electronics

Article

P1OVD: Patch-Based 1-Day Out-of-Bounds Vulnerabilities Detection Tool for Downstream Binaries

Hongyi Li [1,†], Daojing He [1,2,*,†], Xiaogang Zhu [3] and Sammy Chan [4]

[1] Software Engineering Institute, East China Normal University, Shanghai 200062, China; 51194501009@stu.ecnu.edu.cn
[2] School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China
[3] Department of Computer Science and Software Engineering, School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne 3122, Australia; xiaogangzhu@swin.edu.au
[4] Department of Electrical Engineering, City University of Hong Kong, Hong Kong SAR, China; eeschan@cityu.edu.hk
* Correspondence: djhe@sei.ecnu.edu.cn or hedaojinghit@163.com; Tel.: +86-21-6223-1233
† These authors contributed equally to this work.

Abstract: In the past decades, due to the popularity of cloning open-source software, 1-day vulnerabilities are prevalent among cyber-physical devices. Detection tools for 1-day vulnerabilities effectively protect users who fail to adopt 1-day vulnerability patches in time. However, manufacturers can non-standardly build the binaries from customized source codes to multiple architectures. The code variants in the downstream binaries decrease the accuracy of 1-day vulnerability detections, especially when signatures of out-of-bounds vulnerabilities contain incomplete information of vulnerabilities and patches. Motivated by the above observations, in this paper, we propose P1OVD, an effective patch-based 1-day out-of-bounds vulnerability detection tool for downstream binaries. P1OVD first generates signatures containing patch information and vulnerability root cause information. Then, P1OVD uses an accurate and robust matching algorithm to scan target binaries. We have evaluated P1OVD on 104 different versions of 30 out-of-bounds vulnerable functions and 620 target binaries in six different compilation environments. The results show that P1OVD achieved an accuracy of 83.06%. Compared to the widely used patch-level vulnerability detection tool ReDeBug, P1OVD ignores 4.07 unnecessary lines on average. The experiments on the *x86_64* platform and the *O0* optimization show that P1OVD increases the accuracy of the state-of-the-art tool, BinXray, by 8.74%. Besides, it can analyze a single binary in 4 s after a 20-s offline signature extraction on average.

Keywords: out-of-bounds; vulnerable detection; patch

Citation: Li, H.; He, D.; Zhu, X.; Chan, S. P1OVD: Patch-Based 1-Day Out-of-Bounds Vulnerabilities Detection Tool for Downstream Binaries. *Electronics* **2022**, *11*, 260. https://doi.org/10.3390/electronics11020260

Academic Editor: Arman Sargolzaei

Received: 13 December 2021
Accepted: 12 January 2022
Published: 14 January 2022

1. Introduction

Vulnerabilities acknowledged by vendors are called 1-day vulnerabilities and are often fixed by upstream software developers using security patches [1]. In the past decades, 1-day vulnerabilities are widely spread among cyber-physical devices due to the popularity of open-source software cloning [2]. In the Debian system alone, researchers [3] have found 145 cloned 1-day vulnerabilities. Over the last few years, various automatical 1-day vulnerability detection tools for binaries have been proposed [3–18] to protect users who fail to adopt 1-day vulnerability patches in time.

However, manufacturers usually non-standardly build binaries from customized source codes for multiple target architectures. Such code variants decrease the accuracy of 1-day vulnerability detection, especially out-of-bounds vulnerabilities. Moreover, the inaccuracy can lead to safety risks or extra manual efforts for security analysis. Due to the prevalence of 1-day out-of-bounds vulnerabilities, in this paper, we propose P1OVD, a 1-day out-of-bounds vulnerability detection tool, which has higher accuracy when code variants appear in the downstream binaries.

Code variants are common and diverse. Research shows that among 6027 counterparts of 285 Android Kernel functions, over 72% of them contain codes that are different from their mainstream versions [19]. On the one hand, the code variants that are caused by target architectures or optimization levels can be large. Although they heavily change the instructions, function basic blocks, and function CFGs (control flow graph), they hardly change function logics. On the other hand, code variants can be caused by patching vulnerabilities or unexpectedly introducing new vulnerable modules. So these code variants are critical but small. These two kinds of code variants can appear at the same time, causing two challenges.

The first challenge is that out-of-bounds vulnerability signatures can easily neglect small but important code variants. Function-level 1-day vulnerability detection tools such as Asm2Vec [16] generate vulnerability signatures from the whole vulnerable functions [4–16]. Due to their extremely large scope, they fail to capture the precise context of vulnerabilities. At the same time, patch-level 1-day vulnerability detection tools such as ReDeBug [3] and BinXray [17] generate vulnerability signatures only from patches and their signatures contain incomplete vulnerability information [3,17,18]. As a result, when the small code variants influence the vulnerability root causes, the existing tools give false predictions.

The second challenge is that large code variants can decrease the accuracy of the matching methods that depend on AST (abstract syntax tree) shaped out-of-bounds patch signature. Moreover, many existing works have difficulties in balancing the accuracy and robustness. The strict operand-based matching [20] is accurate but sensitive to unimportant code variants, while the graph-similarity-based algorithms [8,9,13,19] improve the robustness but sacrifice accuracy.

To solve the above two challenges, we propose a patch-based 1-day out-of-bounds vulnerability detection tool named P1OVD, which can automatically find 1-day out-of-bounds vulnerabilities in the downstream binaries. It first analyzes patches and outputs source signatures, which solves the first challenge. Then the signature generator maps the source signatures to binary signatures. Finally, the matching engine scans the target binary with the binary signatures. Moreover, when P1OVD matches patch signatures, it uses the novel matching algorithm to solve the second challenge. To evaluate the efficiency and effectiveness, we test P1OVD based on a dataset containing 620 binaries, which are compiled under six compilation environments from 104 different versions of 30 out-of-bounds vulnerable functions. The result shows that P1OVD has a total accuracy of 83.06% and achieves an 8.74% higher accuracythan the state-of-the-art tool BinXray. Besides, it can analyze a single binary in 4 s after a 20-s offline signature extraction on average.

We summarize our contributions as follows:

- We design an out-of-bounds vulnerability signature that mainly contains patch information and vulnerability information.
- We propose a matching algorithm that can accurately and robustly find the patch signatures in downstream binaries.
- We propose a patch-based out-of-bounds vulnerability detection method, P1OVD. P1OVD can accurately locate 1-day out-of-bounds vulnerabilities in downstream binaries even if code variants exist. We evaluate its performance on 620 binaries of 30 real-world patches in Linux Kernel [21].

The rest of this paper is organized as follows. We first summarize the challenges in Section 2. Then we describe the design of P1OVD in Section 3 and evaluate P1OVD in Section 4. Next, we review related work in Section 5. Finally, we give the conclusion in Section 6.

2. Motivation

The open-source software can be built with customized codes and non-standard building configurations to meet the needs of downstream manufacturers [19], causing

code variants. We believe that there are two main challenges caused by code variants: vulnerability signature (Section 2.1) and patch signature matching (Section 2.2).

2.1. Vulnerability Signature

The first challenge is that the code variants can significantly affect the vulnerability detection results but can be hard to detect if the vulnerability signatures do not contain enough patch information and vulnerability information. Version differences are parts of the results of the third-party code customization and cause existing works to have high false rates in out-of-bounds vulnerability detection. We take the function *init_desc* of the Linux Kernel as a motivation example to figure out the severe impacts of code variants. Figure 1 shows three different versions of function *init_desc*. Red codes are the earliest version, at that time, the vulnerable statement *hash_algo_name[hash_algo]* had not existed. Then a commit removed the red codes and introduced green codes, where the out-of-bounds vulnerability is located. The parameter *hash_algo* is possibly tainted and can read the array *hash_algo_name* out of the buffer bound. The blue codes are added by the patch, they restrict the parameter *hash_algo*, and relieve the panic.

```
static struct shash_desc *init_desc(char type)
static struct shash_desc *init_desc(char type, uint8_t hash_algo)
{
        long rc;
        char *algo;
        const char *algo;
        ......
        if (type == EVM_XATTR_HMAC) {
                ......
        } else {
                if (hash_algo >= HASH_ALGO__LAST)
                        return ERR_PTR(-EINVAL);
                tfm = &hmac_tfm;
                algo = evm_hmac;
                tfm = &evm_tfm[hash_algo];
                algo = hash_algo_name[hash_algo];
```

Figure 1. Function *init_desc* in three Versions.

These three versions challenge the state-of-the-art tools because their signatures miss either patch information or vulnerability information. As Figure 2 shows, function-level vulnerability detection tools [4–15] take the whole 53-line function into concern and fail to capture the precise context of vulnerabilities. As a result, they think the functions that are similar to known vulnerable functions are vulnerable. Due to the small differences between these three versions, they think the three versions are all vulnerable, which results in high false positives. As Figure 2 shows, some patch-level vulnerability detection tools [3,17], mistakenly think the patch disappearances are the vulnerabilities and fail to include vulnerability information into their signatures. So they focus on the blue codes rather than the green codes. As a result, the red version is labeled vulnerable even if it has no vulnerable operation at all.

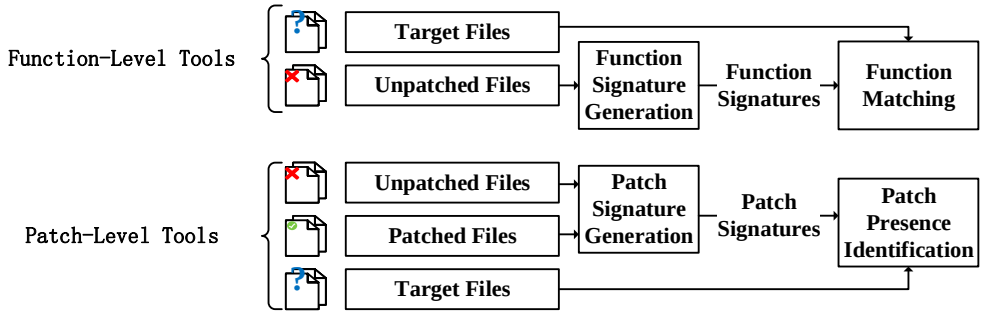

Figure 2. Function-level tools and patch-level tools.

By considering both vulnerabilities and patches, MVP [18] outperforms vulnerability detection tools. Further, researchers manually evaluate MVP's failures by comparing signatures with vulnerability root causes in case studies. Although MVP can successfully identify all three versions, its vulnerable line searching algorithm introduces a few vulnerability-irrelevant codes, e.g., the function call of *ERR_PTR*, which can be replaced by customized error handlings and harm the binary-level signatures.

2.2. Patch Signature Matching

As the solution of the first challenge, the binary-level patch information is AST-shaped. However, code variants caused by optimization levels or target architectures can influence the structures of ASTs, which is the second challenge. As Figure 3 shows, the patch checks inputs at 0x401955 (*x86_64*) and 0x17e4 (*aarch64*). The AST in *x86_64* is *[arg + 6] <= 6*, while the AST in *aarch64* can be *![arg + 6] > 6* if patched or *![arg + 6] > 8* if unpatched. The *arg* represents a function argument and now it stands for the variable *cmd*.

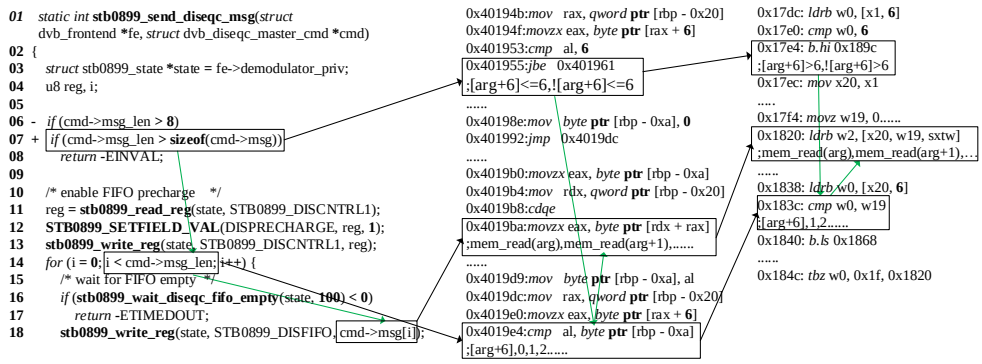

Figure 3. Patch for commit b9f62ffe, patched binary in *x86_64-O0* and target binary in *aarch64-O2*.

There are two kinds of matching algorithms. However, neither of them can balance accuracy and robustness. First, Fiber [20] performs a strict operand-based matching, while assuming that the same semantic can result in the same ASTs with few changes on the address-related immediate numbers. However, as Figure 3 shows, the *[arg + 6] <= 6* in *x86_64* can be transformed into the *![arg + 6] > 6* in *aarch64*. A strict matching can falsely think the patch signatures generated on the *x86_64* platform are different from the patch signatures generated on the *aarch64* platform. Second, Pewny et al. [8,9], Feng et al. [13] and Jiang et al. [19] match ASTs with an inaccurate graph-similarity-based structural matching to improve the robustness. However, as Figure 3 shows, the patch only changes *8* to *sizeof(cmd->msg)*, while the latter is an immediate number *6*. These tools cannot distinguish the unpatched versions from the patch versions because the patch does not cause any structural difference.

3. Design of P1OVD

In this section, we first introduce the architecture of our tool (Section 3.1), and then we will introduce the three main parts of P1OVD in detail, including patch analysis (Section 3.2), signature generator (Section 3.3), and matching engine (Section 3.4). The first challenge (Section 2.1) is solved in patch analysis, and the second challenge (Section 2.2) is solved in equation matching (Section 3.4.2).

3.1. System Architecture

Figure 4 shows that the P1OVD has four inputs, including unpatched sources, patched sources, reference binaries that are compiled from patched sources, and target binaries waiting to be checked. Since a large number of function-level binary similarity tools are currently available, e.g., Asm2Vec [16], we can obtain the address of the possibly vulnerable function in the target binary by finding out the function most similar to the vulnerable function, without requiring a symbol table.

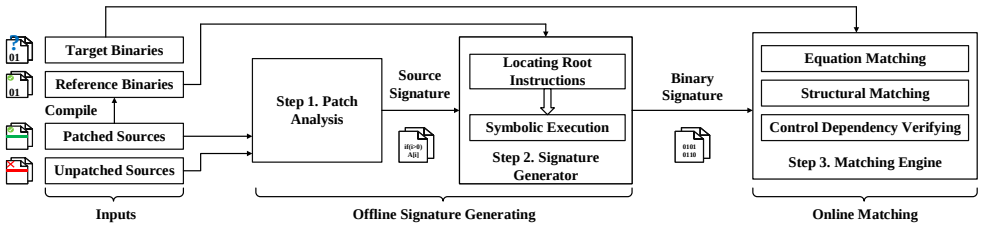

Figure 4. System Architecture.

P1OVD has three parts: patch analysis, signature generator, and matching engine. Patch analysis is designed for generating source signatures from the patched sources and unpatched sources. The generated source signatures are accurate and robust enough to overcome the first challenge. Then the signature generator maps the source signatures to binary signatures while keeping their accuracy and robustness. Patch analysis and signature generator are combined to generate binary-level signatures for out-of-bounds vulnerabilities. Finally, the matching engine searches vulnerabilities in the unknown target binaries according to the binary signatures. Especially, the novel patch signature matching algorithm matches the patch signatures accurately and robustly, while solving the second challenge.

3.2. Patch Analysis

In this step, we generate the signatures to represent vulnerabilities. As mentioned in Section 2.1, important code variants can be ignored when the vulnerability signatures incompletely contain vulnerability information or patch information. Inspired by the fact that both patch information and vulnerability information can increase the signature accuracy and root causes are widely used to evaluate vulnerability signatures [18], we define that out-of-bounds vulnerability signatures should mainly contain patch information and out-of-bounds vulnerability root cause information.

To obtain vulnerability root causes accurately, P1OVD utilizes a patch analysis tool, SID [22]. Patch analysis tools aim at removing the gap between patches and vulnerabilities. Especially, SID outperforms the statical approaches at the accurate out-of-bounds root causes locating. SID defines that the root cause of out-of-bounds vulnerabilities is memory access without proper bound checks. A branching statement, either an if statement or a loop statement that exists in the patch is regarded as a bound check. They are what out-of-bounds patches try to add or correct. Memory access always includes directly indexing arrays by subscripts or calling certain functions to access memory indirectly, which is the root cause of out-of-bounds vulnerabilities. P1OVD locates the memory access and the bound checks according to SID's security rules.

After obtaining patch information and vulnerability root cause information, P1OVD constructs a local PDG (program dependency graph), which is a subgraph of the function PDG starts at bound check and ends at memory access, linking a series of branching statements that are positioned between memory access and bound. The topology of such a local PDG reflects the relationship between patches and vulnerabilities. Compared to local CFG [20], this local PDG is more robust to code variants because compilation environments e.g., optimization levels can significantly change the CFG structures.

Example 1. *As Figure 3 shows, the if statement at line 7 is added by patch. So it is a bound check and is the start of local PDG. The variable i is used to index cmd->msg[i] at line 18. So line 18 is the memory access and is the end of local PDG. Finally, line 14 where the variable i compares with cmd->msg_len, which is important because the dissatisfaction of the bound check can make the function skip line 14 and exit directly. So line 14 is included in the local PDG and is the successor of the bound check and the predecessor of the memory access. In conclusion, we extract only three lines as a signature. With little unnecessary information and complete patch information, as well as vulnerability information, this signature can overcome code variants.*

3.3. Binary Signature Generator

Although the generated signatures are accurate, they are at the source level. Thus, in this step, we map the source signatures to the binary signatures by reference binaries, which are manually generated by compiling the patched sources with the *O0* optimization level to *x86_64* architecture while reserving the debugging information. P1OVD keeps the binary signatures in the form of local PDGs and only maps each node of the local PDGs from source-level to binary-level because the local PDGs remain the same even if the compilation environments change.

Theoretically, all instructions that correspond to the local PDG statement nodes can be part of the binary signature. However, Zhang et al. [20] announced that only a subset of instructions i.e., root instructions actually summarize the statement key behaviors, and the unnecessary instructions in the signatures can lead to mismatches. Hence, in this step, P1OVD accurately locates root instructions and uses the semantic information of root instruction to represent the statements.

3.3.1. Root Instructions Locating

Due to the significant difference between binary and C source codes, a statement that originally contains multiple instructions may even be divided into multiple basic blocks, during the compilation procedure. For example, an *if* statement with a logical operation, e.g., && or ||, will be separated into two multiple basic blocks. Hence, for each statement in the local PDG, P1OVD locates the root instructions accurately by taking line numbers, data dependency, variable names, and statement types into concern.

P1OVD first narrows the scope of possible root instructions by selecting the instructions corresponding to statement lines. This can be done with the help of debugging information from reference binaries.

Next, P1OVD narrows the scope of possible root instructions again by variable-based data dependency analysis because variables represent the behavior of the statement in most cases. For example, the vulnerable statement *cmd->msg[i]* contains two important variables *cmd* and *i* and they are combined to generate an out-of-bounds vulnerability. The variable names can be easily obtained by parsing source codes. However, when variables are parameters of operator *sizeof* they can be turned into a constant and disappear from binaries due to the preprocessing. For example, *sizeof(cmd->msg))* corresponds to the immediate number six in the binary. So P1OVD excludes all variables that are only used in the operator *sizeof*. After extracting variable names, P1OVD uses debugging information to map variable names to *rbp* related addresses on the stack because without optimization, the *GCC* compiler stores each local variable on the stack. Since each extracted variables are part of the original statements, the root instructions should data-depend on all extracted variables. Thus, P1OVD performs a data dependency analysis to exclude irrelevant instructions. We define an instruction data-depends on a certain variable if it directly uses the *rbp* related address or uses the result of another instruction that is data-dependent on the variable.

Finally, one statement can have multiple behaviors at the same time, while some of them are less important. For example, line 18 reads the memory and calls a function. But only reading the memory can cause the exception. Thus, P1OVD locates the root instructions that represent the key behaviors of the statements among the selected candidates. Bound checks and extra branching statements control the values of the program counters. Thus, they are compiled into PSW (program status word) writing instructions and branching instructions. Generally, they are positioned at the end of the basic blocks. We require the root instructions of bound checks are branching instructions because they reserve the important comparison operator information since out-of-bounds patches can only correct the comparison operators. But we require the root instructions of extra branching statements are PSW writing instructions. As mentioned in Section 4.2.3, the results of branching instructions can be simplified. Further, extra branching statements do not need comparison operator information. Finally, there are two kinds of memory access, including function

calling and array indexing and they trigger exceptions through load or store instructions in the current functions or the callees. Hence, such behaviors are stored in the function call instructions and load or store instructions. In conclusion, Table 1 shows the type of root instruction we required.

Table 1. Mapping source statement to binary AST.

Statement	Root Instruction Type	AST
Bound Check	Branching Instruction	Branching Condition
Memory Access (Call Function)	Function Call Instruction	Access Expression (Callee and All Function Arguments)
Memory Access (Index Array)	Load or Store Instruction	Access Expression (Memory Adress)
Extra Branching Satement	Branching Instruction	PSW Write Arguments

3.3.2. Symbolic Execution

In this section, P1OVD generates sufficient information for root instructions so that they can represent the vulnerabilities and patches. Researchers [8,9,13,19,20] have demonstrated that symbolic execution results i.e., ASTs can robustly represent the operands of instructions. Thus, P1OVD symbolically executes the reference functions from their entries and extracts ASTs forass root instructions. Besides, as Table 1 shows, since the operands of different root instructions are different, P1OVD generates different ASTs for them. The extracted ASTs can represent the vulnerabilities and patches. For example, P1OVD extracts *[arg + 6] <= 6* and *![arg + 6] <= 6* for statement *cmd->msg_len>sizeof(cmd->msg)*. The *cmd* is function argument and AST uses *arg* to represent it. Then it load the member *msg _len* with offset six corresponding to *[arg + 6]*. Finally, it is compared to the constant number six and forks the basic block, as ASTs indicate. In conclusion, all statements in the source local PDGs are replaced with ASTs.

Example 2. *As Figure 5 shows. The root cause contains two variables, named cmd and i. P1OVD maps the line to instructions first. Then among these instructions, P1OVD finds that i which is located at rbp-0x20 is used at 0x409b4, while cmd is used at 0x409b8. Since the memory operation at 0x409ba uses both variables to read the memory, P1OVD thinks it is a root instruction. Moreover, by symbolic execution we generate mem_read(arg) etc. to represent the access expression it read. Similarly other nodes of local PDG can be mapped to binary-level.*

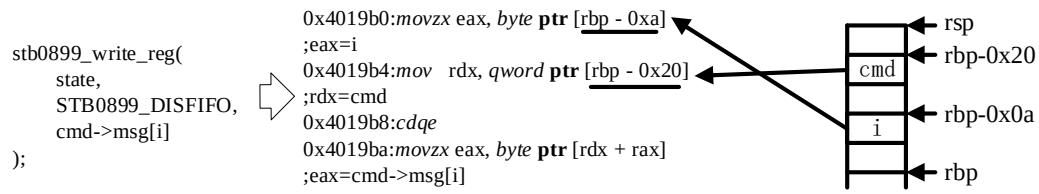

Figure 5. Locating instructions by mapping line to instructions and variable name to stack.

3.4. Matching Engine

The matching engine can judge if an unknown binary is vulnerable or not by using the binary signatures generated from reference binaries. For one binary it has four kinds of output: not vulnerable, patched, vulnerable, unable to judge. Before actually starting to match the vulnerabilities, we use the code similarity to find out the functions that may contain the vulnerabilities in the binaries and use symbolic execution to extract all ASTs of the target functions. Then we start the vulnerability matching.

We find access expressions and PSW write arguments by structural matching and find branching conditions by equation matching. Because the structural matching is faster than the equation matching but dissatisfies the high accuracy required by branching conditions. Finally, we verify the control dependencies using local PDGs.

3.4.1. Structural Matching

Structural matching finds operations with similar semantics to out-of-bounds access expressions or PSW write arguments in the target binaries by calculating the graph similarity of two ASTs. Empirically, we have found that ASTs with the same semantics may have subtle differences when extracted from binary functions of different compilation environments. For example, both *a+((b+1)«1)* and *a+2+(b«1)* can be found in binaries when statement *a[b+1]* appears in the source code. Therefore, we do not require the ASTs to be structurally identical but structurally similar. Thus, the edit-distance-based graph similarity can better reflect the similarity of ASTs and we compare the graph similarity with a predefined threshold to determine whether two ASTs are similar.

3.4.2. Equation Matching

The main task of equation matching is to find a branching condition the same as the patched bound check. However, as mentioned in Section 2.2, both structural matching and strict matching cannot overcome the second challenge.

Fiber [20] matches the same kind of nodes, e.g., immediate numbers with different algorithms according to their positions in the ASTs. Thus, we infer that different parts of branching conditions should be matched according to different precisions. We define a subtree in the AST as a data object if its root node is a memory read operation or it only contains one node that is a function return value or a parameter. Additionally, a data object should not be a subtree of another data object. After extracting data objects, remain the boolean expressions. Empirically, we learn that under different compilation environments, data objects have similar structures, while boolean expressions preserve fixed semantics. So P1OVD matches them according to their structures and semantics correspondingly.

After extracting data objects by traversing the branching conditions, P1OVD uses the structural matching presented in Section 3.4.1 to generate data object pairs, one from the target branching condition and one from the reference branching condition. The matched data object pairs in two ASTs are replaced with the same symbol. In the case that two matched data objects have different sizes, e.g., one is 64-bits long, and the other is 32-bits, P1OVD defines a symbol in the shorter size and replaces two ASTs with this symbol. To satisfy the length requirement of the longer tree, P1OVD pads zero to the left of the defined symbol. P1OVD replaces the data objects to ensure the two bool expressions have identical symbol sets and can be used for solving.

Next, we accurately solve [23] the boolean expressions. Since the boolean expressions are semantically identical, only equal or opposite boolean expressions are matched. In other words, given two boolean expressions $Expr_1$ and $Expr_2$, they are considered matched if only one of $Expr_1 = Expr_2$ or $Expr_1 =! Expr_2$ can be solved.

Example 3. *As Figure 6 shows, to match the [arg + 6] <= 6 and ![arg + 6] > 6, P1OVD first extracts data objects from them. The extracted data objects are both [arg + 6] and they are structurally similar obviously. So P1OVD replaces them with a single symbol x in their original AST. Then P1OVD checks the solving possibility of conditions* $(x \leqslant 6) =! (x > 6)$ *and* $(x \leqslant 6) =!! (x > 6)$. *Results show* $x = 0$ *satisfies the first constrain and the second constrain will never be satisfied, thus P1OVD finds that the two branching conditions are equal. So, in conclusion, the two ASTs represent the same branching condition. At the same time, both strict matching and graph-similarity-based matching cannot distinguish such changes in ASTs.*

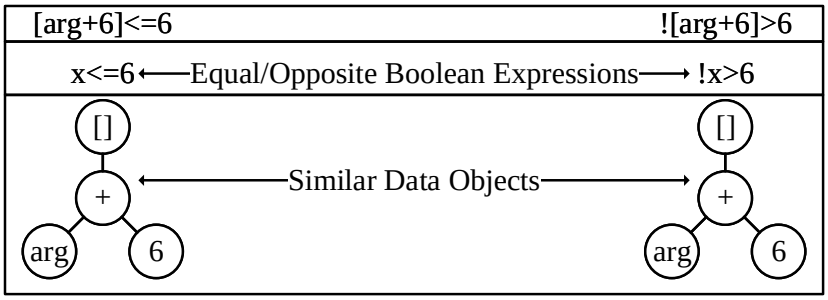

Figure 6. Two-step matching algorithm.

3.4.3. Verify Control Dependency

In this step, P1OVD matches the local PDG topologies by verifying the edge i.e. control dependency of their nodes. This step is to ensure that the bound checks actually control the memory access. Since even if both bound checks and memory access appear in the same function, vulnerabilities can still appear. For example, in Linux Kernel commit 1fa2337, the patch only moves the check of *d->msg_len* forward to secure the *d->msg[i]* used in function *printk*. P1OVD can infer statement A controls statement B from two cases:

- A is a loop statement. The loop structure is often reordered by optimization, and P1OVD requires every trace that passes the B twice or more to contain an A between every neighbor B.
- A is an *if* statement. Since the *if* statement may be in a loop, P1OVD requires each trace that reaches the B from the function entry to pass A.

4. Evaluation

We have developed P1OVD with 1200 lines of python code on top of open source libraries Angr [24], joern [25], and pyelftool [26]. P1OVD supports *aarch64*, *x86_64*, and *x86_32* target architectures as well as, *O0* and *O2* optimization levels. In this section, we first evaluate P1OVD in terms of accuracy (Section 4.2) and efficiency (Section 4.3), and then we compare P1OVD with other tools in terms of the overall performance (Section 4.4) and the effectiveness of signature generation (Section 4.5), and AST matching (Section 4.6).

4.1. Datasets

We evaluate our tool based on Linux Kernel because not only is Linux Kernel widely used [27], but also the out-of-bounds vulnerabilities in the Linux Kernel are widely analyzed [22]. Our tool has four inputs, including unpatched source code, patched source code, patched binary, and target binary. Thus, we collect three datasets.

4.1.1. Source Codes

Source codes include the patched and unpatched source codes. To ensure that the patch analysis can successfully operate, we use 30 out-of-bounds patches listed in the appendix of SID [22] while two of them are patches of CVE-2017-18379 and CVE-2019-15926. We exclude some patches that are too old that cannot be successfully compiled.

4.1.2. Reference Binaries

The reference binaries are obtained by compiling the patched source code. Still, most of the out-of-bounds vulnerabilities are located in the Linux Kernel optional modules, which are difficult to trigger by the default compilation options. We manually adjust the compilation options for each out-of-bounds patch to satisfy the constraints and apply *GCC* to compile the patched source codes while reserving the debugging information.

4.1.3. Target Binaries

The target binaries are used to prove that the P1OVD can fight against binaries built with non-standard configurations from customized codes on multiple architectures. To obtain target binaries, we compile source code into 620 different binaries. The collected binaries vary from three aspects, including versions, optimization levels, and target architectures.

We think the first way to customize source codes is to compile the source codes of different versions. When we refer to versions, we do not mean the software release version e.g., "Linux-5.0-y" because a newer software release version does not change the vulnerable function sometimes. We define a new version according to the vulnerable functions. For a vulnerable function, we regard all commits that change this function as versions and divide these versions into three categories, including not vulnerable versions, vulnerable versions, and patched versions. A function had no vulnerability at first and we think the functions in these versions are not vulnerable. And then, at a notable point, the vulnerable memory access began to appear in the function. We think these functions are vulnerable. After the vulnerability was discovered, the repository maintainer corrected the bound check using a patch. We consider these functions are patched. Totally, we obtain 104 different versions from Linux Kernel "master" branch and we classify these versions manually. Second, optimization levels are usually customized and *O0* and *O2* are the most commonly used optimization. Moreover, the reference binaries are generated based on *O0*. Thus, we evaluate P1OVD on *O0* and *O2* optimization levels. Finally, different manufacturers can build sources for different target architectures. We also evaluate P1OVD on three architectures including *x86_64*, *x86_32*, and *aarch64*. The *x86_32* uses 32-bit addresses while *aarch64* and *x86_64* use 64-bit addresses while *aarch64* has an instruction set different from *x86_64* and *x86_32*. After obtaining 620 binaries, we generate ground truths for them according to their versions because the changes of compilation environments do not affect the vulnerability detection results.

4.2. Accuracy

We evaluate the P1OVD based on the multiclass classification problem. Only the outputs of P1OVD that correctly predict both vulnerability and patch are considered correct. We use precision, recall, F-1 score, and accuracy to measure the accuracy of P1OVD. The four evaluation metrics are defined in Equation (1). P1OVD first generates signatures based on *O0* optimization and *x86_64* architecture and then scans the target binaries with the signatures.

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \\ F_1\ Score &= \frac{2 * Precision * Recall}{Precision + Recall} \\ Accuracy &= \frac{Correct}{Total} \end{aligned} \tag{1}$$

Table 2 shows the accuracy of P1OVD. Each row represents the compilation environments, and each column stands for the evaluation metrics of three version categories. P1OVD obtains an accuracy of 83.06%. And we manually analyzed the false predict cases and summarized the following four reasons, while the first three are common challenges for symbolic execution-based tools.

Table 2. Vulnerability Detection Accuracy Test.

Compilation Environment		Recall			Precision			F-1 Score			Accuracy
		NV	V	P	NV	V	P	NV	V	P	
O0	*aarch64*	100.00%	84.09%	86.00%	45.45%	100.00%	97.73%	0.62	0.91	0.91	86.54%
	x86_64	80.00%	95.45%	100.00%	88.89%	95.45%	98.00%	0.84	0.95	0.99	96.12%
	x86_32	80.00%	63.64%	79.59%	80.00%	100.00%	97.50%	0.80	0.78	0.88	72.82%
O2	*aarch64*	100.00%	84.09%	80.00%	47.62%	90.24%	97.56%	0.65	0.87	0.88	83.65%
	x86_64	80.00%	86.36%	95.92%	80.00%	97.44%	97.92%	0.80	0.92	0.97	90.29%
	x86_32	80.00%	61.36%	73.47%	66.67%	93.10%	97.30%	0.73	0.74	0.84	68.93%
All		86.67%	79.17%	85.81%	61.90%	95.87%	97.69%	0.72	0.87	0.91	83.06%

NV stands for not vulnerable. V stands for vulnerable. P stands for patched.

4.2.1. Function Inline

Function inline contributes most of the false rates. A function with an inline tag may not be inline during the compilation procedure. It is influenced by many factors e.g., the optimization level, and target architecture. For example, function *nvmet_fc_getqueueid* called by the function *nvmet_fc_find_target_queue* is inlined when compiled with *aarch64* and *O0*. However, function *nvmet_fc_getqueueid* is not inlined when compiled with *x86_64* and *O0*. Whether the callee is inline affects the extracted signature through symbolic execution and function inline is also the key reason why P1OVD has worse performance on *aarch64* architecture or *O2* optimization level, compared to *x86_64* and *O0*.

4.2.2. Conditional Execution Instructions

Conditional execution instructions are instructions that select whether to perform operations based on the PSW, e.g., *CSEL*. When multiple conditional branches occur continuously, *aarch64* optimizes the efficiency by replacing branch instructions with conditional execution instructions since branch instructions slow the assembly. Function *qxl_clientcap_ioctl* in commit 62c8ba7 compares *qdev->pdev->revision* with 4 and *byte* with 58 continuously, and only one branch instruction to deal with the exception for them. Since we used the AST of branching conditions for boundary check, the inability to find the correct branching conditions in the binary led to false positives.

4.2.3. Simplified Expression

P1OVD will output a false result when a patch changes the loop bound. For example commit 43622021d2e2b changes operator <= to < in the statement *for(j=0; j<HID_MAX_IDS; j++)*. Every time the executor compares the *j* and *HID_MAX_IDS*, the value of *j* is a constant. As a result, Angr will automatically optimize a boolean expression containing only constants to True or False. This prevents us from extracting branching conditions correctly and this is the key reason why the signature cannot ensure all predictions are correct on *x86_64* and *O0*.

4.2.4. Structure Dissimilar

Function *hid_register_report* in commit 43622021d, indexes an array through *report_enum->report_id_hash[id]* while *report_enum* is calculated by *device->report_enum+type*. However, all the structs have pointer members, which means that when the system address lengths change the sizes of structs change. Although the difference between the two integers is small, optimizing codes by replacing multiplication with an arithmetical left shift is often used in the addressing process, resulting in structural differences. As a result, *x86_32* has the worst performance. We believe that graph embedding is a feasible solution to this type of problem.

4.3. Performance

Experimented on intel-i7-8700 and 12GB RAM, Table 3 records the time consumption of P1OVD from three aspects of patch analysis, signature generator, and matching engine. Patch analysis and signature generator are used to extract binary signatures offline. We calculate the average time used to generate a signature for one patch. The matching engine is used to determine whether a target binary is vulnerable or not. To solve the situation that different vulnerable functions have different numbers of versions and better reflect the time consumption, we first calculate the average time used for finding a certain vulnerability in various binaries, then we calculate the overall average vulnerabilities finding times.

Table 3. Vulnerability Detection Performance Test.

Step		Total Time	Number	Average
Offline	Patch Analyze	470.68 s	30	15.69 s
	Signature Generate	99.90 s	30	3.33 s
Online	Match	108.07 s	30	3.60 s

4.4. Accuracy Comparison with Vulnerability Detection Tools

In this section,we evaluate the accuracy of P1OVD by comparing it with the state-of-the-art vulnerability detection tools. We choose BinXray [17] and Asm2Vec [16] as references because they are both open-sourced and BinXray and Asm2Vec are the state-of-the-art patch-level and function-level vulnerability detection tools. We compare P1OVD, Asm2Vec, and BinXray from four aspects, including precision, recall, F-1 score, and accuracy. For one function, Binxray only has two kinds of outputs vulnerable or patched. So we require P1OVD to predict if the functions are vulnerable or not. Thus, we relabel the patched binaries as not vulnerable binaries. At the same time, Asm2Vec ranks the possibly vulnerable functions. So we think the function in the unknown target binary that has the highest similarity to the vulnerable function in the reference binary is vulnerable.

Table 4 shows the results of the comparison. Asm2Vec, a function-level tool, cannot distinguish three versions well and consider them are all vulnerable, which result in high false positive. Meanwhile, BinXray assumes that vulnerabilities exist when patches disappear. However, bound checks and memory access can both disappear because of code customization. When BinXray cannot detect the patches, it mistakenly believes that the vulnerabilities exist. P1OVD achieves the highest precisions due to its vulnerability signature containing both vulnerability root cause information and patch information. However, P1OVD cannot ensure that all binaries predicted safe are accurately safe. When the patch changes the loop boundary, the simplified expressions described in Section 4.2.3 can cause the results of symbolic execution to contain too little information and P1OVD cannot distinguish the patched version from the unpatched version. However, Binxray uses the patch codes in the binaries as signatures, which works well when the target binary and reference binary are under the same compilation environment.

4.5. Effectiveness of Vulnerability Signatures

In this step, we evaluate the effectiveness of our vulnerability signatures by comparing the source signatures of P1OVD with ReDeBug [3] based on the source dataset. Because ReDeBug is a widely used open-source source-level unpatched buggy code detection tool. As Table 5 shows, compared to ReDeBug, P1OVD extracts fewer lines but our signatures contain more vulnerability root cause information and patch information, which means P1OVD can generate more accurate vulnerability signatures. This is because ReDeBug only pays attention to the lines close to the patches. On the contrary, P1OVD focuses more on the statements that control the function security (vulnerability root cause information and patch information), by which P1OVD overcomes the first challenge (Section 2.1). For example, ReDebug always includes the error handling of bound checks because they are

added by patches. Meanwhile, P1OVD thinks they are widely customized [19] and excludes them from signatures.

Table 4. Comparing P1OVD with BinXray and Asm2Vec.

Tool	Compilation Environment		Recall		Precision		F1-score		Accuracy
			Not Vulnerable	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable	Vulnerable	
P1OVD (Graph Similarity)	*O0*	*aarch64*	60.00%	84.09%	83.72%	61.67%	0.70	0.71	70.19%
		x86_64	50.85%	95.45%	93.75%	59.15%	0.66	0.73	69.90%
		x86_32	52.54%	63.64%	93.94%	62.22%	0.67	0.63	57.28%
	O2	*aarch64*	56.67%	84.09%	85.00%	58.73%	0.68	0.69	68.27%
		x86_64	49.15%	86.36%	93.55%	57.58%	0.64	0.69	65.05%
		x86_32	50.85%	61.36%	90.91%	60.00%	0.65	0.61	55.34%
P1OVD (Two Step)	*O0*	*aarch64*	98.33%	84.09%	89.39%	100.00%	0.94	0.91	92.31%
		x86_64	96.61%	95.45%	96.61%	95.45%	0.97	0.95	96.12%
		x86_32	81.36%	63.64%	96.00%	100.00%	0.88	0.78	73.79%
	O2	*aarch64*	93.33%	84.09%	90.32%	90.24%	0.92	0.87	89.42%
		x86_64	94.92%	86.36%	96.55%	97.44%	0.96	0.92	91.26%
		x86_32	77.97%	61.36%	93.88%	93.10%	0.85	0.74	70.87%
BinXray	*O0*	*x86_64*	81.36%	95.45%	100.00%	84.00%	0.90	0.89	87.38%
Asm2Vec	*O0*	*x86_64*	5.08%	95.45%	60.00%	42.86%	0.09	0.59	43.69%
	O2	*x86_64*	16.95%	79.55%	52.63%	41.67%	0.26	0.55	43.69%

Table 5. Comparing P1OVD with ReDeBug.

	P1OVD	ReDeBug
Bound Check Coverage	100%	100%
Memory Access Coverage	100%	50.94%
Used Lines	2.80	6.87

4.6. Effectiveness of Two-Step AST Matching Algorithm

In this section, we evaluate the effectiveness of our two-step AST matching algorithm by comparing it with the graph-similarity-based AST matching algorithm, because the graph-similarity-based AST matching algorithm is the most widely used AST matching algorithm among vulnerability detection tools [8,9,13] and is used to enhance the robustness of Fiber [20]. To compare with it, we generate a new version of P1OVD by replacing our two-step equation matching component with the graph similarity matching. Table 4 shows that the modified P1OVD has more false positives, which demonstrates that the two-step AST matching algorithm can address the second challenge (Section 2.2). This is because the graph-similarity-based matching can only distinguish action-related nodes and condition-related nodes [13] and cannot distinguish the in-node changes. Moreover, when only operands are different, the graph-similarity-based matching algorithm remains unaware. The two-step matching algorithm splits the AST into two parts. If such changes happen in the data objects, P1OVD ignores them. On the contrary, if boolean expression semantics are changed, P1OVD is warned by the solver.

4.7. Limitation

P1OVD analyzes the out-of-bounds patches based on security rules of SID [22], which leads to one limitation. Although SID outperforms patch analysis tools, it can neither analyze the patches that involve multiple functions nor understand out-of-bounds patches that do not correct bound checks e.g., extending the array size. Thus, P1OVD cannot successfully detect all out-of-bounds vulnerabilities. To address this problem, we are considering replacing SID with other dynamic patch analysis tools e.g., PatchScope [28].

Further, memory access is a common root cause of many kinds of memory-centric vulnerabilities, e.g., use after free and correcting missing or wrong checks before the memory access can also be the key behavior of security patches. We can polish bound checks to generalize the vulnerabilities that P1OVD can detect.

Finally, the performance of P1OVD in 32-bit architecture is not as good as in 64-bit architecture. This is mainly due to the structural changes related to address length (Section 4.2.4). Thus, we try to calculate the similarity scores of two ASTs with a more robust algorithm. Many works [16,29] train neural networks to calculate the graph similarity of CFG or PDG and we think these approaches can be adapted to AST similarity calculation.

5. Related Work

This article is closely related to four branches of study, function-level 1-day vulnerability detection, patch-level 1-day vulnerability detection, patch presence test, and patch analysis. In the following four sections, we give a brief review of the works that lead to our own.

5.1. Function-Level 1-Day Vulnerability Detection

At present, many studies focus on detecting vulnerabilities in source files and binary files by judging whether the target function is similar to the vulnerable function. Usually, function-level 1-day vulnerability detection tools extract features from reference sources or binaries and match them with special algorithms.

Early algorithm using normalized source codes [7], ASTs [5], PDG [4,6], etc. to comprehensively represent the whole source vulnerable function. However, they cannot detect the 1-day vulnerabilities in binaries, due to the lack of binary semantic information.

DiscovRE [10] tries to solve this problem by extracting numerical features from the basic blocks and CFG structural features. Introduced by Genius [12], neural networks use vectors to better represent the function feature, e.g., numerical and structural information [14,15], assembly codes [16]. Some tools use tree-liked formulas to represent basic blocks [8], function IO behaviors [9], or even the high-level function semantic information [13]. These function-level 1-day vulnerability detection tools take the whole vulnerable functions as the vulnerability signatures and their extremely large scope of function-level signatures cause the first challenge (Section 2.1). When small code variants involve patches or vulnerability root causes, the function-level signatures bring much useless information and cannot give correct predictions.

5.2. Patch-Level 1-Day Vulnerability Detection

During the last decade, researchers use patches to improve the function-level 1-day vulnerability detection, which is called patch-level vulnerability detection. Early tools [3,17,30,31] believe the missing patch-added codes are the root causes of vulnerabilities. By using normalized and tokenized patches [3], patch sensitive matching algorithms [30], LSTM-embedded code vectors [31], patch modified traces [17], they enhance the patch searching rather than vulnerability searching.

Li et al. [32] think the patch-removed code is vulnerable and using concolic testing to verify the clone of vulnerable code. MVP first [18] announced that the patch and the corresponding vulnerability have different information and it thinks the deleted codes are vulnerable and the added codes are patches. So it uses CPG (code property graphs) to slice vulnerability-related codes as vulnerability signatures and patch-related codes as patch signatures. Further, researchers manually evaluate MVP's failures by comparing signatures with vulnerability root causes in case studies. Although the signatures containing both patch information and vulnerability information improve source-level 1-day vulnerability detection, they contain too much unnecessary information for binary-level vulnerability detection, which leads to the first challenge (Section 2.1).

5.3. Patch Presence Test

The concept of patch presence tests is first proposed by Zhang et al. [20]. Its main purpose is to accurately confirm whether a binary contains a particular patch or not, while we try to find vulnerabilities rather than patches. Other patch presence tests improve the Fiber [20] in terms of the diversifying source program languages [33] and robustness [19], or polish it with other dynamic tools [34].

Vulnerabilities detections need high accuracy and robustness, which is similar to patch present tests. So, we have a deep look into the Fiber and Pidff [19]. They both locate basic blocks corresponding to the patches in the patched binaries. Later, by symbolic execution, they extract the AST-shaped results of these basic blocks as signatures. However, they match their signatures with different methods. While Fiber proposes a strict operand-based patch matching algorithm with little relaxations (inter-changeable operators, address-related immediate numbers), PDiff proposed a robust but less accurate graph similarity-based matching algorithm. Unfortunately, neither of them can accurately and robustly match branching conditions in downstream binaries, which leads to the second challenge (Section 2.2). In Section 3.4.2 we benefit from both and propose a novel two-step matching algorithm.

5.4. Patch Analysis

Patch analyses are used to understand how security patches fix the vulnerabilities. At first, Corley et al. [35] links between bugs and patches while requiring an issue tracking system. Later, Spain [36] proposed binary-level patch patterns to detect unexplored vulnerabilities but limited by lacking high-level semantic information, it cannot fully understand out-of-bounds patches. SID [22] outperforms other statical tools at the accurate out-of-bounds root causes locating, by utilizing symbolic execution. PatchScop [28] dynamically analyzes the patches and gets the highest accuracy although it requires POCs (Proof of Concepts). In this work, we use the state-of-the-art statical tool, SID to locate the root causes of out-of-bounds vulnerabilities.

6. Conclusions

In this work, we have had a deep look into the 1-day vulnerability detection and identified two challenges introduced by code variants, including vulnerability signature and patch signature matching. To solve the two challenges, we have proposed P1OVD, an accurate detection method for 1-day out-of-bounds vulnerabilities in downstream binaries using patches. P1OVD analyzes the patch to get accurate vulnerability signature, generates binary signatures using debugging information and symbolic execution, and accurately matches the signatures, especially branching condition. Experiments have demonstrated that P1OVD can generate accurate and robust vulnerability signatures and match the signatures accurately. Addressing the above two challenges allows P1OVD to resist interference from code customization, non-standard building configurations, and to detect 1-day out-of-bounds vulnerabilities on multiple architectures more accurately than existing tools.

Author Contributions: Conceptualization, methodology, validation, evaluating, and writing H.L.; writing—review and editing, funding acquisition, project administration, resources D.H.; writing—review and editing S.C.; writing-review and editing X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Natural Science Foundation of China, China (Grant No. U1936120), the University Grants Committee of the Hong Kong Special Administrative Region of China (City U11201421), and the Basic Research Program of State Grid Shanghai Municipal Electric Power Company (52094019007F).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract Syntax Tree
PDG	Program Dependency Graph
PSW	Program Status Word
CFG	Control Flow Graph
POC	Proof of Concepts

References

1. Peng, J.; Li, F.; Liu, B.; Xu, L.; Liu, B.; Chen, K.; Huo, W. 1dVul: Discovering 1-Day Vulnerabilities through Binary Patches. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–27 June 2019; pp. 605–616. [CrossRef]
2. Insights into the 2.3 Billion Android Smartphones in Use Around the World. Available online: https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/ (accessed on 12 December 2021).
3. Jang, J.; Agrawal, A.; Brumley, D. ReDeBug: Finding unpatched code clones in entire os distributions. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 48–62.
4. Pham, N.H.; Nguyen, T.T.; Nguyen, H.A.; Nguyen, T.N. Detection of recurring software vulnerabilities. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 20–24 September 2010; pp. 447–456.
5. Yamaguchi, F.; Lottmann, M.; Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 359–368.
6. Zou, D.; Qi, H.; Li, Z.; Wu, S.; Jin, H.; Sun, G.; Wang, S.; Zhong, Y. SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Bonn, Germany, 6–7 July 2017; pp. 325–344.
7. Kim, S.; Woo, S.; Lee, H.; Oh, H. Vuddy: A scalable approach for vulnerable code clone discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 595–614.
8. Pewny, J.; Schuster, F.; Bernhard, L.; Holz, T.; Rossow, C. Leveraging semantic signatures for bug search in binary programs. In Proceedings of the 30th Annual Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014; pp. 406–415.
9. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724.
10. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; pp. 58–79.
11. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
12. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. 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, Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376.
13. Feng, Q.; Wang, M.; Zhang, M.; Zhou, R.; Henderson, A.; Yin, H. Extracting conditional formulas for cross-platform bug search. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 346–359.
14. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 896–899.
15. Liu, B.; Huo, W.; Zhang, C.; Li, W.; Li, F.; Piao, A.; Zou, W. αdiff: Cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier France, 3–7 September 2018; pp. 667–678.
16. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 472–489.
17. Xu, Y.; Xu, Z.; Chen, B.; Song, F.; Liu, Y.; Liu, T. Patch based vulnerability matching for binary programs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, 18–22 July 2020; pp. 376–387.
18. Xiao, Y.; Chen, B.; Yu, C.; Xu, Z.; Yuan, Z.; Li, F.; Liu, B.; Liu, Y.; Huo, W.; Zou, W.; et al. MVP: Detecting Vulnerabilities Using Patch-Enhanced Vulnerability Signatures. Available online: https://chenbihuan.github.io/paper/sec20-xiao-mvp.pdf (accessed on 12 December 2021).

19. Jiang, Z.; Zhang, Y.; Xu, J.; Wen, Q.; Wang, Z.; Zhang, X.; Xing, X.; Yang, M.; Yang, Z. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, 9–13 November 2020; pp. 1149–1163.
20. Zhang, H.; Qian, Z. Precise and accurate patch presence test for binaries. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 887–902.
21. Linux Kernel. Available online: https://github.com/torvalds/linux (accessed on 12 December 2021).
22. Wu, Q.; He, Y.; McCamant, S.; Lu, K. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2020.
23. Z3Prover/z3: The Z3 Theorem Prover. Available online: https://github.com/Z3Prover/z3 (accessed on 12 December 2021).
24. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016.
25. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 590–604.
26. Parsing ELF and DWARF in Python. Available online: https://github.com/eliben/pyelftools (accessed on 12 December 2021).
27. Hall, C. Survey Shows Linux the Top Operating System for Internet of Things Devices. Available online: https://www.itprotoday.com/iot/survey-shows-linux-top-operating-system-internet-things-devices (accessed on 12 December 2021).
28. Zhao, L.; Zhu, Y.; Ming, J.; Zhang, Y.; Zhang, H.; Yin, H. Patchscope: Memory object centric patch diffing. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, 9–13 November 2020; pp. 149–165.
29. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Tan, H.B.K. Bingo: Cross-architecture cross-os binary search. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 678–689.
30. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Qi, H.; Hu, J. VulPecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, 5–9 December 2016; pp. 201–213.
31. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv* **2018**, arXiv:1801.01681.
32. Li, H.; Kwon, H.; Kwon, J.; Lee, H. A scalable approach for vulnerability discovery based on security patches. In Proceedings of the International Conference on Applications and Techniques in Information Security, Melbourne, Australia, 26–28 November 2014; pp. 109–122.
33. Dai, J.; Zhang, Y.; Jiang, Z.; Zhou, Y.; Chen, J.; Xing, X.; Zhang, X.; Tan, X.; Yang, M.; Yang, Z. BScout: Direct Whole Patch Presence Test for Java Executables. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 1147–1164.
34. Sun, P.; Garcia, L.; Salles-Loustau, G.; Zonouz, S. Hybrid firmware analysis for known mobile and iot security vulnerabilities. In Proceedings of the 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Valencia, Spain, 29 June–2 July 2020; pp. 373–384.
35. Corley, C.S.; Kraft, N.A.; Etzkorn, L.H.; Lukins, S.K. 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, Waikiki, HI, USA, 23 May 2011; pp. 31–37.
36. Xu, Z.; Chen, B.; Chandramohan, M.; Liu, Y.; Song, F. Spain: Security patch analysis for binaries towards understanding the pain and pills. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 462–472.