REACT: IR-Level Patch Presence Test for Binary

Qi Zhan
Zhejiang University
Hangzhou, China
qizhan@zju.edu.cn

Xing Hu*
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Xin Xia
Software Engineering Application Technology Lab, Huawei
China
xin.xia@acm.org

Shanping Li
Zhejiang University
Hangzhou, China
shan@zju.edu.cn

ABSTRACT

Patch presence test is critical in software security to ensure that binary files have been patched for known vulnerabilities. It is challenging due to the semantic gap between the source code and the binary, and the small and subtle nature of patches. In this paper, we propose React, the first patch presence test approach on IR-level. Based on the IR code compiled from the source code and the IR code lifted from the binary, we first extract four types of feature (return value, condition, function call, and memory store) by executing the program symbolically. Then, we refine the features from the source code and rank them. Finally, we match the features to determine the presence of a patch with an SMT solver to check the equivalence of features at the semantic level.

To evaluate our approach, we compare it with state-of-the-art approaches, BinXray and PS3, on a dataset containing binaries compiled from different compilers and optimization levels. Our experimental results show that React achieves scores of 0.88, 0.98, and 0.93, in terms of precision, recall, and F1 score, respectively. React outperforms the baselines by 39% and 12% in terms of the F1 score, while the pure testing speed of our approach is 2x faster than BinXray and 100x faster than PS3. Furthermore, we conduct an ablation study to evaluate the effectiveness of each component in React, which shows that SMT solver and refinement can contribute to 16% and 10% improvement in terms of the F1 score, respectively.

KEYWORDS

Patch presence test, security, program analysis

ACM Reference Format:
Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. 2024. REACT: IR-Level Patch Presence Test for Binary. In *Proceedings of 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE 2024, October 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

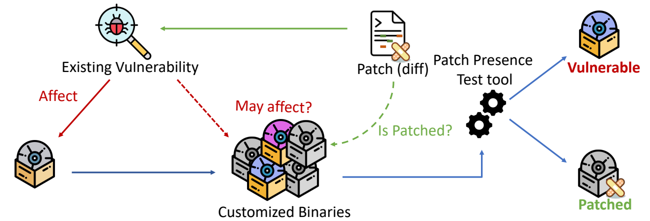

Figure 1: Background of patch presence test

1 INTRODUCTION

Open-source software is widely used in various fields, such as operating systems [36], web servers [20], and databases [21]. Many companies and organizations rely on open-source software to build their products and services. According to the Veracode report [37], 96% of organizations use open-source libraries.

Although the company benefits from open-source software, it also introduces some security problems. As shown in Figure 1, a common workflow in companies is to customize the open-source library to meet the specific requirements of users and release only binaries to them. Users are usually aware of the information about open source libraries used. Problems arise when vulnerabilities are found in open-source software. Attackers may steal important information or remotely take control of the entire system through these vulnerabilities caused by used open-source components. Users want to know whether the binary files they are using are vulnerable or not. Furthermore, the vulnerability can be fixed by applying a patch, and it is essential to ensure that these binary files have been patched for the corresponding vulnerabilities. The process for ensuring this is referred to as the ***patch presence test***.

The patch presence test determines whether a specific patch has been applied to a target binary. The input consists of two parts: (1) information about a particular patch and the source code of corresponding project; (2) the *target* binary to be tested. The output of the patch presence test is a binary decision as to whether the specific patch is present in the target binary.

Determining whether a given binary file has been patched for a corresponding vulnerability is challenging. The key challenge is the semantic gap between the source code and the binary, making it difficult to reason about the presence of a small modification in the binary. Many techniques have been proposed [24, 38, 40, 42, 44] to

automate patch presence test in binaries or Java bytecode. A typical workflow for the binary-level patch presence test is to compile the source code before and after the patch commit to the corresponding binaries. Then, they compare the binaries with the target binary to determine whether the patch has been applied. These approaches formulate the problem from "source to binary" to "binary to binary".

Although these approaches have achieved reasonable results, the lack of semantic information makes analyzing binary code difficult, especially when the target binary is compiled from different compiler optimizations. Generally, the patch file is used to map the line number of the source code modifications to the binary code according to the debug information so that one can recognize which part of the assembly code is added in the patch. However, the rich information in the source code is ignored. In addition, these approaches usually focus on binary in one architecture (FIBER for Arm64 [44], PS3 for Amd64 [42]) and cannot be easily migrated to other architectures.

In this paper, to address the aforementioned challenges, we propose an approach called REACT, referred to i**R** l**E**vel p**A**tch presen**C**e **T**est, to conduct patch presence test on the intermediate representation (IR) level. We obtain the reference IR codes before and after the patch by compiling the source code and target IR code by binary lifting [2]. Compared with previous work, we formulate the problem from the "source to binary" to "IR to IR". IR can help bridge the semantic gap between the source code and the binary, as semantic relationships in the same language are more straightforward. Furthermore, analyzing IR is more convenient than analyzing binary codes. The other advantage is that it can be applied to different architectures and executables in different operating systems directly as long as they can be lifted to IR.

To take advantage of the IR code, we perform fine-grained analysis on the IR level, including feature generation, refinement technique, and feature matching:

❶ **Feature Generation**. We extract four types of features from the lifted and complied IR code, including return value, function call, memory store, and condition by execution basic blocks symbolily.

❷ **Feature Refinement.** The generated features may not be suitable for matching directly due to compiler optimization and local variables. We propose a novel feature refinement technique to simplify the features while keeping the unique parts and ranking them based on their importance.

❸ **Feature Matching.** We compare the features extracted from the complied and the lift IR code one by one to determine the presence of a patch. We utilize an SMT solver to compare the equivalence of features at the semantic level.

Considering the widespread use of LLVM [7, 28, 35], mainstream binary lifting tools [1, 19, 41] choose LLVM IR for the lifting target. Therefore, we implement our approach on top of LLVM IR and use RetDec [26] to lift binaries. We expand the dataset proposed by Zhan et al. [42] and compare our approach with two state-of-the-art approaches, BinXray [40] and PS3 [42]. The experimental results show that our approach outperforms the baselines by 39% and 12% in terms of the F1 score. We also evaluate the results of our approach compared to PS3 for every combination of compiler and optimization level, demonstrating that our approach is more robust to compiler optimization. In addition, we conducted an ablation study to evaluate the effectiveness of each component proposed in our approach, which shows that the refinement technique and SMT solver bring 10% and 16% improvements in terms of the F1 score, respectively. Finally, we compare the efficiency of our approach with the baselines. The results show that REACT is able to detect the patch in 0.025 seconds on average, which is 100 times faster than PS3 and 2 times faster than BinXray.

Contributions: In this paper, we make the following contributions:

- We formulate the patch presence test task from "source to binary " to "IR to IR" and propose REACT, the first IR-based approach for patch presence test to our best knowledge.
- We design a fine-grained analysis framework for patch presence test, including feature generation, refinement, and match.
- We systematically evaluate our approach on the dataset, and the results show its effectiveness with high precision and speed.

The remainder of the paper is organized as follows. Motivation is outlined in Section 2, while the design and implementation of the framework are detailed in Section 3. The evaluation steps and results of REACT are presented in Section 4. Section 5 examines the threats to validity of our approach and investigates why some patches are not detected. Section 6 provides a summary of related research. We conclude the paper in Section 7.

2 MOTIVATION

This section discusses the motivation behind our approach through several read-world patches.

2.1 Binary Lifting

To see why binary lifting is useful in patch presence test, we list the source code, one possible target binary, compiled IR, and lifted IR for CVE-2021-23841 [14] in Figure 2. The patch is to add a new check for `f` in the original C code. As we track the data dependency of `f`, we can find that the value of `f` is return value of `X509_NAME_oneline` and the argument of the function call is `a->cert_info.issuer`, `NULL`, `0`, respectively. Due to the semantic gap between the source code and the binary code, it is difficult to map every modified statement to binary. Although we can find that underlined statement in the target binary is indeed the same condition check, data tracking and recovery are much more difficult and tedious. In addition, we have to take many instruction set architecture-dependent features into account, e.g. calling convention, register usage, so that we can analyze the binary correctly.

Compared with source-to-binary mapping, it is much easier to compare the compiled and lifted IR. The underlined statement in the compiled and lifted IR represents the semantic of the patch. The *getelementptr* instruction in the compiled IR is to calculate the offset to access elements of arrays and structs, which corresponds to `add arg1, 48` in the lifted IR. The later *call* and *icmp* (int-comparison) instruction is to check the return value of the function call. Based on both IR, we can decide the target binary as patched. We can unify all analyses and operations for source and binary as we compile or lift them to the same language, which provides the foundation for fine-grained analysis in the following.

The feature extracted from the above example is a condition to check if the function call value is `NULL`. In addition to the condition

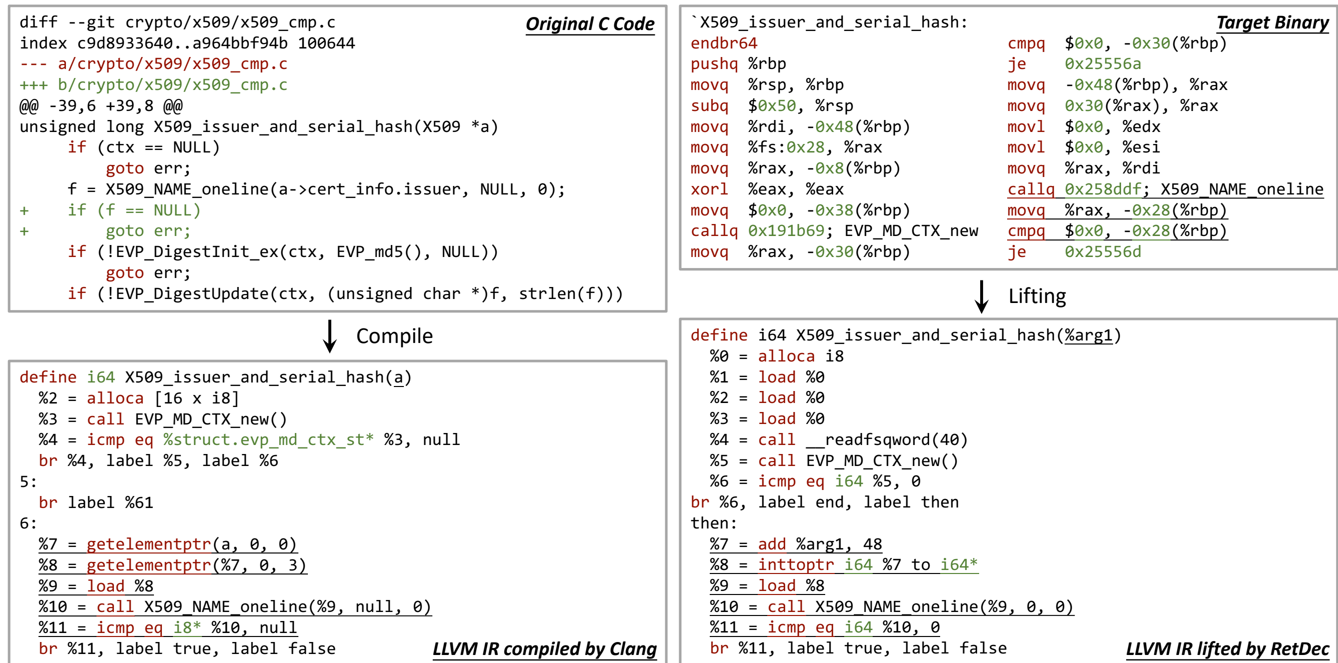

Figure 2: A case study comparing LLVM IR lifted by RetDec with LLVM IR compiled by clang. Both IR codes have been extensively simplified for readability.

and function call, we also select the return value and memory store as features to extract.

2.2 Patch Analysis

Although tracking backward data dependency is a general approach for feature extraction [22, 23, 44], it is not enough. For example, in the first patch of Figure 3a, it fixes a wrong assignment to `ret` on line 4, which is later used as a return value. Since both vulnerable and patched functions call `X509_STORE_CTX_get_error` with the same argument, i.e., `ctx`, the backward data flow and dependency are the same in both versions. Only after analyzing the forward data flow and finding the change of return value can we decide whether the binary has been patched.

Beyond generation, we also need to consider the matching process. Feature generation cannot be perfect, especially for complex arithmetic operations, for example, division or bitwise operations. The compiler may optimize the code and generate very different instructions from the same source code [5]. In addition, as register allocation and instruction scheduling are different, it is difficult to locate a modification of local variables accurately. Complex operations and local variables may affect the result if we compare the feature exactly at the syntactic level. The above limitations motivate us to simplify the features. As long as the feature is unique under the context of the other features, we can remove some parts of the feature to make it more general. For example, in the second patch of Figure 3a, the only differences between the vulnerable code and the patched code are the values of `group_top + 1` and `group_top + 2`, while the other arguments of the function call are the same. Thus, we can ignore the common part of the feature and only keep the unique part. The features can be simplified to `bn_wxpand(-, bn_get_top(-) + 1)` and `bn_wxpand(-, bn_get_top(-) + 2)`, where "-" is a placeholder for any value to match. This process is called **feature refinement** in our approach.

The last lesson we learn from the patches is that not all features are equally important. Although most patches are small and subtle [29], there are some patches with more significant changes. For example, in Figure 3b, there are seven deletion lines and seven insertion lines. In this case, there is more than one feature to extract and we can find that the call to the function `ec_guess_cofactor` on line 16 is the most unique feature. If the function call feature is found in the target code, we can confidently claim the presence of the patch, since the entire function is added in the patch. The other features are less important and can be used as supplements. This motivates us to rank the features based on their importance and focus on more unique features.

We conclude the discussion with the following insights.

💡 **Insight 1.** To generate the complete features for code diff, we need to track data flow in both forward and backward directions.

💡 **Insight 2.** To decide the presence of a patch, we only need to focus on the unique part of the feature.

💡 **Insight 3.** Not all features are equally important; features can be ranked to focus on the most important ones.

Inspired by the above insights, we can sketch the outline of our approach based on binary lifting. First, we consider the four types of feature discussed above and define them formally. To track bidirectional data flow, we symbolically execute the code and collect features; To improve the generality of the features, we refine the

```
@@ -59,9 +59,10 @@ static int ocsp_verify_signer(X509 *signer, int response,
     ret = X509_verify_cert(ctx);
     if (ret <= 0) {
-        ret = X509_STORE_CTX_get_error(ctx);
+        int err = X509_STORE_CTX_get_error(ctx);
         ERR_raise_data(ERR_LIB_OCSP, OCSP_R_CERTIFICATE_VERIFY_ERROR,
-                       "Verify error: %s", X509_verify_cert_error_string(ret));
+                       "Verify error: %s", X509_verify_cert_error_string(err));
         goto end;

@@ -206,8 +206,8 @@ int ec_scalar_mul_ladder(const EC_GROUP *group, EC_POINT *r,
    cardinality_bits = BN_num_bits(cardinality);
    group_top = bn_get_top(cardinality);
-    if ((bn_wexpand(k, group_top + 1) == NULL)
-        || (bn_wexpand(lambda, group_top + 1) == NULL)) {
+    if ((bn_wexpand(k, group_top + 2) == NULL)
+        || (bn_wexpand(lambda, group_top + 2) == NULL)) {
        ECerr(EC_F_EC_SCALAR_MUL_LADDER, ERR_R_BN_LIB);
        goto err;
    }
```

(a) Patches of CVE-2022-1343 [15] and CVE-2018-0735 [9]

```
@@ -281,17 +370,17 @@ int EC_GROUP_set_generator(EC_GROUP *group, const
↪   EC_POINT *generator,
-    if (order != NULL) {
-        if (!BN_copy(group->order, order))
-            return 0;
-    } else
-        BN_zero(group->order);
+    if (!BN_copy(group->order, order))
+        return 0;

-    if (cofactor != NULL) {
+    /* Either take the provided positive cofactor, or try to compute it */
+    if (cofactor != NULL && !BN_is_zero(cofactor)) {
        if (!BN_copy(group->cofactor, cofactor))
            return 0;
-    } else
+    } else if (!ec_guess_cofactor(group)) {
        BN_zero(group->cofactor);
+        return 0;
+    }
```

(b) Patch for CVE-2019-1547 [13]

Figure 3: Examples of patches from real-world projects, adjusted to fit the page.

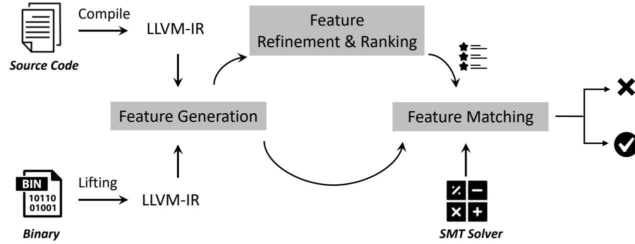

Figure 4: Overview of our approach

$$
\begin{aligned}
\text{Expr } e &\coloneqq e \text{ binop } e \mid \text{unop } e \\
&\quad \mid \text{call}(e, e_1, \dots, e_n) \mid \text{const} \\
&\quad \mid \text{mem}(e) \mid \text{arg}(i) \mid \text{alloc}(i) \\
\text{unop } &\coloneqq - \mid \neg \mid \cdots \\
\text{binop } &\coloneqq + \mid - \mid \times \mid \div \mid \cdots \\
\text{Call } F &\coloneqq f(e_1, \dots, e_n) \\
\text{Return } R &\coloneqq e \\
\text{Store } M &\coloneqq (e_1, e_2) \\
\text{Cond } C &\coloneqq e
\end{aligned}
$$

Figure 5: Syntax of feature

features while keeping the unique part; To find the most important features, we rank the features based on their importance.

3 APPROACH

The overall framework of React is illustrated in Figure 4. We compile the source code to two IR codes corresponding to the commit before and after the patch, referred to as the *vulnerable IR* and *patched IR*, respectively. Then we lift the target binary to LLVM IR, referred to as *target IR*. Based on the generated IR codes, we execute the IR code at symbolic level and collect the features in **feature generation**. Then we **refine** the features from *vulnerable IR* and *patched IR* to make them more comparable, and **rank** them to focus on the most important ones. Finally, we **match** the ranked features with *target IR* to determine whether the binary is patched. During feature matching, an SMT solver is used to prove the semantic equivalence between the features.

3.1 Feature Generation

Feature generation serves as the foundation of our approach. All subsequent steps are based on the features generated by the symbolic executor. In this paper, we consider four types of features, i.e., *return value*, *condition*, *memory store*, and *function call*. The syntax of the feature is shown in Figure 5. The expression can be composed of binary and unary operations, a function call, or a constant. $\textbf{arg}(i)$ and $\textbf{alloc}(i)$ represent the argument of the function and the locations of the local variables, respectively, where i is an integer. $\textbf{mem}(e)$ represents the memory location pointed by the value of e.

Return and condition features are represented as single expressions, memory store is represented as the value and address pair, and function call features are represented as the function name and arguments. It is worth noting that for the memory store feature, we only consider the address of an expression consisting of a function argument or global variable. Store operations on local variables are ignored, since they are not visible to the caller.

To extract the features, we symbolically execute the IR code. We initialize the state with the function arguments as **arg**. The basic block is executed in depth-first order, and the features are collected in the execution trace. For each state, we maintain the data flow by tracking the local variables and the memory store. Ordinary arithmetic operations are executed as symbolic operations, resulting in a symbolic expression. When encountering a conditional branch, we record the symbolic expression as a condition feature. When the basic block ends with a return instruction, we record the return value as a return feature. When loading a value from memory, we record the address and the value in the memory store feature. If the address a does not exist in current memory, we lazily initialize the value and return it with a symbolic value $\textbf{mem}(a)$. All function

calls are skipped and a symbolic call value is assigned to the variable representing the return value. In the meantime, the function call name with the symbolic parameters is recorded in the feature set. As a result, we can obtain a total of three feature sets V, P, T for the vulnerable IR, patched IR, and target IR, respectively.

3.2 Refinement & Ranking

After feature generation, we obtain the feature sets V, P for the vulnerable and patched IR, respectively. We construct the initial feature sets v, p by removing the features in both V and P, i.e., $(v, p) = V \triangle P$. The whole refinement and ranking process is shown in Algorithm 1. The input of the algorithm is the four feature sets mentioned above and the output is the ranked feature list. The features of vulnerable and patched IR are refined by the features P and V (line 1 and line 2). We construct a bipartite graph G with the vertex consisting of the refined features and the edge weight as the similarity between the features. We pair the features according to the maximum weight matching in the bipartite graph (line 6). The features that are not matched are appended to the list (line 8 and line 11). Finally, the features are ranked according to the importance of the features (line 13).

Algorithm 1: Refinement & Ranking

Input: Initial feature sets, v, p, V, P
Output: Ranked feature list, v', p'

1 $v = \{\text{refine}(f, P) \mid f \in v\}$; /* Section 3.2.1 */
2 $p = \{\text{refine}(f, V) \mid f \in p\}$;
3 $V \leftarrow v \bigcup p$; /* Construct the vertex set */
4 $E \leftarrow \{(a, b, \mathcal{S}(a, b)) \mid a \in V, b \in P\}$; /* Def 3.3 */
5 $G \leftarrow (V, E)$;
6 $list(v', p') \leftarrow \text{MaxWeightMatching}(G)$;
7 **foreach** $v \in v \setminus v'$ **do**
8 $\quad list.append((v, None))$;
9 **end**
10 **foreach** $p \in p \setminus p'$ **do**
11 $\quad list.append((None, p))$;
12 **end**
13 rank($list$) ; /* Section 3.2.2 */
14 **return** $list$

3.2.1 *Refinement.* The goal of feature refinement is to generalize the features to make them easier to match. **The intuition of refinement is that, while keeping the unique characteristics of the feature, it should be as simple as possible.** We can remove some parts of the feature as long as they are unique in the context of other parts of the feature.

To achieve this goal, we first expand the syntax of the expression to include $\top$ as follows:

$$\begin{aligned} \text{Expr } e \coloneqq\ & e \text{ binop } e \mid \text{unop } e \\ & \mid \text{mem}(e) \mid \text{const} \mid \text{arg}(i) \\ & \mid \top \end{aligned}$$

Symbol $\top$ can be considered as a wildcard that can match any value. The process of removing some part of the feature is to replace

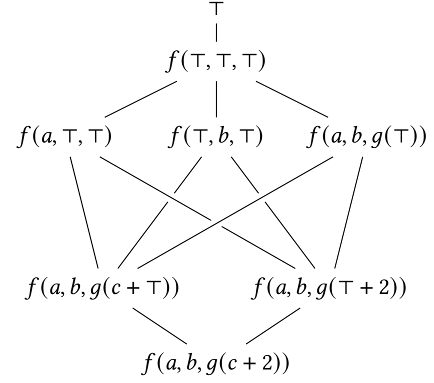

Figure 6: Lattice of the feature $f(a, b, call(c+2))$. We only show a subset of the lattice for simplicity.

the part with $\top$. The whole feature sets from the removed part can constitute a lattice $(\mathcal{L}, \sqsubseteq)$, while partial order $\sqsupseteq$ can be defined recursively based on $x \sqsubseteq \top$:

Definition 3.1 (Partial Order).

$$\begin{gathered} l \sqsubseteq \top \\ m \sqsubset m' \Rightarrow \text{mem}(p) \sqsubseteq \text{mem}(p') \\ e \sqsubseteq e' \Rightarrow \text{unop}(e) \sqsubseteq \text{unop}(e') \\ l_1 \sqsubseteq l_1' \text{ and } l_2 \sqsubseteq l_2' \Rightarrow \text{binop}(l_1, l_2) \sqsubseteq \text{binop}(l_1', l_2') \\ l_i \sqsubseteq l_i' \text{ for } i = 1, \ldots, n \Rightarrow \text{call}(l_1, \ldots, l_n) \sqsubseteq \text{call}(l_1', \ldots, l_n') \end{gathered}$$

Take a specfic feature $f(a, b, g(c + 2))$ for example, where a, b, c are arguments and g, h are function calls, the lattice is shown in Figure 6. The higher the element in the lattice, the less information it contains. The most general and useless element is $\top$, and the most specific element is the original feature. Refinement is to continuously add more information to the value until it is unique compared with the target feature sets. From the top element $\top$ of the lattice, there are many paths to the bottom element, i.e., the original feature. We list some paths as follows:

$$\begin{gathered} \top \sqsupseteq f(\top, \top, \top) \sqsupseteq f(a, \top, \top) \sqsupseteq f(a, b, g(c + \top)) \sqsupseteq f(a, b, g(c + 2)) \\ \top \sqsupseteq f(\top, \top, \top) \sqsupseteq f(\top, b, \top) \sqsupseteq f(a, b, g(\top + 2) \sqsupseteq f(a, b, g(c + 2)) \\ \vdots \\ \top \sqsupseteq f(\top, \top, \top) \sqsupseteq f(a, b, g(\top)) \sqsupseteq f(a, b, g(c + \top)) \sqsupseteq f(a, b, g(c + 2)) \end{gathered}$$

In general, for an arbitrary lattice, the possible paths are as follows:

$$\top = f_0 \sqsupseteq f_{11} \sqsupseteq f_{12} \sqsupseteq \cdots \sqsupseteq f_{1n} = f$$
$$\top = f_0 \sqsupseteq f_{21} \sqsupseteq f_{22} \sqsupseteq \cdots \sqsupseteq f_{2n} = f$$
$$\vdots$$
$$\top = f_0 \sqsupseteq f_{m1} \sqsupseteq f_{m2} \sqsupseteq \cdots \sqsupseteq f_{mn} = f$$
$$f_{ij} \in \mathcal{L}$$

The whole refinement process under context F is to iterate over the paths, from the most general feature $\top$ to the most specific feature f until we find the smallest i satisfying:

$$f_{i-1} \hat{\in} F \wedge f_i \hat{\notin} F$$

where the order can be determined by deep-first or bread-first search along the lattice. The meaning of $\hat{\in}$ here is similar to the common operator $\in$, except that we consider $\top$ to be equal to any value. Note that the element and paths are finite and $f \notin F$ (we have already removed the feature if it is in F), so we can always find such i. In addition, the result of the refinement is not unique.

Example 3.2. Returning to our patch example for motivation, `bn_wexpand(lambda, group_top + 2)` is refined to $b(\top, \top + 2)$ by the following path:

$$\top \longrightarrow b(\top, \top) \longrightarrow b(\top, \top + 2)$$

where we abbreviate the function name `bn_wexpand` to b. Noting that `group_top` and `lambda` are refined to $\top$ since expression `+ 2` is the unique part of the feature. The refinement process cannot stop at $b(\top, \top)$, as it exists in the patched IR features. Similarly, we refine the formula for vulnerable IR to $b(\top, \top + 1)$.

3.2.2 Ranking. After refinement, we rank the features based on the importance of the features. The features in vulnerable and patched IR are seen as two groups of vertex in a bipartite graph. The definition of weight for an edge is given by the following definition, which allows items with similar features and characteristics to be paired together.

Definition 3.3 (Similarity).

$$\mathcal{S}(\top, x) = \mathcal{S}(x, \top) = 1,$$
$$\mathcal{S}(\text{const}(v), \text{const}(v')) = 1, \text{ if } v = v'$$
$$\mathcal{S}(\text{mem}(l), \text{mem}(r)) = 1 + \mathcal{S}(l, r)$$
$$\mathcal{S}(\text{unop}(l), \text{unop}(r)) = 1 + \mathcal{S}(l, r)$$
$$\mathcal{S}(\text{binop}(l_1, l_2), \text{binop}(r_1, r_2)) = 1 + \mathcal{S}(l_1, r_1) + \mathcal{S}(l_2, r_2)$$
$$\mathcal{S}(\text{call}(l_1, \ldots, l_n), \text{call}(r_1, \ldots, r_n)) = 1 + \sum_{i=1}^{n} \mathcal{S}(l_i, r_i)$$
$$\mathcal{S}(l, r) = 0, \text{ otherwise}$$

As a result, we can construct a bipartite graph $G = (V, E)$ with the vertex consisting of the refined features and the edge weight as the similarity between the features. The goal is to find the matching that maximizes the sum of the edge weight, and it is indeed a bipartite graph matching problem. We use the Hungarian algorithm [27] to solve the problem of combining maximum weights.

After obtaining the results for the graph matching, we are ready to rank the features. Matched features are preferred because we can detect them in both vulnerable and patched IR. For the same group (matched or unmatched) features, a higher complexity means that it is of lower importance because they are more unlikely to be matched for the target features. The complexity of the feature is defined recursively on the basis of the complexity of the subexpression, which can be formally defined as follows.

Definition 3.4 (Complexity).

$$C(\top) = 0,$$
$$C(\text{const}(v)) = 1$$
$$C(\text{unop}(e)) = 1 + C(e),$$
$$C(\text{binop}(e_1, e_2)) = 1 + C(e_1) + C(e_2),$$
$$C(\text{mem}(e)) = 1 + C(e),$$
$$C(\text{call}(f, e_1, \ldots, e_n)) = 1 + \sum_{i=1}^{n} C(e_i),$$

3.3 Feature Matching

Based on the ranked feature lists, we match them with target feature sets T one by one. For every pair of features v, p, there are a total of four cases:

- v, p both exist or both do not exist in the target feature set, we consider the next feature pair.
- v exists in the target feature set, but p does not, we consider the target as vulnerable.
- p exists in the target feature set, but v does not, we consider the target as patched.

Since the patch presence test is security critical, we make a conservative decision when none of the feature pairs matches, and we consider the target vulnerable. The only exception is when the patch file is a pure deletion of code, which means that there should be no feature matched in the patched target. In this case, we consider the target patched.

In the feature matching process, the $\top$ generated in the refinement is considered as a wildcard that can match any value. If the feature cannot match any of the features in the target feature list, we use the SMT solver [17] to prove the equivalence of the features. For each pair of features a, b, we transform them into SMT constraints $a \neq b$ and check the satisfiability. If the constraints are unsatisfiable, we consider the pair as match; otherwise, we regard them as mismatch. The argument and allocation of functions are modeled as variables. The memory load and call to functions are modeled as global functions.

3.4 Implementation

We implement React in Rust[1]. We compile the source code for LLVM IR[2] with optimization levels of O0 and O3 with Clang 14.0[3] to combat the inline function caused by optimization. We use RetDec

[1] https://www.rust-lang.org/
[2] https://llvm.org/
[3] https://clang.llvm.org/

5.0[4] to lift the binary to LLVM IR. Z3[5] is used for SMT solving in feature matching. We also transform the expressions into the canonical form following KLEE [6], which makes it easier to match.

In feature generation, for the *phi* instruction in LLVM IR, we allow the corresponding basic block to be executed up to the number of incoming blocks of the *phi* instruction to meet all possible incoming values. We transform the index calculation in the instruction *gep* into the calculation of the offset of the structure so that it can be matched to the IR lifted from the binary. When encountering arguments that are pointers to an integer or string (i.e., array of unsigned char), we extract the point-to integer or string and record it as the argument in the function call feature.

We need to limit the power of refinement because of the uncertainty of the target binary. Original refinement is good enough if the target binary always complies with the commit before or after the patch. However, the developer can modify the code and the target binary might be compiled from newer code so that refined features may match irrelevant features. For example, feature $f((a+2), b)$ might be refined to $f((\top + \top), \top)$, which is unique in reference IRs. However, the feature matches $f(c+d, g(e))$ from a new function call added in the target binary, which is unexpected. Therefore, we limit the refinement by (1) one level replacement in call feature and (2) const value and function call name can not be replaced by $\top$. These rules limit refinement and improve accuracy.

4 EVALUATION

To evaluate the effectiveness of our approach, we conducted experiments following three research questions.

- **RQ.1** How effective is our approach at patch presence test compared to the state-of-the-art baselines
- **RQ.2** Does our refinement, ranking, and SMT solver technique improve the performance of our approach?
- **RQ.3** How efficient is REACT?

4.1 Experimental Setup

4.1.1 Baselines. We compare REACT with two baselines:

BinXray [40]. A state-of-the-art patch presence test tool without the presence of source code. BinXray is a similarity-based approach that uses basic block mapping to extract the signature of a patch by comparing vulnerable and patched binary programs.

PS3 [42]. A state-of-the-art patch presence test approach designed to combat compiler optimization. PS3 is based on the symbolic signature of the function, which is extracted from the binary.

We do not consider FIBER [44] and PDiff [24] in our evaluation because the features they extracted are kernel code specific, which is unsuitable for our dataset. The patch presence test on Java bytecode is also not our baseline.

4.1.2 Dataset. We conducted experiments on the dataset proposed in PS3 [42]. The dataset consists of four projects, i.e., OpenSSL, FFmpeg, LibXML2, and tcpdump. It contains 62 CVEs and 3,631 test cases compiled from different compilers, including GCC and Clang, with varying levels of optimization, from O0 to O3. We exclude one CVE from the dataset because PS3 cannot find the correctly vulnerable function in the binary. Each test case aims to check for the presence of a specific patch in a specific binary. Furthermore, we extend the original dataset used in PS3 [42] by adding CVEs from the same projects and constructing test cases following PS3. In total, our dataset contains 70 CVEs and 4,156 test cases. We list the statistics of our dataset in Table 1 and highlight the number of additional test cases.

As our approach is based on LLVM IR, we lift the binary to LLVM IR using RetDec for each binary in the original data set, which constructs *target IRs*. For each CVE, we compile the source code before and after the patch commit to emit the LLVM IR directly, which constructs *vulnerable IRs* and *patched IRs*. We manually fix the constant value in lifted LLVM IR, which is obviously incorrect corresponding to the binary, and report the mislifting issue to the RetDec team.

4.1.3 Metrics. Following previous work [31, 42] in the patch presence test, we use precision (P), recall (R), and F1 score (F1) as metrics to evaluate our approach with other baselines.

Precision Let R_R represent the count of truly vulnerable binaries detected, and R_I represent the count of patched binaries mistakenly identified as vulnerable. Precision (P) can then be formulated as:

$$P = \frac{R_R}{R_R + R_I}$$

Recall indicates the proportion of truly vulnerable binaries detected R_R out of all vulnerable binaries present R_N. Recall (R) can be expressed as:

$$R = \frac{R_R}{R_N}$$

F1 Score assesses the overall effectiveness of the test. With precision (P) and recall (R) defined, the F1 score is computed as follows:

$$F1 = \frac{2PR}{P+R}$$

4.2 Results

4.2.1 Effectiveness of our approach vs. baselines. To answer RQ1, we compare the effectiveness of our approach with the baselines in terms of precision, recall, and the F1 score. The results are shown in Table 2.

REACT achieves the highest precision, recall, and F1 score among the three approaches. The F1 score of our approach is 12% and 39% higher than PS3 and BinXray, respectively. BinXray achieves an F1 score of 0.67, which is lower than PS3 and our approach. It cannot identify the patch when compiler optimization is applied; thus, it decides that the binary is vulnerable by default. Compared to REACT, PS3 only considers forward instructions. In cases where the root of the vulnerability lies backward, as shown in Figure 3a, PS3 is unable to identify the differences.

To further compare our approach with PS3, we list the performance of each compiler option in Table 3. We have the following findings:

(1) Our approach outperforms PS3 in the *all* compiler options, while the lowest F1 score of our approach is as high as the highest F1 score of PS3.

[4] https://github.com/avast/retdec
[5] https://github.com/Z3Prover/z3

Table 1: Statistics of our dataset

Project	#CVE	Clang & O0		GCC & O0		Clang & O1		GCC & O1		Clang & O2		GCC & O2		Clang & O3		GCC & O3		#Test
		#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	#V	#P	
OpenSSL	27	164	200	170	202	155	156	163	173	155	155	151	135	155	171	151	135	2,591 (+358)
FFmpeg	30	85	127	85	127	61	88	66	90	61	88	53	78	61	78	53	78	1,289 (+166)
Tcpdump	11	33	11	33	11	21	7	24	8	21	7	24	8	21	7	21	6	263
LibXml2	2	2	0	4	0	2	0	1	0	1	0	1	0	1	0	1	0	13 (+1)
Total	70 (+8)	284	338	292	340	239	251	254	271	238	250	229	221	238	266	226	219	4,156 (+525)

#P represents the number of patched pairs while *#V* represents vulnerable pairs.

Table 2: Effectiveness of our approach vs. baselines

Approach	Precision	Recall	F1
BinXray	0.51	0.96	0.67
PS3	0.75	0.93	0.83
REACT	0.88	0.98	0.93

(2) The performance on the GCC compiler is better than that on the Clang compiler for both approaches. It is reasonable for PS3 because the reference binary is compiled by GCC. For our approach, the reason may be that the RetDec is more suitable to lift the GCC compiled binary.
(3) The F1 score of PS3 ranges from 0.79 to 0.90, while the F1 score of REACT ranges from 0.90 to 0.95. Although the F1 score of our approach and PS3 drops as the optimization level increases, our approach is more robust to compiler optimization.

> **Answer to RQ1:** REACT can effectively identify the patch with a F1 score of 0.93. It outperforms BinXray and PS3 by 39% and 12%, respectively. In addition, our approach outperforms PS3 in all compiler options.

4.2.2 Abalation Study. To investigate the contribution of feature refinement, ranking, and SMT solver in the effectiveness of our approach, we performed an ablation study. We create the following variants:

- -SMT: SMT solver is removed. We only compare the features at syntactic level.
- -RANK: Feature ranking is removed. We use the original feature set to match the features without matching them together and ranking them.
- -REFINEMENT: We skip the feature refinement step and directly match the features.

We evaluated the four variants on the same dataset, as described in Section 4.1.2. The results are illustrated in Table 4. F1 score decreases by 16%, 1.1%, 10% when the SMT solver, ranking step, and refinement are removed.

❶ The reason why the SMT solver is important in our approach is that it can check the equivalence of features at a semantic level. For example, it is a common case that the compiler optimizes the code $a < 4$ to $a \leq 3$ when a is an integer. The syntactic of the two expressions is different, but the semantic features are the same. With the help of the SMT solver, REACT can correctly detect the presence of a patch.

❷ Ranking does not contribute much to the effectiveness of our approach. This is because most patches are small [29] and there are few features to extract, and most test cases do not benefit from feature ranking directly. When encountering a situation like that in Figure 3b, ranking can make a difference and improve results.

❸ The refinement step can improve the F1 score by 10%. As we mentioned in Section 3, the refinement step can reduce the negative impact of local variables and complex arithmetic operations. For example, in Figure 7, the refined feature for `sigalg != NULL` in line 6 is `@tls1_lookup_sigalg(T) = 0`, where the function call `tls1_lookup_sigalg` is from line 3. In contrast, the original feature involves complex computations of offset and array indexes in its arguments. In IR lifted from the target binary, RetDec fails to recover the correct offset. As a result, REACT fails to detect the patch without the refine technique.

```
1 @@ -2130,7 +2130,7 @@ static int tls1_check_sig_alg(SSL *s, X509 *x, int
  ↪   default_nid)
2           sigalg = use_pc_sigalgs
3                   ? tls1_lookup_sigalg(s->s3->tmp.peer_cert_sigalgs[i])
4                   : s->shared_sigalgs[i];
5 -         if (sig_nid == sigalg->sigandhash)
6 +         if (sigalg != NULL && sig_nid == sigalg->sigandhash)
7               return 1;
8       }
```

Figure 7: Patch of CVE-2020-1967 [11]

> **Answer to RQ2:** Our approach benefits from the SMT solver and feature refinement, which can improve the F1 score by 16% and 10%, respectively.

4.2.3 Efficiency of our approach. To answer RQ3, we compare the time cost of our approach with BinXray and PS3. We list the average, minimum, and maximum times for one testcase in Table 5. Our approach takes an average of 0.025 seconds to test the patch presence of binaries, which is the fastest among the three approaches. The reason may be that our approach is implemented in Rust, which is more efficient than the Python implementation of PS3 and BinXray. In addition to the programming language, the ranking step in our approach can reduce the number of features to match, which saves time. In contrast, PS3 compares all extracted features and decides the presence of the patch by voting, which is

Table 3: Results on different compiler options

Compiler Option Combination	PS3			React		
	Precision	Recall	F1	Precision	Recall	F1
GCC & O0	0.88	0.91	0.90	0.92	0.98	0.95
GCC & O1	0.77	0.96	0.86	0.89	1	0.94
GCC & O2	0.77	0.96	0.85	0.87	1	0.93
GCC & O3	0.77	0.96	0.85	0.87	1	0.93
Clang & O0	0.72	0.92	0.81	0.91	0.98	0.94
Clang & O1	0.74	0.92	0.82	0.86	0.96	0.91
Clang & O2	0.70	0.92	0.79	0.85	0.96	0.90
Clang & O3	0.69	0.92	0.79	0.85	0.96	0.90

Table 4: Ablation study

Approach	Precision	Recall	F1
React	0.87	0.98	0.93
-Smt	0.66	0.99	0.80 (↓16%)
-Rank	0.88	0.95	0.92 (↓1.1%)
-Refinement	0.79	0.95	0.86 (↓10%)

Table 5: Time cost of our approach vs. baselines (seconds)

Approach	Max	Min	Average
BinXray	0.1	-	0.06
PS3	23	0.14	3
React	0.317	0.003	0.025

time-consuming. BinXray does not utilize an SMT solver and only performs similarity-based matching, so it is much faster than PS3.

> **Answer to RQ3:** React can test the patch presence of binaries with an average of 25 milliseconds, which is 100x faster than PS3 and 2x faster than BinXray.

5 DISCUSSION

5.1 Why React fails

We manually inspect the test case where React fails and find out why React fails to detect the patch. The reasons as follows:

- **Mislifting:** The lifted IR code is not semantically equivalent to the original binary, making it impossible to detect the patch based on the IR code [2].
- **Complier optimization:** The key semantic information extracted by our approach is lost during optimization.
- **Lack of context:** The feature extracted is only unique under the context of constraints, but our approach does not consider them and takes them as duplicate features.

5.1.1 Mislifting. Our approach is based on binary lifting tools to obtain the intermediate representation of the binary, so miscompilation of the lifted IR code may unavoidably affect the result of our approach. For example, in the CVE-2018-14468 patch in Figure 8a, RetDec cannot recover the string `"(invalid length)"` on line 6 in the lifted IR code, which exits the original binary. As a result, the lifted IR code is not semantically equivalent to the original binary, making it impossible to detect the patch based on the lifted IR code. In addition, control flow is sometimes misrecovered. As the lifting tool improves [25, 43, 45], we believe that the problem of mislifting can be mitigated over time.

```
1 @@ -493,6 +493,11 @@ mfr_print(netdissect_options *ndo,
2      switch (ie_type) {
3      case MFR_CTRL_IE_MAGIC_NUM:
4 +        /* FRF.16.1 Section 3.4.3 Magic Number Information Element */
5 +        if (ie_len != 4) {
6 +            ND_PRINT((ndo, "(invalid length)"));
7 +            break;
8 +        }
9          ND_PRINT((ndo, "0x%08x", EXTRACT_32BITS(tptr)));
```

(a) Patch of CVE-2018-14468 [10]

```
1 @@ -257,6 +257,9 @@ int ossl_rsaz_mod_exp_avx512_x2(BN_ULONG *res1,
2      from_words52(res1, factor_size, rr1_red);
3      from_words52(res2, factor_size, rr2_red);
4
5 +    /* bn_reduce_once_in_place expects number of BN_ULONG, not bit size */
6 +    factor_size /= sizeof(BN_ULONG) * 8;
7      bn_reduce_once_in_place(res1, /*carry=*/0, m1, storage, factor_size);
8      bn_reduce_once_in_place(res2, /*carry=*/0, m2, storage, factor_size);
```

(b) Patch of CVE-2022-2274 [16]

```
1 @@ -3272,6 +3272,7 @@ static int open_files(OptionGroupList *l, const char
  ↪  *inout,
2      if (ret < 0) {
3          av_log(NULL, AV_LOG_ERROR, "Error parsing options for %s file "
4                 "%s.\n", inout, g->arg);
5 +          uninit_options(&o);
6          return ret;
7      }
8      av_log(NULL, AV_LOG_DEBUG, "Opening an %s file: %s.\n", inout, g->arg);
9      ret = open_file(&o, g->arg);
10     uninit_options(&o);
```

(c) Patch of CVE-2020-20451 [12]

Figure 8: Case study examples

5.1.2 Compiler Optimization. Though we consider the compiler's optimization level, the resulting IR may lose some semantic information. For example, the patch in Figure 8b changes the fifth parameter of the call function `bn_reduce_once_in_place`. Our

algorithm should detect the patch, as REACT can find that the parameter is from the original value of `factor_size` to the modified value of `factor_size / sizeof(BN_ULONG) * 8`. However, the function call in the lifted IR does not contain five parameters, since the compiler recognizes that the fifth parameter must be a constant value and optimizes it. This kind of optimization based on program analysis is difficult to detect using our approach.

5.1.3 Lack of Context. In the case where features generated in vulnerable and patched IR code are unique under the context of constraints, our approach fails to detect the patch. For example, in Figure 8c, the function call `uninit_options` is not unique because there exists the same function call with the same parameter on line 10. REACT cannot distinguish the call to the function on lines 5 and 10, which leads to the failure to detect the patch. In our future work, we plan to consider the context to improve our approach's accuracy.

5.2 Execution Path

In feature generation, we perform a basic block execution and permit multiple times on *phi* block. However, it is still not enough to obtain all possible features. In an ideal case, we should iterate all paths that do not contain a basic block more than two times from the entry of the function to the exit so that every possible useful feature can be collected. There can exist exponential paths that need to be executed, which is not feasible in practice. Thus, we have to make a trade-off by limiting the paths in our approach. The common basic block is only allowed to be executed once, whereas the basic block starting by the *phi* instruction can be executed up to the number of incoming blocks.

5.3 Threats to Validity

As mentioned in Section 5.1.1, the miscompilation of the IR code lifted threatens our approach's internal validity. The symbolic execution in our approach is not complete and only considers certain program paths, which may lead to the loss of some features. Our approach cannot identify the patches if the target binary's symbol table does not contain the target function.

We only conduct the experiments for binaries compiled for the Intel X86 64-bit system, though our approach is intended to support any architecture, even cross-architecture. We do not consider the preprocessing time in RQ3, as BinXray must work on top of IDA-Pro [32], PS3 needs to compile the reference binary, and our approach must use lifted IR codes. It may affect the real-world performance of our approach. In addition, the precision of the information on the affected version of the vulnerable software in the National Vulnerability Database (NVD) is often inconsistent [4], potentially affecting the accuracy of our dataset.

6 RELATED WORK

In this section, we briefly review the work related to the patch presence test. The summary is shown in Table 6.

Table 6: Summary of related work

Work	Approach	Year
Binary to Binary (C/C++)		
FIBER [44]	feature matching	2018
PDiff [24]	feature similarity	2020
BinXray [40]	basic block mapping	2020
Osprey [34]	lightweight flow slices	2021
PatchDiscovery [39]	key basic blocks	2023
PS3 [42]	SMT solver	2024
Source to Binary (Java)		
BSCOUT [8]	first work in Java	2020
PHunter [38]	against obfuscation	2023
PPT4J [31]	find-grained semantics	2024
IR to IR (LLVM IR)		
REACT		

6.1 C/C++ Binary

Zhang et al. [44] are the first to publish patch presence testing, introducing the "patch presence test" and proposing FIBER for kernel customization scenarios. FIBER uses angr [33] and VEX [30] to generate fine-grained binary signatures from patch files and debug information, reflecting changes made by patch modifications. These signatures determine whether the target binary has been patched. Following FIBER, Jiang et al. [24] introduce PDiff, a system that uses images from the downstream kernel for highly reliable patch presence testing. PDiff relies on semantic similarity in patch checking, offering high fault tolerance for code changes. Compared to FIBER, PDiff achieves high-precision testing with an exceptionally low miss rate in experiments. Considering the high overhead of symbolic execution technology in FIBER [3], Sun et al. [34] propose Osprey, which utilizes lightweight copy propagation and data flow slices instead of symbolic execution. This modification enables Osprey to accelerate the testing process by more than 10 times without significantly compromising accuracy, maintaining a performance of over 90% in their experiments.

PS3 [42] uses the SMT solver to verify the equivalence of signatures at the semantic level. PS3 is designed to directly utilize signatures for patch presence testing, which can achieve high accuracy in the presence of compiler optimization. In addition, PS3 uses extracted signatures for voting rather than ranking them exploited in our approach. Our feature generation is inspired by BLEX [18], while we use LLVM IR as input language.

The studies mentioned above are based on the existence of source code, while Xu et al. [40] propose BinXray. It does not presume the existence of the source code and patch commit. BinXray derives patch signatures through a comparison between a vulnerable code snippet and its patched counterpart using a basic block mapping algorithm. In 2023, following BinXray, Xu et al. [39] propose a novel algorithm to capture the signatures of key basic blocks, which outperforms BinXray.

6.2 Java Bytecode

In addition to the detection of executable files compiled in the mainstream C/C++ languages mentioned above, there are also some work that focus on Java bytecodes. Compared with executable binary, Java bytecodes contain much richer semantic information, and optimization is performed at runtime (known as JIT) rather than compile time. Hence, the accuracy of the patch presence test for Java bytecode is usually higher than that of binaries.

Dai et al. [8] propose BSCOUT, which is the first tool that can check the existence of patches in Java files. It checks fine-grained patch semantics in the entire target executable. Pan et al. [31] propose PPT4J, which uses type analysis and control flow information to generate find-grained patch semantics, which outperforms BSCOUT. To counteract the challenge introduced by code obfuscation in Android applications, Xie et al. [38] propose PHunter to address the challenge by combining coarse-grained search for the target code and fine-grained patch semantics.

Compared with all previous work which are essentially binary-to-binary or source-to-binary, our approach is the first to use LLVM IR as the input language. This means that our approach can detect not only binaries compiled from the C/C++ language, but also other programming languages compiled to LLVM IR, like Rust.

7 CONCLUSION

Detecting the presence of patches in binary code is a critical task in software security. Existing approaches are based on source code or binary code, which have limitations in terms of accuracy and efficiency. In this paper, we propose REACT, the first patch presence test approach based on binary lifting. We extract symbolic-level features from reference and target IR codes, then match the features to determine the presence of a patch. In addition, we propose a feature refinement method to improve the generality of the features. The experimental results show that REACT outperforms the current approaches by 0.93 in terms of the F1 score. In practice, REACT can efficiently detect the presence of patches in an average of 0.025 seconds per binary.

Our implementation and dataset are available at https://github.com/Qi-Zhan/React.

ACKNOWLEDGMENTS

This research is supported by Ningbo Natural Science Foundation (No. 2023J292).

REFERENCES

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[2] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3342195.3387550
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657
[4] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2352–2364. https://doi.org/10.1145/3510003.3510113
[5] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. 2022. Verified compilation and optimization of floating-point programs in cakeml. In *European Conference on Object-Oriented Programming (ECOOP 2022)*.
[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[7] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1126–1140. https://doi.org/10.1145/3453483.3454099
[8] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct whole patch presence test for java executables. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1147–1164.
[9] National Vulnerability Database. 2018. CVE-2018-0735. https://nvd.nist.gov/vuln/detail/CVE-2018-0735
[10] National Vulnerability Database. 2018. CVE-2018-14468. https://nvd.nist.gov/vuln/detail/CVE-2018-14468
[11] National Vulnerability Database. 2020. CVE-2020-1967. https://nvd.nist.gov/vuln/detail/CVE-2020-1967
[12] National Vulnerability Database. 2020. CVE-2020-20451. https://nvd.nist.gov/vuln/detail/CVE-2020-20451
[13] National Vulnerability Database. 2021. CVE-2019-1547. https://nvd.nist.gov/vuln/detail/CVE-2019-1547
[14] National Vulnerability Database. 2021. CVE-2021-23841. https://nvd.nist.gov/vuln/detail/CVE-2021-23841
[15] National Vulnerability Database. 2022. CVE-2022-1343. https://nvd.nist.gov/vuln/detail/CVE-2022-1343
[16] National Vulnerability Database. 2022. CVE-2022-2274. https://nvd.nist.gov/vuln/detail/CVE-2022-2274
[17] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[18] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 303–317.
[19] Alexis Engelke and Martin Schulz. 2020. Instrew: Leveraging LLVM for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 172–184.
[20] Apache Software Foundation. 2024. Apache HTTP Server Project. https://httpd.apache.org/
[21] DuckDB Foundation. 2024. DuckDB. https://duckdb.org/
[22] Jian Gao, Yu Jiang, Zhe Liu, Xin Yang, Cong Wang, Xun Jiao, Zijiang Yang, and Jiaguang Sun. 2021. Semantic Learning and Emulation Based Cross-Platform Binary Vulnerability Seeker. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2575–2589. https://doi.org/10.1109/TSE.2019.2956932
[23] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480
[24] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1149–1163.
[25] Lukáš Korenčik. 2019. *Decompiling binaries into LLVM IR using McSema and Dyninst*. Ph. D. Dissertation. Masarykova univerzita, Fakulta informatiky.
[26] Jakub Křoustek, Peter Matula, and Petr Zemek. 2017. Retdec: An open-source machine-code decompiler. In *July 2018*.
[27] Harold W. Kuhn. 2010. *The Hungarian Method for the Assignment Problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 29–47. https://doi.org/10.1007/978-3-540-68279-0_2
[28] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.
[29] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2201–2215. https://doi.org/10.1145/3133956.3134072

[30] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[31] Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 225, 12 pages. https://doi.org/10.1145/3597503.3639231

[32] Hex Rays. 2023. IDA Pro. https://www.hex-rays.com/products/ida/

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 138–157.

[34] Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. 2021. Osprey: A fast and accurate patch presence test framework for binaries. *Computer Communications* 173 (2021), 95–106.

[35] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 66 (mar 2024), 33 pages. https://doi.org/10.1145/3632743

[36] Linus Torvalds. 2024. The Linux Kernel Archives. https://www.kernel.org

[37] Veracode. 2020. State of Software Security. https://www.veracode.com/sites/default/files/pdf/resources/reports/state-of-software-security-open-source-edition-veracode-report.pdf

[38] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 347–359.

[39] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, Zhaohui Zhou, Haijun Wang, and Ting Liu. 2023. PatchDiscovery: Patch Presence Test for Identifying Binary Vulnerabilities Based on Key Basic Blocks. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5279–5294. https://doi.org/10.1109/TSE.2023.3332732

[40] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387.

[41] S. Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). 213–218. https://doi.org/10.1145/3316482.3326354

[42] Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. 2024. PS3: Precise Patch Presence Test based on Semantic Symbolic Signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 167, 12 pages. https://doi.org/10.1145/3597503.3639134

[43] Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2024. SIRO: Empowering Version Compatibility in Intermediate Representations via Program Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 882–899. https://doi.org/10.1145/3620666.3651366

[44] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries.. In *USENIX Security Symposium*. 887–902.

[45] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 912–928. https://doi.org/10.1145/3620665.3640382