Patch Based Vulnerability Matching for Binary Programs

Yifei Xu*
School of Software Engineering
Xi'an Jiaotong University
China
xyf0921@stu.xjtu.edu.cn

Zhengzi Xu*
School of Computer Science and Engineering
Nanyang Technological University
Singapore
xu0002zi@e.ntu.edu.sg

Bihuan Chen
School of Computer Science
Fudan University
China
bhchen@fudan.edu.cn

Fu Song
School of Information Science and Technology
ShanghaiTech University
China
songfu@shanghaitech.edu.cn

Yang Liu
Nanyang Technological University
Singapore
Institute of Computing Innovation
Zhejiang University
China
yangliu@ntu.edu.sg

Ting Liu
School of Cyber Science and Engineering , MoE KLINNS
Xi'an Jiaotong University
China
tingliu@mail.xjtu.edu.cn

ABSTRACT

The binary-level function matching has been widely used to detect whether there are 1-day vulnerabilities in released programs. However, the high false positive is a challenge for current function matching solutions, since the vulnerable function is highly similar to its corresponding patched version. In this paper, the Binary X-Ray (BinXray), a patch based vulnerability matching approach, is proposed to identify the specific 1-day vulnerabilities in target programs accurately and effectively. In the preparing step, a basic block mapping algorithm is designed to extract the signature of a patch, by comparing the given vulnerable and patched programs. The signature is represented as a set of basic block traces. In the detection step, the patching semantics is applied to reduce irrelevant basic block traces to speed up the signature searching. The trace similarity is also designed to identify whether a target program is patched. In experiments, 12 real software projects related to 479 CVEs are collected. BinXray achieves 93.31% accuracy and the analysis time cost is only 296.17ms per function, outperforming the state-of-the-art works.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Security and privacy → Software reverse engineering**.

KEYWORDS

Vulnerability Matching, Patch Presence Identification, Binary Analysis, Security

*Both authors contributed equally to this research.

ACM Reference Format:
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 (ISSTA '20), July 18–22, 2020, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397361

1 INTRODUCTION

Vulnerability whose patch has been released is called as 1-day vulnerability. It would be exploited to attack the users who fail to adopt the latest security patches. It is one of the most serious and common security threats. The binary-level code matching has been considered as a good solution to detect 1-day vulnerabilities in released programs [14, 20, 43]. It compares the similarity between functions with known vulnerabilities and target functions in a given binary executable. If a target function is similar to a known vulnerable function, it will be predicated as vulnerable.

To improve the vulnerability detection capability, many works have been proposed to improve the binary-level code matching accuracy. DiscovRE [18] and CACompare [23] achieve function matching across architectures by lifting the binary instructions to a unified intermediate representation. BLEX [17] uses program execution to extract the semantic features to improve the matching accuracy. BinGo [11] and BinGo-E [45] combines syntactic, structural and semantic features to produce more accurate matching results. However, it is difficult for the current function matching solutions to differentiate vulnerable and patched functions, since patches usually introduce subtle changes to fix vulnerabilities [37]. The patched functions would be identified as vulnerable, resulting in high false positive rates in detecting vulnerabilities [34]. As a result, these works require security experts to manually analyze potential vulnerable functions to find the genuine ones, which is time-consuming.

It is not trivial to address this problem. On the one hand, approaches need to be tolerant enough to identify vulnerable functions even with the presence of vulnerability-irrelevant small code changes like function upgrades and compiler optimizations. On the other hand, approaches also need to be precise enough to filter out those already patched functions. Zhang and Qian [46] proposed an algorithm to determine whether a function has been patched or not. They extract syntactic and semantic changes from source code and build a "source code to binary" matching model. However, it needs to analyze the source code, and thus it is not applicable when the source code is not available. To the best of our knowledge, there lacks an effective and efficient approach for binary-level vulnerability matching with patch identification. We have summarized the following three properties for such an approach to be practical for real-world projects.

- **P1.** The approach needs to be accurate in identifying the patches in the target functions.
- **P2.** The approach needs to be scalable for large real-world programs.
- **P3.** The approach should use no information from the source code to work in closed source program binaries.

To fulfill these properties, we propose a patch based vulnerability matching approach, named as Binary X-Ray (BinXray). It can precisely differentiate patched functions from vulnerable functions in the binaries. It reduces the error rate by more than 30% compared to the state-of-the-art function matching tool, Bingo-E [45], with less time consumed. It is more accurate than the patch identification tool FIBER[46], without source code.

For P1. To accurately identify patched functions and detect real vulnerable functions, BinXray introduces two-step signature matching approach. First, to narrow the searching space, BinXray generates the function signature from the functions with known vulnerabilities, and uses the signatures to search for suspicious target functions in the binaries through matching. Second, it generates patch signatures by comparing the differences between the vulnerable functions and their patched versions. The patch signatures will be used to identify the patched functions from the suspicious target functions. To extract accurate patch signatures, we propose a structural basic block mapping algorithm to locate the changed and unchanged basic blocks between two functions. BinXray makes patch prediction based on the length sensitive similarity matching of the patch signature with the target function.

For P2. To improve the scalability, BinXray proposes patch signature extraction algorithm, which only captures essential parts of the patching semantics. Since most of the security patches only induce small changes within a few basic blocks in binary programs [29, 42, 47], BinXray locates the areas which only consist of the changes induced by patching vulnerabilities, and generates patch signatures based on them instead of generating signatures at the granularity of the whole function. This can also greatly improves the matching speed. Moreover, changes in other parts of the functions won't be included in signatures, which are irrelevant to vulnerabilities. Thus, they will not affect the predicting results, leading to a more robust prediction against the noises from other changes.

For P3. From signature generation to patch identification, BinXray performs all the analysis at binary level. Therefore, it does not need any source code information which makes it suitable for programs without source code, such as firmware or third-party libraries.

This work makes the following contributions.

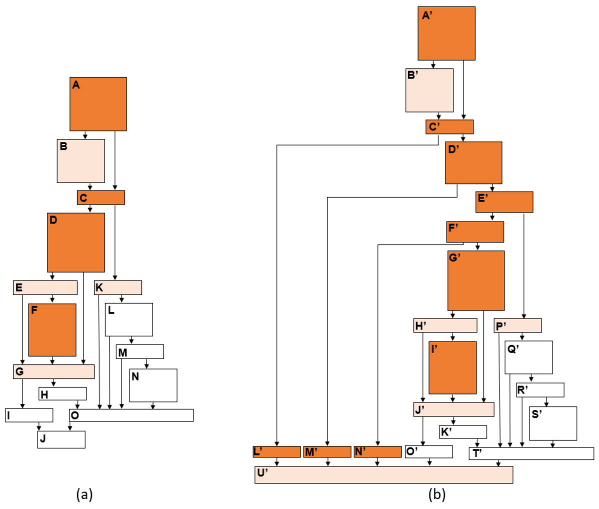


Figure 1: Running Example: control flow graphs of the function dtls1_process_heartbeat() with the HeartBleed Bug in OpenSSL versions 1.0.1f (a) and 1.0.1g (b)

- We propose a basic block mapping algorithm to accurately and efficiently map blocks in function differencing to generate patch signatures.
- A basic block boundary based algorithm is designed to locate the changes induced by patching. By reducing the size of patch signatures, it can improve the speed and scalability.
- We implement the prototype of BinXray, which can automatically extract patch signatures and accurately identify patched functions without any source code information. In experiments, 12 real software projects related to 479 CVEs are collected. BinXray achieves 93.31% accuracy in predicting the patch presence, and the analysis time cost is only 296.17ms per function, outperforming the state-of-the-art works.

2 OVERVIEW

In this section, we first introduce a motivating example, and then present the overview of our approach.

2.1 Motivating Example

We use the HeartBleed bug (`CVE-2014-0160` [1]) in OpenSSL as the running example. It occurs in `dtls1_process_heartbeat()` function. Figure 1(a) shows the control flow graph (CFG) of the vulnerable function in version `1.0.1f` and Figure 1(b) shows the CFG of the patched function in version `1.0.1g`. The patch adds a check in the source code which results in six new basic blocks in binary, named E', F', G', L', M' and N' in Figure 1(b). In addition, due to patching, the instructions in another four basic blocks, named A', C', D' and I' in Figure 1(b), are also slightly changed.

Given an unknown binary executable which may contain this function, we want to know whether the HeartBleed bug exists or not. If a traditional function matching approach is adopted and the vulnerable function in Figure 1(a) is used as the matching target, the corresponding function in the unknown binary may be matched. For instance, the patched function in Figure 1(b) can be matched,

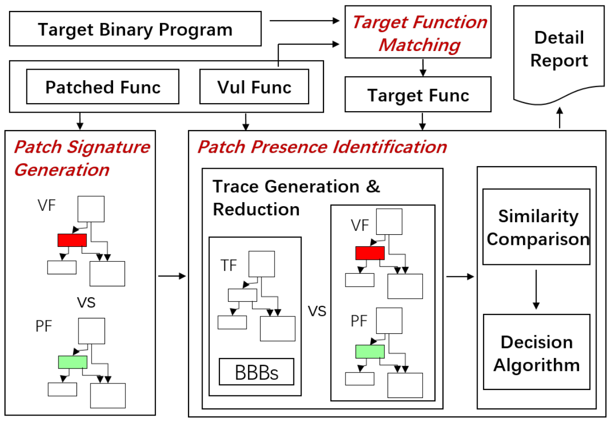


Figure 2: Overview of BinXray

since they share a large number of common blocks B', H', P', J', K', O', Q', R', S', T' and U' with the same structure. However, it is unclear whether the matched function has been patched or not, since the two functions have a high degree of similarity, so tedious manual examination is usually necessary to differentiate genuine and spurious one.

BinXray is designed to address this problem. Its goal is to differentiate patched functions from vulnerable functions by identifying patch presences so that vulnerable functions can be identified with a low false positive rate.

2.2 Approach Overview

Figure 2 shows the overview of our approach, named BinXray. Taking the binary code of a vulnerable function (VF), a patched version of the same function, called patched function (PF), and a target program as inputs, the goal of BinXray is to effectively and efficiently check whether the target program has any functions (TF) that are similar to the vulnerable function but have not been patched yet. For each CVE, VF is the function before the patch commit. PF is the function after it. By diffing VF and PF, BinXray can generate the patch signature. TF is detected by the function matching algorithm using VF as the signature in the target binary. The core components of BinXray are: target function matching (Section 3.1), patch signature generation (Section 3.2), and patch presence identification (Section 3.3).

Terms and acronyms. For convenient reference, we summarize the frequently used terms and their acronyms in Table 1.

Target function matching. BinXray generates lightweight function signatures and leverages function matching algorithm to quickly identify target functions (TFs) that are similar to the VF. Note that these TFs may not have been patched. The matching algorithm uses syntactic and structural information of VF, as the function signature, to identify TFs. Hence it is scalable and accurate enough to identify TFs.

Patch signature generation. BinXray automatically generates binary level patch signatures from the normalized binary code of the given VF and PF. Instead of incorporating the entire function into the signature, it first creates a mapping between the basic blocks (BBs) of two functions, from which BinXray identifies Changed Basic Blocks (CBBs) and computes two sets of valid traces (VTs): one set T_1 from the CBBs of the VF and one set T_2 from the CBBs of the PF. The two trace sets (T_1, T_2) are regarded as a patch signature.

Table 1: Terms and Acronyms

Term	Acronym	Description
Vulnerable Function	VF	A function that contains a vulnerability
Patched Function	PF	A function whose vulnerability has been patched
Target Function	TF	A function to be checked whether the vulnerability has been patched or not
Basic Block	BB	A sequence of consecutive instructions without any branching
Changed Basic Block	CBB	A block that has been changed, added, or deleted in the differences between a VF and its PF
Boundary Basic Block	BBB	A block that is the parent or the child of a CBB, or the root or leave of a CFG
Trace	-	A sequence of consecutive BBs without any loops
Valid Trace	VT	A sequence of consecutive basic blocks that starts and ends with some BBBs, crosses at least one CBB, without any loops

Patch presence identification. BinXray determines whether each identified TF has been patched or not, by matching it with the patch signature. If the TF is more close to VF than PF, it is considered to be vulnerable. Otherwise, it is considered to be patched.

3 METHODOLOGY

In this section, we elaborate the core components of our approach.

3.1 Target Function Matching

Given a target program, a VF and its PF, in order to reduce the time consumption, we first narrow down the searching scope by locating TFs in the target program that are similar to VF, on which the patch presence identification is performed. We adopt a lightweight technique for target function matching. For scalability consideration, taking insights from [11, 18, 20], we use syntactic and structural information of functions to construct the function signatures and check whether two functions are similar or not. The syntactic information consists of the sequence of mnemonic operators in binary instructions, function calls and constant values. The structural information consists of the number of instructions, basic blocks, branches and the control flow graph.

Using the function signatures, our target function matching achieves a high accuracy to identify the TFs. This is because that vulnerable functions are usually large whose syntactic and structural information are rich and unique, which allows us to differentiate them from other functions. Therefore, although the lightweight technique scarify a little accuracy in order to be scalable, the results are still accuracy enough for patch presence identification in our experiments. Remark that this target function matching is not the contribution of this work, hence it may be similar to other existing approaches. We include it because we would like to show the complete workflow of our framework.

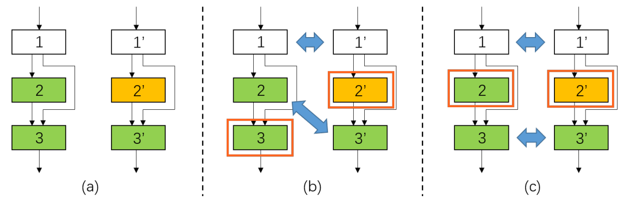


Figure 3: Case for Duplicate Basic Blocks Mapping

3.2 Patch Signature Generation

In patch signature generation, BinXray first computes the differences between the binary code of a VF and its corresponding PF, then creates a patch signature, which will be used in patch presence identification. It consists of three sub-components: binary instruction normalization, basic block mapping and valid trace generation.

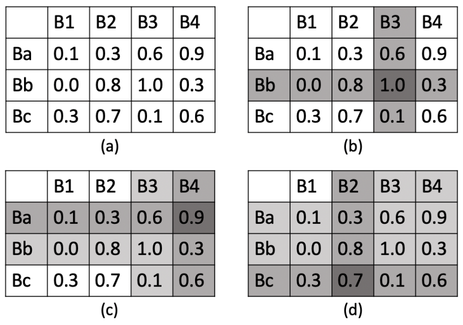
3.2.1 Binary Instruction Normalization. The source code of a program is compiled into binary instructions by compilers. Due to compilation optimization, register allocation and assignment, address shifting, and other compilation settings, the binary instructions of two similar or same functions may be dissimilar after compilation. Since BinXray generates patch signatures by leveraging the differences between vulnerable and patched functions, it is important to eliminate changes that are introduced by compilers. Otherwise, the generated patch signatures will contain many irrelevant changes. Therefore, normalization is used to reduce the compiler introduced changes.

BinXray applies three normalization rules to binary instructions: **address normalization:** replacing concrete addresses by a symbolic term "address"; **memory normalization:** replacing indirect memory access by a symbolic term "mem"; and **register normalization:** replacing concrete registers by a symbolic term "reg". The normalization process is exemplified as follows:

Address normalization:
call 0x80488094 ->call address
Memory normalization:
mov [ebp], edx ->mov mem, edx
Register normalization:
mov ebp, esp ->mov, reg, reg

It is important to mention that BinXray does not normalize constants since some patches may change constants only in order to fix vulnerabilities, e.g., increasing the buffer size to fix a buffer overflow vulnerability. Normalizing constants will prevent BinXray from capturing this kind of patches.

3.2.2 Basic Block Mapping. Basic block mapping tries to map the same basic blocks between two functions. It is a technique to compute differences of the functions, in which unmatched blocks are regarded as the differences. There are basic blocks mapping algorithms in literature such as Bindiff [21] and Diaphora [5]. However, they sacrifice the accuracy in order to maximize the mapping speed and improve the scalability. Indeed, they use the hash values of the basic blocks to perform the mapping. If there are several duplicated blocks in one function that have the same hash value, they may fail to match the block with the correct one. For example, Figure 3(a) shows two CFGs need to be matched, as block1 and block1' are the same, block2, block3 and block3' are the same, and block2' is



	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

(a)

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

(b)

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

(c)

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

(d)

Figure 4: Example for Greedy Matching Algorithm

changed from block2 and different compared to others. Existing approaches can match block1 with block1'. However they may mismatch block2 with block3' since they share the same hash value as shown in Figure 3(b). The inaccuracy mapping will significantly affect the validity of the generated patch signatures. Therefore, we propose a new basic block mapping algorithm, which takes the syntax and context information of the basic blocks to alleviate the hash collision problem (i.e., different basic blocks have same hash value). Our method can complete mapping like Figure 3(c), where block2 and block2' can be recognized as CBBs correctly. We manually verified 36 real cases, that duplicate blocks turn up in one function. Our experimental results show that it outperforms the existing mapping algorithms in [5, 21].

Our basic block mapping algorithm first computes the hash values of basic blocks in both functions, based on their normalized instructions. Then it puts all the basic blocks with same hash value into the same basket. After that, if a basket has exactly two basic blocks from two different functions respectively, these two basic blocks are matched. If a basket only have basic blocks from one function, then these blocks must be changed basic blocks (CBB). If a basket has multiple blocks from both functions, then it is nontrivial to connect a mapping between these basic blocks. To solve this problem, we leverage their structural information and propose a greedy algorithm to establish the mapping between two basic blocks using similarity scores. For each pair of basic blocks from two different functions in one basket, BinXray computes a similarity score between them using their structural information (note that their syntactic information is already encoded as hash values). The similarity score of two basic blocks is computed by calculating the edit distance of the normalized instruction sequence in their adjacent basic blocks. If a basic block has more than one adjacent basic blocks, the similarity score will be weighted according to the control flow. The aggregated weight of the in-degree is normalized to be the same as the out-degree. After obtaining the pair-wise similarity scores of basic blocks, all the pairs will be put in a matrix. BinXray iteratively selects a pair of basic blocks from the matrix that has the highest similarity score, regards it as a pair of matched basic blocks and removes them from the matrix, until no more pair of basic blocks can match. Finally, all the remaining basic blocks in the matrix are regarded as CBBs. After computing the basic block mapping in one basket, the information will be propagated to other baskets as the contexts for other basic blocks.

Example. Suppose there are seven basic blocks Ba, Bb, Bc, B1, B2, B3, and B4 in one basket, where Ba, Bb, and Bc are from a VF, and the others are from its corresponding PF. The similarity scores of basic block pairs are computed as shown in Figure 4(a). The greedy algorithm first selects the pair (Bb,B3) which has highest similarity score 1.0 and removes them from the matrix, resulting the matrix as shown in Figure 4(b). Repeating this procedure, the pairs (Ba, B4) and (Bc, B2) are matched, the resulting matrixes are shown in Figure 4(c) and Figure 4(d). Finally, the remaining basic block is B1 from the PF which is regarded as a CBB.

3.2.3 Valid Trace Generation. To precisely express the patch signature of the given VF and PF, BinXray generates two sets of valid traces, one for VF and the another for PF. Based on the basic block mapping results between the VF and PF, it locates all the CBBs of these two functions. BinXray identifies the boundary basic blocks (BBBs) for each CBB, where a BBB is either an adjacent basic block of the CBB but not a CBB, or the root or leaf of the CFG of the function. A valid trace (VT) is a sequence of consecutive basic blocks that starts and ends with some BBBs, crosses at least one CBB, without any loops. If a loop occurs, we will flat (to treat the loop as being iterated once) it so that they will not affect the number of traces. Since BinXray relies mostly on syntax information, flatten the loop is a good choice which does not alter the syntax much. To build the VT, we put all the CBBs and BBBs into one connected graph. All the BBBs are the root and leaf nodes of the graph; and the CBBs are internal nodes. The valid traces are all the possible paths in the graph. BinXray generates two sets of VTs for the VF and PF, which are regarded as the patch signature. We denote by T_1 the set of VTs of the VF, and T_2 the set of VTs of the PF.

Example. Recalling the running example, CBBs in the function `dtls1_process_heartbeat()` in the version `1.0.1f` (cf. Figure 1(a)) are: A, C, D, and F, and the BBBs in this function are: B, E, K, G. The CBBs in the patched function (cf. Figure 1(b)) are: A', C', D', E', F', G', I', L', M' and N', and the BBBs in this function are: B', H', P', J' and U'.

The VTs of the function `dtls1_process_heartbeat()` in version `1.0.1f` (i.e., Figure 1(a)) are:

A->B; B->C->D->E;
A->C->D->E; B->C->D->G;
A->C->D->G; B->C->K;
A->C->K; E->F->G;

Similarly, the VTs for the patched function (i.e., Figure 1(b)) are:

A'->B'; A'->C'->L'->U';
A'->C'->D'->M'->U'; A'->C'->D'->E'->F'->N'->U';
A'->C'->D'->E'->F'->G'->H'; A'->C'->D'->E'->F'->G'->J';
A'->C'->D'->E'->P'; B'->C'->L'->U';
B'->C'->D'->M'->U'; B'->C'->D'->E'->F'->N'->U';
B'->C'->D'->E'->F'->G'->H'; B'->C'->D'->E'->F'->G'->J';
B'->C'->D'->E'->P'; H'->I'->J';

The VT sets can be optimized by merging the VTs which are connected end to end. The advantage of combining CBBs with BBBs to form the VTs is twofold. First, it pinpoints changes induced by vulnerability patching. Figure 5 has shown the real-world vulnerability `CVE-2015-1790` in OpenSSL as an example. The CFG on the left part shows the related function `PKCS7_dataDecode()` in OpenSSL 1.0.1 l. The right part presents the detail of the function

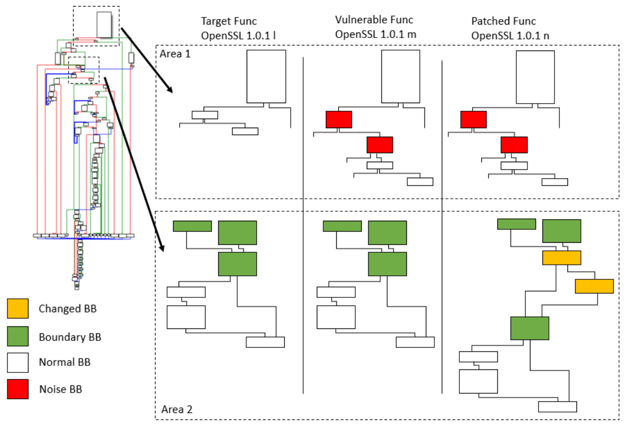


Figure 5: Justification of Using Boundary Basic Blocks

in three consecutive versions. In OpenSSL 1.0.1 l, the function is vulnerable, and we regard it as a target. In 1.0.1 m, the function has been updated (the red color blocks) in Area 1, but it is still vulnerable. In 1.0.1 n, the vulnerability has been fixed by adding sanity checks (the yellow blocks) in Area 2. BBBs help to locate the areas which are related to the patch signatures in target functions. Changes outside the areas will be filtered out. If we use the signature to detect the patch presence in the target function (1.0.1 l), according to the BBBs, only changes in Area 2 will be considered. Other changes will be treated as noises so that they will not affect the patch presence identification. On this example, using BBBs (green blocks), BinXray can focus on the changed (yellow) blocks, while neglecting the noises (red blocks).

Second, it significantly reduces the number of basic blocks and traces used in signature generation and patch presence identification. As shown in Figure 5, the function `PKCS7_dataDecode()` has around 100 basic blocks, which can form thousands of different traces. If we enumerate all the blocks and traces to build the signature, the signature size would be too large and the patch identification process would take a long time. Using BBBs, the signature size is reduced to 5 basic blocks and 2 traces. Our experimental results show that using BBBs significantly improves BinXray's performance. Indeed, BinXray can finish analysis in less than one second even on a large function, while it will takes more than ten minutes, if the entire function is used to build the signature.

3.3 Patch Presence Identification

Patch presence identification predicates whether a TF has been patched or not. The key idea is check whether the TF is more similar to VF or PF. If the TF is more similar to VF than PF, then the TF is regarded as a vulnerable function, otherwise it is regarded as a patched one. The patch presence identification consists four parts: trace generation, trace reduction, similarity comparison and decision algorithm.

3.3.1 Trace Generation. To match patch signatures in target function, BinXray first creates four sets of valid traces from VF, PF, and TF: T_3, T_4, T_5 and T_6, as shown in Figure 6, where (T_1, T_2) is the patch signature.

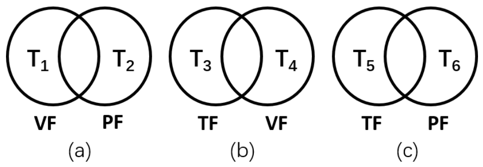


Figure 6: The Relationship between Valid Traces Sets

To generate T_3 and T_4, BinXray first builds the mapping between the VF and TF by using the basic block mapping algorithm in Section 3.2.2, and then extracts the CBBs of the VF and TF. Unlike the patch signature generation, BinXray reuses the BBBs generated between the VF and PF. It creates set T_3 by combining the CBBs in TF with BBBs from patch signatures which are adjacent to those CBBs. One or more local CFGs can be constructed, and trace sets are generated by traversing those local CFGs. Similarly, it creates set T_4 by combining the CBBs in VF with the corresponding BBBs. The reason for using BBBs is that it ensures only the relevant blocks are included in the sets. If there are some CBBs in other part of the function which are not related to the vulnerability, they will not form valid traces since no BBBs are adjacent to them. Therefore, it helps to remove the noises. Finally, the same procedure is applied to get the set T_5 from the TF and the set T_6 from the PF by leveraging the mapping between the TF and PF.

3.3.2 *Trace Reduction.*

To further improve the prediction accuracy and performance, BinXray will eliminate irrelevant traces in the aforementioned trace sets (T_4 and T_6). For each valid trace t in the set T_4, if there does not exist any valid trace in T_1 that contains the same CBB of t, the valid trace t will be eliminated from the set T_4, resulting in a reduced set of valid traces T_{41}. Similarly, BinXray will eliminate the irrelevant trace from T_6, by comparing with the traces in T_2, resulting in a reduced set of valid traces T_{62}. The reduction ensures that the remaining traces in T_{41} and T_{62} will contain some CBBs in the patch signature, which are considered as relevant in the patch presence identification.

Example. Figure 7 provides the complementary part for the running example in Figure 1. The VF in Figure 7(a) is the same as the one in Figure 1(a). Figure 7(b) shows the function in a modified version as the TF. The basic block P'' is added into the TF, resulting in blocks (I, J) being marked as CBB in the VF in Figure 7(a). Recall that the BBBs (B, E, K, G) are in the patch signature, BinXray generates the trace set T_4 for the VF, which contains only one valid trace as shown below:

$$G\text{->}I\text{->}J;$$

In trace reduction, each trace in the reduced sets needs to contain at least one block in the CBB sets of the patch signature. In Figure 1(a), the CBBs are: A, C, D and F. Therefore, the trace G->I->J will be removed by the trace reduction. By removing it, BinXray filters out the irrelevant traces and the final accuracy will be improved.

3.3.3 *Similarity Comparison.*

BinXray performs patch presence identification by calculating the similarity score between trace sets and patch signatures. To compute similarity score for each pair of trace sets, we first show how to compute similarity score for a pair

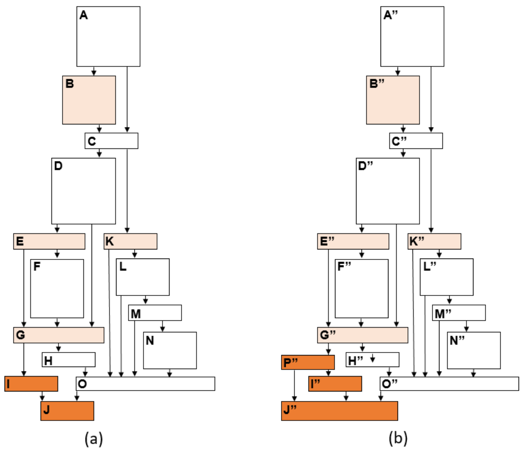


Figure 7: Running Example Continues: function dtls1_process_heartbeat() in OpenSSL

of traces. For each trace in a trace pair, BinXray will first connect the basic blocks in the trace to form a sequential instruction trace by removing all the jump instructions. Then, two instruction traces are compared to obtain the similarity score. The score is computed according to the `Equation (1)`.

$$Sim(t_1, t_2) = \frac{max(len(t_1), len(t_2)) - edit(t_1, t_2))}{max(len(t_1), len(t_2))} \quad (1)$$

BinXray computes the Levenshtein distance between two instruction traces, denoted as $edit(t_1, t_2)$. Then the distance is deducted from the maximum length of the traces and divided by the maximum length. The resulting score measures the normalized similarity between two traces.

After having individual score of each pair of traces, the final similarity between two trace sets is computed according to the following equation.

$$Sim(T_1, T_2) = \sum_{\substack{t_1 \in T_1, \\ t_2 \in T_2}} \frac{Sim(t_1, t_2) * (len(t_1) + len(t_2))}{|T_1| * len(T_2) + |T_2| * len(T_1)} \quad (2)$$

The similarity score of two trace sets $Sim(T_1, T_2)$ is normalized according to the trace length. The score is timed by the length of the t_1 and t_2, and divided by a base, which is the number of traces in the T_1 times the total length of all traces in T_2 add the number of traces in the T_2 times the total length of all traces in T_1. It ensures that the final score is scaled within the range [0,1].

The equation guarantees that the longer traces contribute more weight than the shorter ones. The long traces are usually the ones that contain the patch information. Therefore, they should be more contributive to the similarity score. The short traces usually come from the small and isolated blocks in the function, which might be the noise with little patch information. They should have less weight so that the overall score will not be affected by them.

Example. Figure 8 displays two sets of valid traces for the similarity computation. BinXray will compare every trace in A with every trace in B. According to `Equation (1)`, the similarity score of (`A1`, `B1`) is 0.8 (4 out of 5 instructions match) and the similarity score

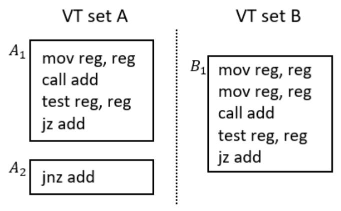


Figure 8: Trace Sets Example for Similarity Score Calculation

of (A2, B1) is 0 (none of the instructions match). Then, according to `Equation (2)` the overall similarity is normalized based on the trace length, which is `0.48` $(= 0.8 * 9/15 + 0 * 6/15)$. Due to the normalization, although the trace A2 is different from B1, it has less effect to the overall similarity score than trace pair (A1,B1). It helps to mitigate the noise by lowering the weight of the short instruction traces.

3.3.4 Decision algorithm. We propose a decision algorithm to determine whether a target function has been patched. The core idea of this algorithm is to infer the relationship of functions by leveraging their differences and similarities. It is more accurate in dealing with partial similarity problem. Figure 6 shows the relationship between valid traces sets. Each trace set represents the unique part related to vulnerability in the corresponding function. There are three cases described below:

CASE 1: T_1 and T_2 are both non-empty. If the TF has been patched, then the difference between the TF and VF should be more significant than the one between the TF and PF. Otherwise if the TF is vulnerable, then the difference between the TF and PF should be more significant than the one between the TF and VF. Therefore, in this case, BinXray checks whether Sim(T_3,T_2)>Sim(T_5,T_1) holds or not. If Sim(T_3,T_2)>Sim(T_5,T_1) holds, then we say the TF has been patched; otherwise it is vulnerable.

CASE 2: T_1 is empty and T_2 is non-empty. T_1 is empty meaning that the patch has added some fresh code. If the TF has been patched, then T_3 will be similar to T_2 and T_{62} will be empty. Otherwise if the TF is vulnerable, then T_{62} will be similar to T_2, and T_3 will be empty. In this case, BinXray checks whether Sim(T_2,T_3)>Sim(T_2,T_{62}) holds or not. If Sim(T_2,T_3)>Sim(T_2,T_{62}) holds, we say the TF has been patched; otherwise it is vulnerable.

CASE 3: T_2 is empty and T_1 is non-empty. T_2 is empty meaning that the patch deleted some code. Similar to CASE 2, we say the TF is patched if Sim(T_1,T_{41})>Sim(T_1,T_5); otherwise, it is vulnerable.

4 EVALUATION

4.1 Experiment Setups

Our approach takes the binary code of pairs of vulnerable and patched functions as input. We choose to collect vulnerabilities from widely-used libraries with Common Vulnerabilities and Exposures (CVE) identifiers. There are websites providing CVE information, such as CVE Details [3], NVD [7] and CVE List [4]. In addition, some open source library websites also provide well-documented security updates, such as OpenSSL [9]. We extract CVE information from those websites, which consists of the CVE IDs, names of involved functions, versions of programs that are vulnerable or patched. For each CVE used in the experiments, we manually confirmed that its information is valid. In future, we plan to collect more vulnerabilities from security update commits using security-related commit identification approaches, such as the approach in [42] for source code and [44] for binary executable.

Table 2: Real-World Programs and Their Patch Presence Identification Results

Program	CVE (#)	Version (#)	Function (#)	Function Accuracy	CVE Accuracy
Openssl	70	22	2280	95.04%	95.72%
FFmpeg	52	55	1486	89.64%	92.37%
Libxml2	37	12	852	94.95%	97.92%
Freetype	57	19	616	93.34%	98.14%
binutils	147	10	412	90.33%	96.24%
Tcpdump	88	3	273	100%	100%
Libpng	4	18	162	95.68%	100%
Openvpn	6	7	53	100%	100%
Sqlite3	6	10	53	81.13%	93.75%
Libeixf	7	12	28	75%	100%
Libxslt	3	5	15	100%	100%
Expat	2	4	8	87.50%	100%
Total	479	-	6238	93.31%	96.87%

Using the collected CVE information, we download the source code of target programs, compile them into binary executables using gcc with default optimization (-O2), and then dump the binary code of the functions using the binary disassembler IDA Pro [6]. For each function with a CVE, we save two versions: its vulnerable and patched versions respectively. For each CVE the changed functions before the patch commit are considered as vulnerable and functions after the patch commit are considered as patched.

In the experiments, we aim to answer the following research questions:

RQ1 - How accurate is BinXray for patch detection?

RQ2 - What is the (breakdown) performance of BinXray?

RQ3 - How is the result of BinXray, compared to other related works?

RQ4 - What are the useful applications of BinXray?

RQ1 and **RQ2** evaluate whether BinXray meets **P1** and **P2** described in Section 1. As BinXray is designed not to use any source code level information, it meets **P3** by nature.

In the experiments, BinXray utilizes IDA Pro [6] to disassemble binaries and dump the binary function information. BinXray is implemented in Python 2.7 with more than 2K lines of code, and supports for Intel X86 32bit and 64bit, ARM 64bit architectures. All the programs run at HP desktop Z640 with CPU E5-2697 at 2.60GHz and 64GB memory. We have released all the experimental data at our website [8].

4.2 Accuracy Evaluation (RQ1)

In this evaluation, we aim to test whether BinXray can successfully predict the patch presence given a potentially vulnerable function. To prepare the data for the experiments, we have selected 12 different real-world programs across different domains such as cryptography, database, and image processing. The selected benchmark programs are diverse, representative, and widely used in the literature [15, 17, 31, 45] and have adequate amount of well-documented vulnerabilities with CVE numbers. For each program, we collected the binary level vulnerable and patched functions by using function matching with its CVE function signatures. We manually confirmed that all the functions used in the experiments are either vulnerable or patched so that we can evaluate the patch identification accuracy in controlled settings. Then, we collected binaries in all the versions for each program as targets. Note that in some versions of binaries, there are a few vulnerable functions that are inlined or removed from the binary during the compilation process. We exclude these cases in our experiments. In total, we collected 479 CVEs of these programs, and collected all the functions affected by these CVEs. After matching those functions in different versions of binaries, we obtained 6238 target functions as test data. We run the patch identification process of BinXray on these functions and compared its results with the ground truth to obtain the accuracy measurement.

Table 2 shows the prediction results on these functions. The first four columns report the program name, the number of CVE collected, the number of versions tested in the experiments, and the number of functions to be predicted respectively. The fifth column provides the accuracy in predicting the patch presence for each of the target functions found by BinXray. The last column provides the accuracy in predicting whether the CVE (vulnerability) has been patched or not. When a CVE affects only one function, BinXray predicts the CVE still exists in the binary if this function is predicted as vulnerable, and predicts the CVE has been patched if this function is predicted as patched. When one CVE contains multiple vulnerable functions, BinXray will predict as vulnerable, if more than one of the functions are predicted as vulnerable Otherwise, it is predicted as been patched.

The overall accuracy is above 93% for function level patch identification and over 96% for predicting the patches of CVEs. The results show that for most of the programs with a few CVEs (i.e., TCPdump, Libpng, Openvpn, Libeixf, Libxslt, and Expat), BinXray can identify the patch presence of the CVE with 100% accuracy. The reason for the high accuracy is twofold. First, some of the functions are very stable with only a few changes across all the versions. Therefore, the patch signature will be the dominant change that contributes the most to the final result, and make the function easy to be identified correctly. Second, the functions that contain the CVEs are usually large, so that different changes in one function will have a high chance to be placed in different locations. For multiple changes in one function, since BinXray is good at handling the cases where the patch-related changes and other changes are separated, it can accurately identify the patches and correctly predict its presence.

False Prediction Discussion: We have manually examined the false prediction cases made by BinXray. Most of them are caused by the multiple times of changes made at the same location of the function. Since most of the original code in patch related area is modified by multiple changes, the patch signature generated from the old version cannot match the newly modified functions. Therefore, BinXray will make false predictions.

There is a special false prediction case in the experiment of OpenSSL. `CVE-2015-1791` [2] is a race condition vulnerability. It is patched in function `ssl3_get_new_session_ticket()` at version `1.0.1n`. BinXray can successfully distinguish the patch and unpatched functions from version `1.0.1a` to `1.0.1n`. However, after 3 versions at `1.0.1q`, the patch change has been reverted back into the original functions. Therefore, BinXray classifies the version after `1.0.1q` as vulnerable. In fact, the vulnerability has been fixed in another function. Even for human experts, it will takes significant efforts to understand that the vulnerability patch has been replaced by other changes in different places. Theoretically, BinXray has made a correct prediction that the TF is vulnerable. However, due to the patches in other functions, the vulnerability is patched, resulting in a false positive in CVE prediction.

Table 3: Performance of BinXray

Program	Basic Blocks (#)	Total Time (s)	Time per Function (ms)
OpenSSL	54.08	561.95	246.47
FFmpeg	90.18	6.42	4.32
Libxml2	60.47	55.61	65.27
Freetype	66.99	271.37	440.53
Binutils	83.21	375.16	911.32
TCPdump	45.31	28.57	104.65
Libpng	26.40	12.99	80.20
Openvpn	20.20	2.91	54.90
Sqlite3	291.16	449.27	8476.85
Libeixf	208.15	82.87	2959.66
Libxslt	95.07	0.22	14.67
Expat	15.38	0.09	11.25
Average	88.05	-	296.17

Answering RQ1: The results on the real-world programs show that BinXray can effectively identify the patch in the target functions. It has on average 93.31% accuracy in predicting the patch presence and 96.87% accuracy in identifying CVEs.

4.3 Performance Evaluation (RQ2)

Table 3 reports the performance to complete the patch presence identification for the 12 real-world programs. The second column reports the average number of basic blocks in the target function, the third column gives the total overhead for all functions, and the last column reports the average time overhead to identify patch for one function. According to the table, the number of basic blocks in each function is 88.05 on average. BinXray is very efficient, which can make prediction in 296.17ms per function on average.

In the experiment, running BinXray on Sqlite3 took the longest time (8476.85 ms per function). We manually verified the program and located one function that caused the problem. Specifically, there is a very large function, named `sqlite3VdbeExec()`, with more than 1000 basic blocks. The trace sets generated by BinXray consists of thousands of traces, which resulted in the significant time consumption. The predictions on the other functions finished within reasonable time period.

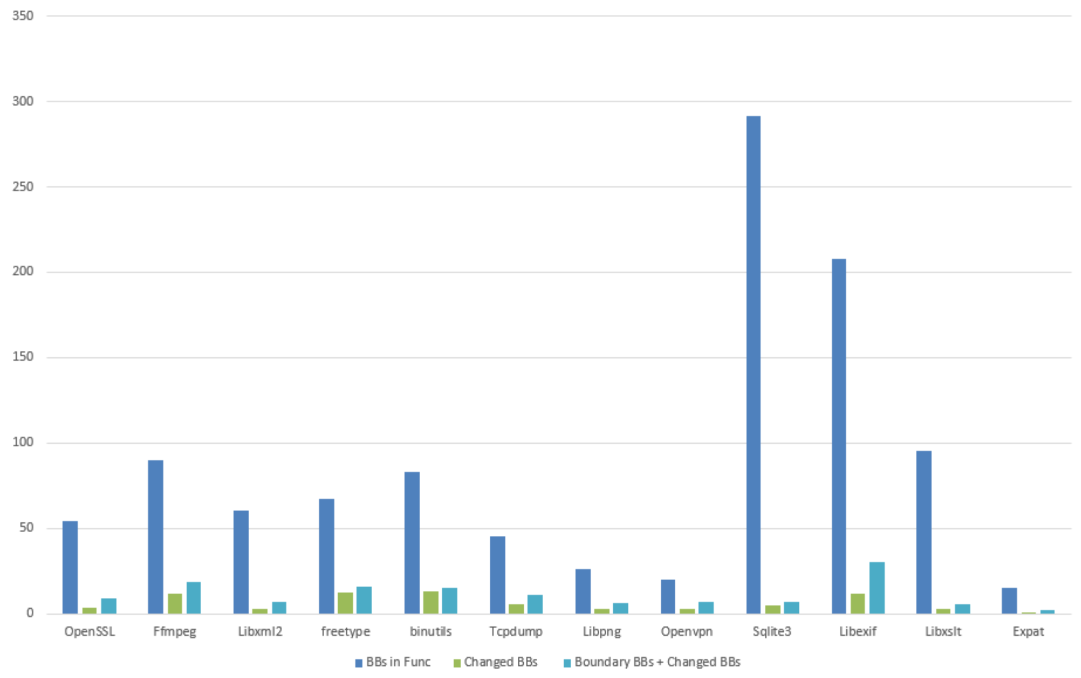


Figure 9: Basic Blocks in Original Functions vs Basic Blocks Used in BinXray

Table 4: Comparison with Binary Function Matching

Tool	Vul (#) Func.	Patched Func.(#)	FP	FN	Time per func.
BinXray	1412	868	29 (1.27%)	84 (3.68%)	246.47ms
BinGO-E			868 (38.07%)	0 (0%)	428.09ms

Figure 9 reports the number of basic blocks that were used by BinXray in the experiment for the 12 real-world projects (a larger version of the figure can be found at [8]). For each project, the first bar stands for the average number of basic blocks inside the target functions. The second bar shows the number of CBBs in the function, and the third bar shows the basic blocks that used by BinXray for signature generation (CBBs and BBBs). From these bars, we can observe that BinXray effectively reduces the size of the basic block sets by 82.55% on average to perform the signature generation and patch identification. The number of basic blocks in the changed area used in BinXray is much smaller than the number of blocks in the entire function. The reduction of basic blocks significantly improves the speed and accuracy of patch identification and make BinXray practical for large real-world programs.

Answering RQ2: BinXray effectively reduces the number of basic blocks used by 82.55%. BinXray can perform vulnerability matching with very low overhead so that it is scalable for real-world binary programs.

4.4 Related Works Comparison (RQ3)

Comparison to Function Matching works. To compare with the existing binary function matching techniques for vulnerability matching, we have selected BinGo-E [45] from recent state-of-the-art works as baseline tool. BinGo-E is selected because it has the best performance and its source code is successfully requested from the authors. We use the Openssl data set (2280 target functions of 70 CVEs in 22 versions) to test BinGo-E. Table 4 shows the results of BinGo-E against BinXray. The second and third columns report the number of vulnerable functions and patched functions in the ground truth. The fourth and fifth columns provide the false positive and false negative. The last column reports the time used to process one function. BinGo-E is an accurate function matching tool, which can match all the functions in data set. However, since it cannot determine the patch presence, it labels all the matched function as vulnerable. However, there are 868 functions having been patched, which results in a high false positive rate (38.07%). In comparison, BinXray has a much lower false positive rate with a reasonable low false negative rate. BinXray takes half of the time to make prediction on one function.

Comparison to other Patch Identification works. In this experiment, we compare BinXray with FIBER [46], a state-of-the-art patch presence identification tool for Android. We have obtained the patch data set from the author of [46]. It contains 107 vulnerabilities in different versions of Android kernels. We managed to select an Android kernel, and construct two versions of it, one reproduces all of the CVEs and the other patched all of them. Excluding the mismatching between the kernel and patch and the impact of compilation, eventually, we manually verify 76 vulnerabilities in the

Table 5: Comparison with Patch Identification

Tool	Data	Accuracy	Time (per Vul.)
FIBER	107 CVEs	94% aver.	min: 12.59s; max: 23.74s
BinXray	76 out of 107	97.37%	49.06ms

compiled Android binaries (Samsung bullhead configuration) and test the performance of BinXray on them.

Table 5 shows the result of FIBER and BinXray on detecting the Android vulnerability patches. The accuracy of FIBER is calculated based on the 107 CVEs as stated in its paper, while BinXray is based on the 76 manually verified CVEs. Overall, BinXray has a slightly better performance than FIBER in terms of accuracy. It is much faster than FIBER, since FIBER requires heavy program analysis. Moreover, FIBER needs the source code information so that it is not applicable for closed source software. BinXray can work in larger scope, since it only requires the binary diffs.

Answering RQ3: BinXray outperforms the vulnerability matching tool BinGo-E. Compared with patch identification tool FIBER, BinXray achieves higher accuracy and speed without using the source code information.

4.5 Applications (RQ4)

4.5.1 Binary Program Version Identification. One potential application of BinXray is to determine the version of unknown binary programs. Version detection is an important step in binary analysis, especially for software security [16]. In different versions of programs, there are changes to update or patch functions. Since BinXray can precisely detect these changes, it can leverage them as signatures to determine the program versions. We conducted experiments in identifying program versions with OpenSSL binaries. First, we used function diffing technique to obtain a changed functions list for OpenSSL. Table 6 has shown the OpenSSL version 1.0.1 a to 1.0.1 e with their corresponding changed functions. The original function is labeled as tag0. After one change has been made, the tag number will be increased by one (tag1). These binary functions are fed into BinXray to learn the signatures. After extracting the signature, BinXray can correctly identify the tags for given binary functions in the table. To determine the binary version, BinXray will predict all the 11 functions' tags and match them against the table. If all the 11 functions have tag0, the given binary should be version 1.0.1 a. If the functions `rc4_hmac_md5_cipher` and `ssl23_client_hello` are tag1 and the rest functions are tag0, the given binary should be version 1.0.1 b, etc. The experimental results show that, by identifying function tags, BinXray can precisely identifies the binary versions.

4.5.2 IoT Device Vulnerability Detection. Firmware in IoT devices utilize many open source libraries, which may contain software vulnerabilities. However, these libraries are often in binary format without source code information. Therefore, as an outsider, it is difficult to know what changes have been made to the libraries and how many vulnerabilities remain. In this situation, BinXray can be used to detect and validate the vulnerabilities in libraries that used by firmware. To demonstrate this capability, we have used BinXray

Table 6: OpenSSL Version Timeline

Function Name	OpenSSL 1.0.1				
	a	b	c	d	e
rc4_hmac_md5_cipher	tag0	tag1	tag1	tag1	tag1
ssl23_client_hello	tag0	tag1	tag1	tag1	tag1
dtls1_enc	tag0	tag0	tag1	tag2	tag2
tls1_enc	tag0	tag0	tag1	tag2	tag2
cms_EncryptedContent...	tag0	tag0	tag1	tag1	tag1
ssl3_get_client_hello	tag0	tag0	tag0	tag1	tag1
www_body	tag0	tag0	tag0	tag1	tag1
do_ssl3_write	tag0	tag0	tag0	tag1	tag1
ssl3_cbc_digest_record	tag0	tag0	tag0	tag0	tag1
tls1_cbc_remove_padding	tag0	tag0	tag0	tag0	tag1
ssl3_cbc_copy_mac	tag0	tag0	tag0	tag0	tag1

to search the vulnerabilities in 7 real-world IoT device firmware that contain Openssl library. We extracted and parsed the binary files in those firmware and used patch signatures generated from Openssl in previous experiments to detect vulnerabilities. In total, it successfully identifies that 49 vulnerabilities have been patched and 48 vulnerabilities still remain in the firmware. The results have been manually confirmed. Due to the complex composition of the firmware and the difficulty of parsing firmware, the average accuracy of BinXray in this experiment is 81.5%. More details can be found at [8]. The result shows that BinXray can provide an effective solution in the scenario mentioned above.

Answering RQ4: Experimental results have demonstrated the capability of BinXray in determining the exact version of the real-world binaries. Moreover, it can precisely detect CVEs in firmware.

4.6 Threats to Validity & Future Work

BinXray has some limitations, which need to be overcome. First, the prerequisite of our work is accurate function matching. If the function matching algorithm cannot find the target functions, BinXray cannot proceed to identify the patches. Therefore, if the target function has been significantly changed so that the function cannot be matched, our algorithm cannot work.

Second, the experiments are conducted on binaries compiled for Intel X86 32 bit, 64 bit, ARM 64bits system. BinXray is designed to support any kind of architectures as long as the signature is extracted within the same type of architecture. However, BinXray currently does not support cross-architecture patch detection. In the future, we plan to use the intermediate representation to lift the binary instruction to higher level to support the cross-architecture comparisons.

Third, as discussed in Section 4.2, BinXray is not able to handle the case where a function receives multiple changes at the same location in different versions. Such changes may mislead BinXray to make incorrect decisions. A possible solution is to use inter-function semantic analysis to find the root cause of the vulnerability and check whether the problem has been addressed or not. However, heavy program analysis will significantly decrease the performance.

Therefore, we need to design algorithms to make trade-off between the performance and accuracy.

5 RELATED WORK

In this section, we review the most closely related works in two directions: binary function matching and binary patch identification.

Binary Function Matching. Binary-level function similarity matching has drawn much attention because of its important applications, e.g., copyright checking, malware analysis and binary program auditing [17, 19, 25, 40]. Saebjornsen et al. [40] attempt to normalize the binary instructions and utilize the structural information to match the similar function codes. BinHunt [22] and iBinHunt [35] use symbolic execution and taint analysis to determine the differences between functions. The heavy program analysis methods introduce high overhead. Therefore, various works try to address the problem by introducing lightweight matching methods. TRACY [15] proposes to use k-tracelet to address basic block merging problem in matching binary functions. DiscovRE [18] tries to identify similar vulnerable functions across architectures with the help of numeric and structural features. BLEX [17] adopts seven dynamic features by executing the functions to be tolerant for small changes. BinGo [11] and BinGo-E [45] try to find similar function pairs by using multiple kinds of features (i.e., syntax, semantics, and emulation features). αDiff [31] uses deep neural networks to automatically learn features from raw bytes of the binaries. Bourquin et al. [10] present a polynomial algorithm to improve the accuracy in calculating the function similarity. Xu et al. [43] use neural networks to embed the features and match the embedded graph to improve the matching speed and accuracy. Luo et al. [33] extract semantic features of the functions and match the functions with obfuscations. Genius [20] builds CFG of functions, embeds the CFG into high level numeric features and searches for known vulnerability in the firmware. Pewny et al. [38, 39] propose cross-architecture bug search by translating the binary instructions into intermediate representations. Lin et al. [30] search the bug in the firmware by using the attributed CFG with support vector machine technique. Hu et al. [23] try to rebuild the argument and indirect jumps in the binary to increase the matching precision. Hu et al. [24] combine the static and the dynamic approaches to detect the code clones with obfuscation. David et al. [13] use statistical probability to measure the similarity of the program strands. Ming et al. [36] determine whether two program execution traces are similar by using the system call sliced segment equivalence checking algorithm.

These works aim to find matching function pairs with the tolerance of small changes. Therefore, they may generate high false positive by matching the vulnerable functions with the patched functions. Our work tries to focus on summarizing the differences between the vulnerable functions and their patched versions. Thus, it can significantly improve the accuracy of similarity-based vulnerability detection by filtering out the patched functions.

Source Code Function Matching and Patch Analysis There are also many works which try to search for similar code or similar functions at source code level (e.g., [26, 28]), while ours works at the binary code level. For known vulnerabilities, source-level patch identification can be done directly by checking whether the code has been updated with patches. Binary-level identification is more challenging since functions will change due to factors (compiler type, version and settings, program updates, and architecture). One needs to distinguish the patch changes out of different changes to perform patch identification.

Binary Patch Analysis and Identification Several studies [29, 42, 47] have surveyed the vulnerability patches in the real-world software to understand the common characterizations of patches. BugTrace [12] tries to connect the link between the vulnerabilities and their fixes via patch analysis. Soto et al. [41] study the patch behaviors in Java program. Kim and Notkin [27] leverage the understanding of patch diffs to help the programmer to find vulnerabilities. These works have shown the detailed studies for vulnerability patches in real-world software. However, they are on the source code level and are not designed specifically for identify vulnerable functions.

Zhang and Qian [46] propose FIBER to predict whether the function in Linux kernel has been patched or not. It relies on source code information of the target function, so that the additional preprocessing is required for different projects. Our work uses only binary information so that it is more convenient and more suitable when the source code is not available. VMPBL [32] tries to further distinguish the patched and unpatched functions by building a vulnerable function database and a patched function database. It applies machine learning to find the real vulnerable functions. It can reduce the false positive rate by half, but requires prior knowledge to build the database. Our work tries to automatically extract patch signatures for each CVE and does not require any prior knowledge. Spain [44] tries to identify the secretly patched vulnerabilities by comparing the semantic changes at binary level. It aims to verify whether a patch is security-related. Our work aims to identity whether the patch exists or not.

6 CONCLUSIONS

This work proposes BinXray to eliminate the false positives in vulnerable binary function matching by identifying the patches in the target functions without using source code level information. It can automatically extract the signatures of the vulnerability patches by diffing the vulnerable and patched functions. The patch signatures are used to match the target functions to determine whether they have been patched or not. The experimental results show that BinXray is able to achieve high accuracy with very low overhead. Moreover, BinXray can help to determine the binary level program versions and finds real-world vulnerabilities in IoT firmware.

ACKNOWLEDGMENTS

This work was partially funded by National Key R&D Program of China (2018YFB0803501), the Fundamental Research Funds for the Central Universities, Singapore National Research Foundation, under its National Cybersecurity R&D Program (Grant Nos.: NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Grant Nos.: NRF2018NCR-NSOE003-0001), NRF Investigatorship (Grant Nos.: NRFI06-2020-0022), National Natural Science Foundation of China (NSFC) grants (No. 61772408, No. U1766215, No. U1736205, No. 61721002, No. 61632015, No. 61532019 , No. 61761136011 and No. 61902306), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] 2014. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[2] 2015. CVE-2015-1791. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-1791.

[3] 2020. CVE Details. https://www.cvedetails.com/.

[4] 2020. CVE List. https://cve.mitre.org/index.html.

[5] 2020. Diaphora. https://github.com/joxeankoret/diaphora.

[6] 2020. IDA Pro. https://www.hex-rays.com/products/ida/.

[7] 2020. NVD. https://nvd.nist.gov/.

[8] 2020. Open Source Data and Results for the Paper. https://sites.google.com/view/submission-for-issta-2020.

[9] 2020. OpenSSL Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html.

[10] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 4.

[11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.

[12] Christopher S Corley, Nicholas A Kraft, Letha H Etzkorn, and Stacy K Lukins. 2011. Recovering traceability links between source code and fixed bugs via patch analysis. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 31–37.

[13] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 392–404.

[15] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[16] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. ACM, 2169–2185.

[17] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.

[18] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.

[19] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Proceedings of the 8th International Conference on Software Security and Reliability*. 78–87.

[20] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.

[21] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*. Citeseer, 161–174.

[22] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.

[23] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 88–98.

[24] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A Semantics-based Hybrid Approach on Binary Code Clone Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 104–114.

[25] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. 309–320.

[26] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.

[27] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 309–319.

[28] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: a scalable approach for vulnerable code clone discovery. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 595–614.

[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*. ACM, 2201–2215.

[30] Hong Lin, Dongdong Zhao, Linjun Ran, Mushuai Han, Jing Tian, Jianwen Xiang, Xian Ma, and Yingshou Zhong. 2017. CVSSA: Cross-Architecture Vulnerability Search in Firmware Based on Support Vector Machine and Attributed Control Flow Graph. In *Dependable Systems and Their Applications (DSA), 2017 International Conference on*. IEEE, 35–41.

[31] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 667–678.

[32] Danjun Liu, Yao Li, Yong Tang, Baosheng Wang, and Wei Xie. 2018. VMPBL: Identifying Vulnerable Functions Based on Machine Learning Combining Patched Information and Binary Comparison Technique by LCS. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 800–807.

[33] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.

[34] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 24–35.

[35] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*. Springer, 92–109.

[36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*.

[37] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 180–190.

[38] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.

[39] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 406–415.

[40] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 117–128.

[41] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 512–515.

[42] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 386–396.

[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 363–376.

[44] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 462–472.

[45] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* (2018).

[46] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 887–902.

[47] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 913–923.