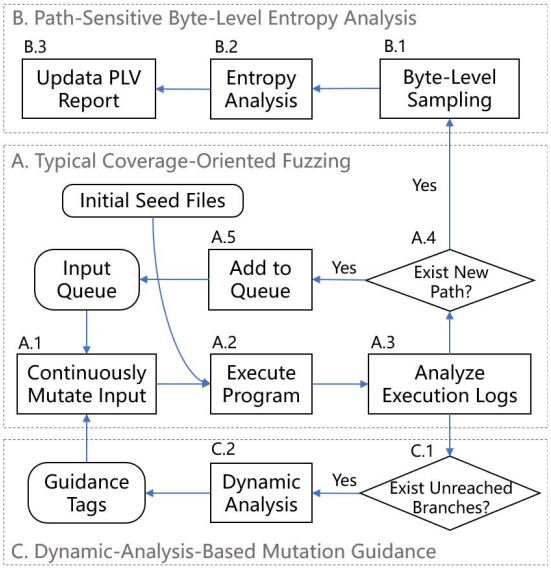

Fig. 2. Workflow of FuzzLeaks.

lead to a large number of branches and make it difficult for fuzzers to cover the code that causes privacy leakage.

Therefore, Fuzzleaks adopts two methods to solve these problems.

- A path-sensitive byte-level entropy analysis method to perform sampling that covering the target program without traversing the input domain, thus realizing effective entropy analysis for programs with extremely large input domain.
- A dynamic-information-flow-analysis based mutation guidance method to locate the unreached branches and guide the subsequent input mutation for better exploring these branches, thus realizing effective fuzzing for programs with complex data-processing logic.

Fig.2 shows the workflow of FuzzLeaks. The inputs in the input queue (initialized with the given seed files) will be continuously mutated (A.1), and the target program will be executed with each mutated input (A.2). Then, the execution logs will be analyzed (A.3) to judge whether there is a new execution path (A.4). If so, the input will be added to the input queue for further mutation (A.5).

At the same time, whenever a new input is added to the input queue, further byte-level sampling (B.1) and entropy analysis (B.2) will be carried out to quantify privacy leakage and determine whether there is PLV in the execution path triggered by this input (B.3).

In addition, FuzzLeaks will also judge whether there are branches encountered but not jumped into (C.1). If so, FuzzLeaks will further perform dynamic information flow analysis on each branch to locate the index range of the input piece that is closely related to the branch judgement (C.2). These results will be used to guide the subsequent mutation.

A. Quantifying Privacy Leakage with Entropy

PPCS programs usually access sensitive data (i.e. secrets) and send out relevant analysis results (i.e. observations). An intuitive idea is that we can understand the degree of privacy leakage as the difficulty of inferring the secret with the observation, that is, to which extent the uncertainty of the secret can be decreased when one gets the observation. Therefore, we can use Shannon entropy and information gain (also known as mutual information) to quantitatively evaluate the privacy leakage.

We regard a PPCS program as a mapping relationship F from the secret X to the observation Y:

$$Y = F(X) \tag{1}$$

$p(x)$ means the probability that X takes the value x. The Shannon entropy of the secret X is:

$$H(X) = -\sum_{x \in X} p(x) \, logp(x) \tag{2}$$

When the observation Y is known, the conditional entropy of the secret X is:

$$H(X|Y) = \sum_{y \in Y} p(y) \, H(X|Y = y) \tag{3}$$

The information gain that the observation Y brings to the secret X (i.e. the decreased uncertainty) is:

$$g(X, Y) = H(X) - H(X|Y) \tag{4}$$

In general, the greater the information gain $g(X, Y)$ brought by Y to X, the more uncertainty of X is decreased, and the greater the privacy leakage of $Y = F(X)$.

However, information gain could reflect the overall leakage, but fails when there is a special observation that makes the attacker easily and accurately infer the secret (i.e. occasional leakage). For example, $g(X, Y)$ of $Y = (0 : 1 \ ? \ X == 0)$ is not high, but the attacker can easily infer that X is 0 if he knows that Y is 0. Such situations usually occur when there are branch statements with strict conditions. Therefore, we further introduce quantification based on min-entropy [19] as follows to characterize the worst-case uncertainty and reflect the occasional leakage.

The vulnerability of the secret X means the maximum probability that an attacker could guess the value of X correctly in one try:

$$V(X) = \max_{x \in X} p(x) \tag{5}$$

The worst-case uncertainty (min-entropy) of X is:

$$H_\infty(X) = -\log V(X) \tag{6}$$

When the observation Y is known, the worst-case uncertainty of X is:

TABLE I
EXAMPLES OF LEAKAGE QUANTIFICATION.

Y=F(X), X=0,1...9	$g(X,Y)$	$\mathcal{L}(X,Y)$	Meaning
Y=int(X/3)	0.473	0.523	Relatively high overall leakage.
Y=(0:1 ? X==0)	0.141	1	Low overall leakage but high occasional leakage.

$$H_\infty(X|Y) = \min_{y \in Y} H_\infty(X|Y=y) \tag{7}$$

Therefore, the decrease of difficulty for an attacker to guess the value of the secret X correctly in one try due to observation of Y can be expressed as:

$$\mathcal{L}(X,Y) = H_\infty(X) - H_\infty(X|Y) \tag{8}$$

In this paper, we use both of the two methods to quantify privacy leakage. Table.I shows some examples of the quantification. When $g(X,Y)$ is large and $\mathcal{L}(X,Y)$ is close to $g(X,Y)$, it indicates that there may be an overall leakage. For example, the result of $Y = int(X/3)$ enables the attacker to significantly reduce the possible range of X. When $g(X,Y)$ is small but $\mathcal{L}(X,Y)$ is quite large, it indicates that there is no overall leakage but an occasional leakage, just like $Y = (0:1\ ?\ X == 0)$.

B. Path-Sensitive Byte-Level Entropy Analysis

The input of a PPCS program often contains a set of high-dimensional data of indeterminate length, that is, the domain of the input secret X is infinite. Such state explosion will lead to the insufficient coverage of the test cases, and then affect the accuracy of the test-based approaches.

Therefore, we perform byte-level sampling and entropy analysis for each test case that triggers a specific execution path, thus effectively detecting PLVs while avoiding state explosion. The procedure of this method is shown in Algorithm.1. For each given input (i.e. test case), we carry out byte by byte leakage detection (line 3-10): mutate $byte_i$ for n times, record the input and output, and evaluate the leakage with Shannon entropy and min-entropy. If any one is greater than the threshold ($log(n/3)$ for $g(X,Y)$, and 0.99 for $\mathcal{L}(X,Y)$), the $byte_i$ will be marked as a leaked byte.

In this way, the sampling times is reduced from n^{input_lenth} to $path\_num \times input\_lenth \times n$. However, when there is a statistical calculation of multiple input bytes (e.g. sum, average), the method may produce false positives (FP) because it regards different input bytes as independent elements and ignores the potential correlation between multiple input bytes.

Taking $Y = X[0] + X[1]$ as an example, since our method conducts independent sampling and entropy analysis on $X[0]$ and $X[1]$, it will be considered that both $X[0]$ and $X[1]$ were leaked, while such statistical calculation is usually compliant. Therefore, we further design a method to identify the statistical calculation and eliminate such FPs.

After marking the input bytes with abnormal result as leaked bytes, FuzzLeaks will further analyze the records to determine

Algorithm 1 Byte-level leakage quantification

Input: $input$, $program$
Output: $detectResult$
1: $sinkRelation \leftarrow$ empty vector of set
2: $reverseSinkRelation \leftarrow$ empty vector of set
3: **for** $i = 0 \rightarrow input.length()$ **do**
4: $records \leftarrow$ empty set of pair
5: **for** $cnt = 0 \rightarrow n$ **do**
6: $newInput \leftarrow$ MutateOneByte($input, i$)
7: $output \leftarrow$ RunProgram($program, newInput$)
8: $records$.insert(make_pair($newInput, output$))
9: **end for**
10: $SE\_leak, ME\_leak \leftarrow$ EntropyAnalysis($records$)
11: **if** ExistLeakage(SE_leak, ME_leak) **then**
12: $sinkRelation[i] \leftarrow$ GetSinkIndexs($records$)
13: **for all** $index$ in $sinkRelation[i]$ **do**
14: $reverseSinkRelation[index]$.push_back($i$)
15: **end for**
16: **end if**
17: **end for**
18: $detectResult \leftarrow$ empty vector
19: **for** $i = 0 \rightarrow input.length()$ **do**
20: $leakageTag \leftarrow 0$
21: **if** $sinkRelation[i]$.size() > 0 **then**
22: $leakageTag \leftarrow 1$
23: **for all** $index$ in $sinkRelation[i]$ **do**
24: **if** $reverseSinkRelation[index]$.size() == 1 **then**
25: $leakageTag \leftarrow 2$
26: **end if**
27: **end for**
28: **end if**
29: $detectResult[i] \leftarrow leakageTag$
30: **end for**
31: **return** $detectResult$

the $sinkRelation$ that represents which output bytes leak an input bytes and the $reverseSinkRelation$ that represents which input bytes are leaked by an output byte (lines 11-14). Then, a two-step detection will be carry out to identify the statistical calculation on the basis of the $sinkRelation$ and $reverseSinkRelation$ (lines 19-25): 1) if a output byte leaks more than one input byte, it is considered that the output byte is the statistical results of these input bytes, 2) if an input byte is leaked and all output bytes leaking it are statistical results, it is considered that the input byte is statistically calculated but not leaked ones.

Taking $Y = X[0]+X[1]$ as an example, since $Y[0]$ and $Y[1]$ both leak $X[0]$ and $X[1]$, it is considered that $Y[0]$ and $Y[1]$ are statistical results. Since $X[0]$ and $X[1]$ are only leaked by $Y[0]$ and $Y[1]$, it is considered that $X[0]$ and $X[1]$ are statistically calculated rather than leaked.

C. Dynamic-Analysis-Based Mutation Guidance

Some leakages occur only when the input data meets certain conditions. Therefore, we leverage coverage-oriented fuzzing

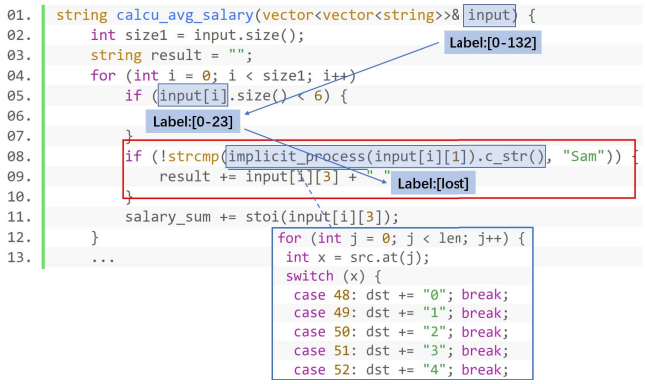

Fig. 3. Example of the loss of taint labels.

to generate test cases covering the PLV code before conducting the above byte-level quantification method. For this purpose, we focus on the code coverage of the fuzzer.

To improve the code coverage, state-of-the-art fuzzers try to determine the relationship between the input data and the conditions statements of specific branches, and then guide the mutation of input data to intentionally jump into the unreached branches. In most cases, they leverage dynamic taint analysis to track the input data by propagating taint labels.

However, PPCS programs usually contain a large number of conditional statements and loop statements to process and analysis the input data, which leads to complex implicit propagation of taint labels. Such codes are likely to destroy the information flow structure of dynamic taint analysis, making the taint-based mutation guidance invalid due to the loss of taint labels.

For example, the boxed code shown in Fig.3 can lead to the loss of taint labels because there is no explicit assignment from the variable src to variable dst, which greatly reduces the effectiveness of the subsequent mutation. Research works [22], [23] also show how to realize such attacks by maliciously constructing implicit control flow. Considering that PPCS programs may be developed by attackers, such problems are more likely to occur.

To solve this problem, we design a method to expand the taint labels through dynamic information flow analysis, which is shown in Algorithm.2. For each unreached branch, we mutate each input byte and record the value of the conditional variable (line 4-9). Then, we add the index of an input byte to the taint labels of this variable if its value changes (line 10-14). After that, the subsequent mutation will be guided with the expanded taint labels, thus explore the unreached branch more efficiently.

V. Evaluation

According to the approach introduced above, we implemented FuzzLeaks in Rust and C language with about 1400 lines based on Angora [24].

Specifically, the instrumentation module is implemented with the LLVM Pass framework [25] on the basis of *angora-clang*. The input mutation guidance based on dynamic information flow analysis is extended on the basis of Angora's fuzzing workflow. The leakage detection based on entropy analysis is implemented as a new independent module.

Algorithm 2 Mutation guidance based on dynamic analysis

Input: $program$, $input$
Output: $branch2labels$

1: $branch2labels \leftarrow$ empty map
2: $path, taintInfo \leftarrow$ RunProgram($program, input$)
3: **for all** unreached branch b in $path$ **do**
4: $branch2labels[b] \leftarrow$ GetTaintLabels($taintInfo, b$)
5: $tmpLabels \leftarrow$ empty set
6: $orig \leftarrow$ GetBranchValue($program, input, b$)
7: **for** $i = 0 \rightarrow input.length()$ **do**
8: $newInput \leftarrow$ MutateOneByte($input, i$)
9: $curr \leftarrow$ GetBranchValue($program, newInput, b$)
10: **if** $curr \neq$ INF and $curr \neq orig$ **then**
11: $tmpLabels$.insert(i)
12: **end if**
13: **end for**
14: $branch2labels[b] \leftarrow branch2labels[b] \cup tmpLabels$
15: **end for**
16: **return** $branch2labels$

```
int score1 = payment_history_score(default_record);
int score2 = credit_util_score(monthly_balance);
int score3 = credit_history_score(apply_history);
int score4 = new_credit_score(apply_history);
int score5 = credit_mix_score(apply_history);

static char ret[1000];
sprintf(ret, "%06d\0", score1 + score2 + score3 + score4 + score5);
sprintf(ret, "%s%s\0", ret, cJSON_PrintUnformatted(apply_history));
return ret;
```

The injected malicious code that leaks *apply_history* data

Fig. 4. FICO sample program and injected malicious code.

A. Case Study: Joint Credit Investigation

Joint credit investigation generates credit scores for users according to the personal credit data incluing repayment history, credit usage, credit application and so on. We implemented a PPCS program to calculate the credit score based on a sample model from FICO [26] and injected some malicious code that could leak the $apply\_history$ data. The program with malicious code is shown as Fig.4.

We applied FuzzLeaks to the program. FuzzLeaks performed fuzz testing to explore the program, and carried out byte-level entropy analysis on each generated test case. Fig.5 shows the analysis result of the initial test case, in which the red part indicates the input location with leakage. Since we only treated the value fields of the input JSON string as sensitive data in this case, only leakages in the value fields is shown. It can be seen that the leakage of the $apply\_history$ data caused by the injected malicious code is successfully detected.

```
In case: id:000000,orig:BankRecords.json, leak detected!
{"records":[{"bank1":{"default_record":[{"date":"2020-10-18","amoun
t":150}],"monthly_balance":[{"date":"2020-08-01","balance':300},{"d
ate":"2020-09-01","balance":400},{"date":"2020-10-01","balance":500
},{"date":"2020-11-01","balance":600},{"date":"2020-12-01","balance
":700}],"apply_history":[{"date":"2020-06-15","type":"credit_card",
"status":"approved"}]}},{"bank2":{"default_record":[],"monthly_bala
nce":[{"date":"2020-08-01","balance":3000},{"date":"2020-09-01","ba
lance":3000},{"date":"2020-10-01","balance":3000},{"date":"2020-11-
01","balance":3000},{"date":"2020-12-01","balance":3000}],"apply_hi
story":[{"date":"2020-04-15","type":"mortage","status":"approved"}]
}},{"bank3":{"default_record":[],"monthly_balance":[],"apply_histor
y":[{"date":"2020-11-08","type":"credit_card","status":"rejected"}]
```


Fig. 5. Detection result of FuzzLeaks on the program with malicious code.

TABLE II
PROGRAMS IN PLDA DATA SET AND THE EXPECTED RESULTS.

Program	Functionality	PLV
1.Counting	Count the number of rows of data that meet certain conditions.	None
2.Averaging	Calculate the average of some data items.	None
3.Summation	Calculate the sum of some data items.	None
4.Verification	Determine whether the input data meets certain conditions.	None
5.Desensitization	Calculate the hash of some data items.	None
6.Intersection	Finds all rows that exist in both sets of data.	None
7.MixedAnalysis	Combined of programs 1, 2, 5 and 6.	None
8.DirectLeak	Directly return some data items.	Exist
9.ReversibleLeak	Return some data items after reversible processing.	Exist
10.LoopLeak	Take some data items as the termination condition of loops, thereby indirectly leaking them.	Exist
11.BranchLeak	Take some data items as the judgment condition of branches, thereby indirectly leaking them.	Exist
12.ConditionalLeak	Directly leak some data items when the input data meets certain conditions.	Exist
13.ConfusedLeak	Directly leak some data items when the input data meets certain conditions while performs confusing processing with ways in programs 10 and 11 on the variables in the conditional statements in advance.	Exist
14.MixedLeak	Combined of programs 10, 11, 12 and 13.	Exist

B. Controlled Experiments

1) Experiment Design: The experiments on FuzzLeaks attempt to answer the following two research questions.

RQ 1 How does FuzzLeaks compare with other PLV-detection approaches, and does the path-sensitive byte-level entropy analysis work?

RQ 2 How effective is the dynamic-information-flow-analysis-based mutation guidance?

We conduct experiments on the PLDA data set [14] and the LAVA-M data set [15] to answer these research questions. PLDA is a data set for privacy leakage detection of data analysis programs, which contains 7 compliant programs (without PLVs) and 7 non-compliant programs (with PLVs). The programs in the PLDA data set and the expected detection results are shown in Table.II. LAVA-M is one of the most commonly used data sets for evaluating fuzzers. It selects four widely used real-world programs *base64*, *md5sum*, *uniq* and *who* from *coreutils* (the GNU core utilities), and injects a total of 2265 verified vulnerabilities into them. Specifically, the injected program *base64* contains 255 lines of code and 44 vulnerabilities, *md5sum* contains 785 and 57, *uniq* contains 546 and 28, and *who* contains 4426 and 2136.

TABLE III
COMPARISON OF FUZZLEAKS AND OTHER WORKS.

Program	Expectation	FuzzLeaks	PhASAR	DFSan	leakiEst
1.Counting	None	None	**Found**	None	**Found**
2.Averaging	None	None	**Found**	**Found**	None
3.Summation	None	None	**Found**	**Found**	None
4.Verification	None	None	**Found**	None	None
5.Desensitization	None	None	None	None	**Found**
6.Intersection	None	**Found**	**Found**	**Found**	**Found**
7.MixedAnalysis	None	None	**Found**	**Found**	**Found**
8.DirectLeak	Found	Found	Found	Found	Found
9.ReversibleLeak	Found	Found	Found	Found	Found
10.LoopLeak	Found	Found	Found	Found	Found
11.BranchLeak	Found	Found	Found	**None**	Found
12.ConditionalLeak	Found	Found	Found	Found	**None**
13.ConfusedLeak	Found	Found	Found	**None**	**None**
14.MixedLeak	Found	Found	Found	**None**	**None**
Precision	/	87.5%	53.85%	50%	50%
Recall	/	100%	100%	62.5%	62.5%
F1-Score	/	0.93	0.70	0.56	0.56
Accuracy	/	92.86%	57.14%	50%	50%

For all experiments, We use a docker container with Ubuntu 16.04 (AMD64) featuring Intel Core i5-11600KF @ 3.9GHz with 8GB of memory to run the experiments. All experiments were conducted three times in a single core configuration and the median results were displayed.

2) Comparison with PhASAR, DFSan and leakiEst (RQ1): For RQ1, we took the static analysis framework PhASAR (version v0521) [27], the dynamic data flow analysis framework DataFlow Sanitizer (DFSan, the version built into Clang 7.0.0) [28] and the quantitative leakage estimation tool leakiEst (version 1.4.9) [7] as the comparison objects, and conducted experiments on the PLDA data set (the LAVA-M data set is used for common fuzzing tasks).

In the FuzzLeaks experiments, we took a simple test case that cannot trigger PLVs in programs 12, 13 and 14 as the initial fuzzing input. In the PhASAR and DFSan experiments, we took *fread()* as the taint source, *printf()* as the taint sink, and *str_hash()* as a harmless treatment. In the PhASAR experiments, we carried out static taint analysis on the basis of its IFDSTaintAnalysis module and took the source code of the programs as input. In the DFSan experiments, we carried out dynamic taint analysis on the basis of its data flow analysis interfaces and took test cases that can trigger all the PLVs as input. In the leakiEst experiments, we took all the sampling records in experiments with FuzzLeaks as input. We used the two modes (Shannon entropy and min-entropy) of leakiEst to detect PLVs. We considered a leakage to exist as long as there is one mode reporting a leakage.

The experimental results are shown in Table.III. For Programs without PLV (program 1 to 7), FuzzLeaks produced a FP only in program 6, while PhASAR, DFSan and leakiEst produced FPs 6, 4, and 5 times respectively. Among them, because program 6 does return a part of the input data, these

methods all report that there is a PLV. For Programs with PLV (program 8 to 14), FuzzLeaks and PhASAR found all PLVs, while DFSan and leakiEst produced false negatives 3 times respectively. The results show that FuzzLeaks significantly outperforms other approaches in terms of precision, precision and recall and F1-score.

Since PhASAR adopts a relatively loose taint propagation rule, it successfully detected all PLVs but produced a lot of FPs. Unlike PhASAR, in order to avoid the huge overhead caused by the excessive pollution (propagate too many taint labels), DFSan will not propagate the taint labels when dealing with the implicit assignment through branches (e.g. in program 11). Therefore, DFSan failed to detect PLVs in programs 11, 13 and 14 but did not produce the FPs to programs 1 and 4 compared with PhASAR.

The difference between the results of PhASAR and DFSan reflects our motivation to use entropy analysis instead of taint analysis for PLV detection, that is, it is extremely difficult to design appropriate implicit taint propagation rules for the complex data-processing programs.

LeakiEst quantitatively evaluates privacy leakage based on given inputs and outputs, and does not care how to obtain them. In the experiments with leakiEst, we used those generated during FuzzLeaks's workflow, which eliminates the difference in sampling. Since leakiEst is mainly designed for cases that the secret input is a single variable, it performs coarse-grained entropy analysis on the whole secret input rather than the fine-grained analysis like FuzzLeaks, which makes it difficult to accurately detect PLVs when applied to PPCS programs with extremely large input domain. The improvement of FuzzLeaks relative to leakiEst reflects the effectiveness of the proposed path-sensitive byte-level entropy analysis method.

In addition, dynamic approaches like leakiEst and DFSan are limited by the completeness of test cases. If the test cases cannot trigger the execution of the code that includes a PLV, dynamic approaches will fail to find the PLV. In our experiments, we provide test cases covering all PLV code to leakiEst and DFSan, so they may be able to detect PLVs in program 12, 13 and 14. Otherwise, all these PLVs would be impossible to be detected. This is the reason why we perform entropy analysis while doing coverage-oriented fuzzing and further design the mutation guidance method.

In summary, FuzzLeaks is more suitable for PLV detection in PPCS programs than approaches based on taint analysis and quantitative estimation. FuzzLeaks outperforms traditional approaches in accuracy by 35.72% on the PLDA dataset, in which the path-sensitive byte-level entropy analysis brings a great improvement.

3) Mutation Guidance Strategy (RQ2): For RQ2, we took the state-of-the-art open-source fuzzer Angora (version 1.2.2) [24] as the comparison object, which outperforms fuzzers including FUZZER, SES, VUzzer, Steelix and AFL on the LAVA-M data set, and conducted experiments on both the PLDA data set and the LAVA-M data set.

TABLE IV
EFFECTIVENESS OF THE PROPOSED MUTATION GUIDANCE METHOD.

Program	Line Coverage			Branch Coverage		
	Angora	FuzzLeaks	Improvement	Angora	FuzzLeaks	Improvement
1.Counting	45	45	/	45	46	2.2%
2.Averaging	42	44	4.8%	37	38	2.7%
3.Summation	40	41	2.5%	34	35	2.9%
4.Verification	44	44	/	39	39	/
5.Desensitization	37	38	2.7%	44	45	2.3%
6.Intersection	42	42	/	46	46	/
7.MixedAnalysis	49	49	/	54	54	/
8.DirectLeak	38	38	/	44	44	/
9.ReversibleLeak	37	38	2.7%	51	52	2.0%
10.LoopLeak	56	56	/	56	56	/
11.BranchLeak	68	68	/	110	110	/
12.ConditionalLeak	39	40	2.6%	43	44	2.3%
13.ConfusedLeak	116	119	2.6%	179	186	3.9%
14.MixedLeak	125	129	3.2%	188	205	9.0%

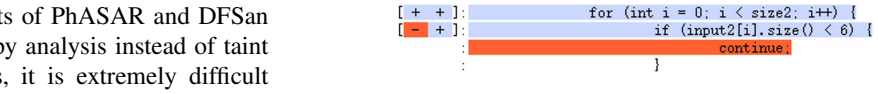

Fig. 6. Unreached branch 1 of Angora.

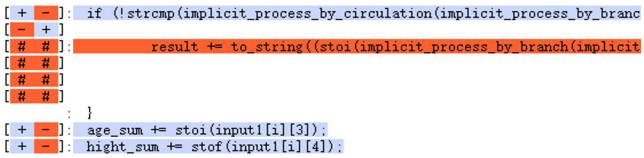

Fig. 7. Unreached branch 2 of Angora.

We compared the fuzzing module of FuzzLeaks with Angora to evaluate the effectiveness of the mutation guidance method adopted in FuzzLeaks. The only difference between the fuzzing module of FuzzLeaks and Angora is the adoption of the mutation guidance method. All experiments adopted the default parameters of Angora 1.2.2, and we used gcov 5.4.0, lcov 1.12 and afl-cov 0.6.2 for code coverage statistics.

Table.IV. shows the experimental results on the PLDA data set. The execution time was 1 minute for each program. In programs 2, 3, 5, 9 and 12, Angora failed to cover the code in the branch shown in Fig.6. This is due to the implicitly process for the taint variable in *input[2].size()*, which leads to the loss of taint tracking information. In programs 13 and 14, Angora failed to cover the code in the branch shown in Fig.7. This is because there is some confusing code that processes the string to be compared in these two programs, which makes Angora unable to take intentional mutation due to the loss of taint tracking information and failed to explore this branch.

It should be noted that FuzzLeaks did not change any specific mutation strategy. All it did is use the proposed dynamic analysis method to expand the taint labels that are used to guide the mutation, which shows the effectiveness of the method and its compatibility with the existing mutation strategies.

We further observed how code coverage changes over time. Fig.8 shows the fuzzing on program 14 by Angora and FuzzLeaks. It can be seen that after adding the mutation

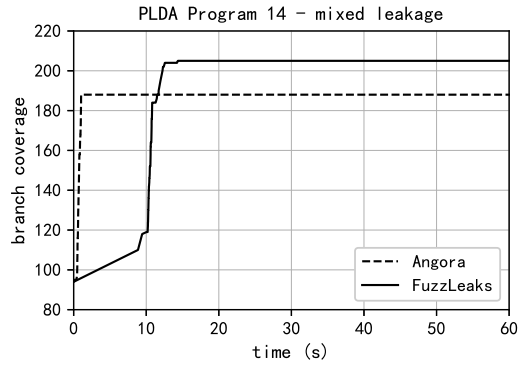

Fig. 8. Fuzzing on program 14 of PLDA by Angora and FuzzLeaks.

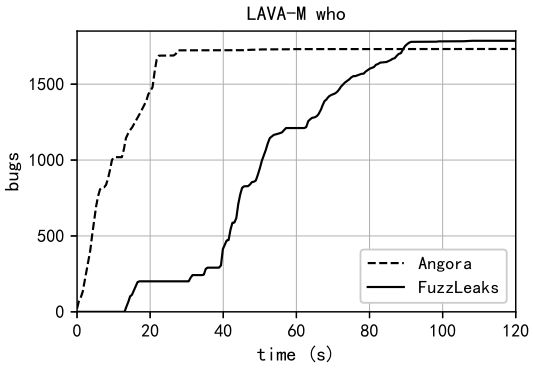

Fig. 9. Fuzzing on LAVA-M who by Angora and FuzzLeaks.

guidance method, the fuzzing consumed more time in the early stage to expand the lost taint labels on the variables of the unreached branches, so as to explore these branches more effectively in the subsequent stage and finally achieve better coverage.

In addition, we also verified the effectiveness of the mutation guidance method adopted by FuzzLeaks on the LAVA-M data set. LAVA-M contains four programs *base64*, *md5sum*, *uniq* and *who* that injected several vulnerabilities. Since Angora and FuzzLeaks both found all the vulnerabilities injected in *base64*, *md5sum* and *uniq*, we only show the experimental result on *who*. As shown in Fig.9, FuzzLeaks found 50 more vulnerabilities than Angora after running for 2 hours, which means that the mutation guidance method adopted in FuzzLeaks also works on normal fuzzing tasks.

The experimental results show that the dynamic-information-flow-analysis-based mutation guidance method adopted in FuzzLeaks can effectively improve the coverage of fuzz testing especially for PPCS programs.

C. Discussion

1) Although FuzzLeaks has adopted many optimizations to better detect PLVs, it still cannot ensure that there are no PLVs in the program that passes the detection.

2) In the process of entropy analysis, the selection of sampling times and entropy threshold has influence on the detection results to some degree. Currently, they are determined in a empirical way.

3) Compared with dynamic taint tracking, the proposed mutation guidance method is more effective when dealing with programs that have many implicit assignments, but it will cause more overhead. Therefore, it is more recommended to use it as a supplement when dynamic stain tracking fails.

4) At present, FuzzLeaks is not suitable for programs with randomness. Fortunately, the execution of PPCS programs often depends only on the input data.

5) As a fuzzing-based method, FuzzLeaks requires more time for programs with high computational complexity. Of course, it is a common problem with test-based approaches.

VI. Related Work

We focus on the PLV detection in PPCS programs. The related work can be divided into the following three categories: 1) vulnerability detection for services, 2) leakage detection based on taint analysis, and 3) leakage detection based on quantitative estimation.

A. Vulnerability Detection for Services

The effect of vulnerability detection tools for services largely depends on the particularity of the application scenario (e.g. the type of target service, the type of vulnerability) [29]. Many works have targeted designs for vulnerability detection scenarios in different services.

Injection vulnerabilities are the most common vulnerabilities in Web services, which allow attackers to alter the construction of backend SQL statements or modify an XPath query to access the database or XML documents in a way differing from the programmer's intention. CIVS-WS [30] detects existing SQL and XPath injection vulnerabilities in web services code by matching incoming commands during attacks with the valid set of commands previously learned . Sign-WS [31] is a black-box detection tool for injection vulnerabilities, which uses attack signatures and interface monitoring to increase the visibility of the penetration testing process without needing to access the internals of web services. SQLEXP [32] is an effective testing approach based on Feature Matrix (FM) to test SQL Injection Vulnerabilities (SQLIV) in running web applications with a dynamic matrix selection algorithm to find out the optimum FM during SQLIV penetration test procedure automatically.

Data races are common in service concurrency. The execution of service components usually involves multiple threads, and resource sharing among threads may cause data races, resulting in performance degradation or execution errors. SDA-Cloud [33] offers trace-based dynamic analysis on service components using a novel multi-VM architecture approach in the cloud. ServiceRacer [34] is a data race detection tool which provides a framework that can automatically detect data races for composite Web services by static analysis and constraint solving. In addition, other works pay attention to DOS vulnerabilities in Web services [35] and memory leakage vulnerabilities in cloud services [36].

Different from all these works, we focus on PLVs in PPCS programs and propose a PLV-detection approach.

B. Leakage Detection Based on Taint Analysis

Taint-analysis-based approaches can detect PLVs by marking sensitive data and tracking the spread of marked data in the program.

Many of these works focus on privacy leakage detection of Android applications. Some works combine static taint analysis and alias analysis to design approaches according to the characteristics of Android framework. For example, FlowDroid [10] pays attention to the call graph construction of the callback model of Android life cycle. IccTA [11] focuses on the spread of context information between Android components. Some methods utilize dynamic taint analysis for detection. For example, TaintDroid [12] modifies Dalvik VM to intercept system calls for runtime collection and analysis of taint information. SmartDroid [13] finds suspicious paths through static analysis at first and then executes corresponding UI events for dynamic taint analysis.

The limitation of these approaches is that the extensively used conditional statements and loop statements in PPCS programs make it extremely difficult to design appropriate implicit taint propagation rules to deal with the PLV-detection task.

C. Leakage Detection Based on Quantitatively Estimation

Quantitatively-estimation-based approaches usually analyze the input and output of the target program to evaluate the information leakage. To achieve better results, it is better to enumerate the input domain as much as possible.

leakiEst [7] and LeakWatch [6] are the most famous works, which measure the information leakage by repeatedly running the system on the selected secret inputs and calculate the the relative frequency of the secret inputs and the respective outputs. F-BLEAU [8] utilizes machine learning to strengthen the scalability of such quantitative estimation approaches. Agrigento [9] conducts differential quantitative analysis on the association between sensitive user data and network traffic to detect privacy leakage of Android applications.

The limitation of these approaches is their high requirement on the completeness of test cases. Once the test cases fail to cover the code that leaks privacy, the PLV will escape detection. Even with the PLV code covered, the extremely large input domain of PPCS programs will still lead to lower detection performance.

VII. Conclusion

PPCS is widely used to preserve data privacy during data sharing and cooperation. However, PPCS program may still expose sensitive data outside imperceptibly along with the returned result if there are intentionally or careless designed PLVs. This paper proposes a fuzzing-based approach named FuzzLeaks to detect PLVs. It uses coverage-oriented fuzz testing to generate test cases for checking PPCS programs and thus to carry out leakage estimation to detect PLVs. In addition, it leverages path-sensitive byte-level entropy analysis to effectively quantify privacy leakage under the extremely large input domain, and guides the fuzzing mutation based on dynamic information flow analysis to improve the coverage of fuzz testing on complex data-processing programs. We implement FuzzLeaks and conduct experiments on the PLDA data set and LAVA-M data set to validate FuzzLeaks. Results show that FuzzLeaks outperforms traditional approaches in accuracy by 35.72% on the PLDA dataset and the dynamic-analysis-based mutation guidance adopted by FuzzLeaks can even resulted in 50 more non-PLV bugs found on the LAVA-M data set.

References

[1] "Facebook-cambridge analytica data scandal," https://id.wikipedia.org/wiki/Skandal_data_Facebook\OT1\textendashCambridge_Analytica.

[2] F. Kerschbaum, "Privacy-preserving computation," in *Annual Privacy Forum*. Springer, 2012, pp. 41–54.

[3] "Aws nitro enclaves," https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[4] "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[5] "Alibaba cloud datatrust," https://www.aliyun.com/activity/Datatrust/home.

[6] T. Chothia, Y. Kawamoto, and C. Novakovic, "Leakwatch: Estimating information leakage from java programs," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 219–236.

[7] T. Chothia, Y. Kawamoto, and C. Novakovic, "A tool for estimating information leakage," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 690–695.

[8] G. Cherubin, K. Chatzikokolakis, and C. Palamidessi, "F-bleau: Fast black-box leakage estimation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 835–852.

[9] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *NDSS*, 2017.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[11] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[13] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012, pp. 93–104.

[14] "Plda data set," https://github.com/samsuzhang/PLDA-data-set.

[15] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[16] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 1982, pp. 160–164.

[17] GlobalPlatform, "Tee system architecture," https://en.wikipedia.org/wiki/Facebook-ambridge_Analytica_data_scandal.

[18] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[19] G. Smith, "Quantifying information flow using min-entropy," in *2011 Eighth International Conference on Quantitative Evaluation of SysTems*. IEEE, 2011, pp. 159–167.

[20] "Fuzz testing of application reliability," http://www.cs.wisc.edu/~bart/fuzz/.

[21] "American fuzzy lop," https://lcamtuf.coredump.cx/afl/.
[22] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.
[23] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices." in *SECRYPT*, vol. 96435, 2013.
[24] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.
[25] "Llvm pass framework," https://www.llvm.org/docs/WritingAnLLVMPass.html.
[26] "Fico," https://www.fico.com/en/latest-thinking.
[27] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural static analysis framework for c/c++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 393–410.
[28] "Dataflowsanitizer design document," https://clang.llvm.org/docs/DataFlowSanitizerDesign.html.
[29] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, 2014.
[30] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of sql/xpath injection vulnerabilities in web services," in *2009 IEEE International Conference on Services Computing*. IEEE, 2009, pp. 260–267.
[31] N. Antunes and M. Vieira, "Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services," in *2011 IEEE International Conference on Services Computing*. IEEE, 2011, pp. 104–111.
[32] L. Liu, J. Xu, H. Yang, C. Guo, J. Kang, S. Xu, B. Zhang, and G. Si, "An effective penetration test approach based on feature matrix for exposing sql injection vulnerability," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 123–132.
[33] C. Jia, C. Yang, W. K. Chan, and Y. T. Yu, "Sda-cloud: a multi-vm architecture for adaptive dynamic data race detection," *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 80–93, 2016.
[34] L. Xu, Q. Sun, B. Xu, and W. Zhang, "Statically detect data races for ws-bpel web services by constraint solver," in *2016 IEEE International Conference on Web Services (ICWS)*. IEEE, 2016, pp. 476–483.
[35] A. Falkenberg, C. Mainka, J. Somorovsky, and J. Schwenk, "A new approach towards dos penetration testing on web services," in *2013 IEEE 20th International Conference on Web Services*. IEEE, 2013, pp. 491–498.
[36] A. Jindal, P. Staab, J. Cardoso, M. Gerndt, and V. Podolskiy, "Online memory leak detection in the cloud-based infrastructures," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 188–200.