Detecting API Post-Handling Bugs Using Code and Description in Patches

Miaoqian Lin[1,2] Kai Chen[1,2*] Yang Xiao[1,2]

[1]{SKLOIS[†], CAS-KLONAT[‡], BKLONSPT[§]}, Institute of Information Engineering, Chinese Academy of Sciences, China

[2]School of Cyber Security, University of Chinese Academy of Sciences, China

{linmiaoqian, chenkai, xiaoyang}@iie.ac.cn

Abstract

Program APIs must be used in accordance with their specifications. API post-handling (APH) is a common type of specification that deals with APIs' return checks, resource releases, etc. Violation of APH specifications (aka, APH bug) could cause serious security problems, including memory corruption, resource leaks, etc. API documents, as a good source of APH specifications, are often analyzed to extract specifications for APH bug detection. However, documents are not always complete, which makes many bugs fail to be detected. In this paper, we find that patches could be another good source of APH specifications. In addition to the code differences introduced by patches, patches also contain descriptions, which help to accurately extract APH specifications. In order to make bug detection accurate and efficient, we design API specification-based graph for reducing the number of paths to be analyzed and perform partial path-sensitive analysis. We implement a prototype named APHP (API Post-Handling bugs detector using Patches) for static detection of APH bugs. We evaluate APHP on four popular open-source programs, including the Linux kernel, QEMU, Git and Redis, and detect 410 new bugs, outperforming existing state-of-the-art work. 216 of the bugs have been confirmed by the maintainers, and 2 CVEs have been assigned. Some bugs have existed for more than 12 years. Till now, many submitted patches have been backported to long-term stable versions of the Linux kernel.

1 Introduction

Application Programming Interfaces (APIs) are provided for developers to avoid code duplication and improve code maintainability. In this way, developers can enjoy the rich functionalities of APIs without knowing their detailed implementation. Relatively, developers should be cautious when using them. APIs should conform to the specifications in documents, such as checking parameters before calling APIs and following the orders of API calls. Otherwise, serious security issues could occur, e.g., memory corruption and denial of service. API Post-Handling (*APH*) is a common type of specification. Specifically, after a particular API call, developers need to execute the corresponding *post-operation* (e.g., return checks and memory release). For example, Figure 1 shows the patch for a memory leak vulnerability caused by improper post-handling for a *target API* `platform_device_alloc` (CVE-2019-18813) [6], with a severity score of 7.5 (High). When `platform_device_add_properties` fails at line 7, the variable `dwc->dwc3` allocated at line 3 is not freed via the post-operation `platform_device_put`. By triggering this failure repeatedly, attackers can launch a denial of service attack.

```
// linux/drivers/usb/dwc3/dwc3-pci.c (9bbfcee)
01 static int dwc3_pci_probe(struct pci_dev *pci, ...)
02 {
03     dwc->dwc3 = platform_device_alloc("dwc3", ...);
04     if (!dwc->dwc3)
05         return -ENOMEM;
06
07     ret = platform_device_add_properties(dwc->dwc3, p);
08     if (ret < 0)
09-        return ret;
10+        goto err;
11
12 err:
13     platform_device_put(dwc->dwc3);
14     return ret;
15 }
```

Figure 1: Patch for CVE-2019-18813.

Due to the diversity of API usage, it is challenging to detect *improper API post-handling* (aka, APH bug). Although various approaches have been proposed, they usually rely on API documents [30, 36, 61] or comparable source code [27, 29] to infer the APH specifications and check whether the specifications are consistent with the code implementation. Unfortunately, the quality of documents may not be good enough to extract specifications automatically. Sometimes, APH specifications are missing, making specification extraction impossible. For example, the API

*Corresponding author

†State Key Laboratory of Information Security, IIE, CAS

‡Key Laboratory of Network Assessment Technology, CAS

§Beijing Key Laboratory of Network Security and Protection Technology

`platform_device_alloc` does not clearly state to use the post-operation `platform_device_put`. Extracting specifications from source code is also challenging since the correct use of the API may not be found.

Alternatively, we find that bug patches can be a good source of APH specifications. The patched code indicates the correct use of the API and the suitable post-handling code. Once the specifications are summarized, they can be used to detect APH bugs that use the same API. We also note that previous studies have utilized bug patches to detect new bugs (e.g., through clone detection [16, 20, 53]); however, they are limited to similar bugs. The code should be similar so that similar bugs can be detected. However, even for the same API, its uses could be very diverse, which prevents previous patch-based methods from detecting them.

Our approach. In this paper, we present APHP (API Post-Handling bugs detector using Patches), a static framework for detecting APH bugs using patches. APHP automatically obtains APH specifications from patches and uses them to detect APH bugs in a target program, without requiring similarity to the buggy program. APHP has two key components: specification extraction and APH bug detection. Extracting specifications involves identifying the target APIs and corresponding post-operations from the patched code. However, this is not straightforward since both of them may not appear in code differences. Patches could use the `goto` statement to divert the control flow to existing code for bug fixes (e.g., Figure 1). We address this by comparing the path differences between the original and patched code and identifying candidate target APIs and post-operations. To accurately locate the target APIs, we leverage patch descriptions, which typically emphasizes the target APIs to explain patch design.

Subsequently, APHP detects APH bugs by checking the consistency between the code and the extracted specification. To enhance precision, we characterize the APH specification using path conditions. However, the path-sensitive analysis can be inefficient, especially for large programs like the Linux kernel. To tackle this problem, we propose the *APH specification-based graph (ASG)*, which retains only the necessary information related to APH specifications. Specifically, ASG focuses on *critical* variables changed by the target APIs and thus require the post-operations (e.g., the variable `dwc->dwc3` in Figure 1), thereby significantly reducing the amount of code to be analyzed.

We evaluated APHP on four popular open-source programs, including the Linux kernel, QEMU, Git, and Redis. We confirmed that APHP found 410 new APH bugs, with 2 CVEs assigned. Until the submission of the paper, we have reported the APH bugs and submitted 274 patches. Among them, 216 APH bugs have been confirmed by maintainers, and 201 patches have been accepted. Interestingly, 50% of the detected bugs existed for over five years and seven months before our detection. Compared with the state-of-the-art tools such as MVP [53], APHP outperforms it by improving precision by 0.49 and recall by 0.97.

Contributions. We summarize our contributions below.

• **A novel approach for detecting APH bugs.** We propose a new framework, APHP, aimed at detecting APH bugs by extracting APH specifications from patches (Section 4). APHP automatically obtains APH specifications using both code and descriptions in patches. To improve efficiency, we design four-tuple defined APH specifications (Section 2.2) and APH specification-based graph (ASG) (Section 5), enabling APHP to scale to large programs. We plan to release our code and dataset[1] to facilitate further research.

• **Numerous new bugs on popular programs.** We evaluate APHP on four well-tested popular programs. APHP revealed 410 APH bugs that could cause various security issues. Among them, 216 bugs or patches have been confirmed or accepted by developers, and 2 CVEs assigned. Many patches have been backported to long-term stable versions of the Linux kernel to improve system security (Section 7.1).

• **Comprehensive comparisons with existing approaches.** We compare APHP with three types of bug detectors: patch-based, document-based, and inconsistency-based approaches. Our results demonstrate that APHP outperforms or compensates these approaches in detecting APH bugs (Section 7.2).

• **New findings and insights.** Our experimental analysis provides new insights into the nature of APH bugs, revealing error-prone APIs, implicit conventions and deviations from standard API designs can cause hundreds of bugs when developers are unaware of them. Based on these findings, we provide practical suggestions for avoiding and fixing APH bugs (Section 7.5).

2 Background and Motivation

2.1 API Post-Handling

API Post-Handling (APH) manages the API's effects after its call. It typically involves return value checks and paired function calls. *(1) return value checks*: APIs return different values, indicating various internal states. Developers must handle these values to catch errors and maintain correct program states. *(2) paired function calls*: APIs may request resources like memory and reference counts. Paired functions clean up these resources after usage, e.g., using `closedir` after `opendir`. Proper APH is crucial for security, while improper APH can lead to APH bugs. These bugs can be classified into two categories: missing APH and incorrect APH. Missing APH omits necessary operations, while incorrect APH performs wrong operations after API calls. Previous studies discuss APH bugs' security implications [15, 27, 29, 42]. For instance, missing return value checks may cause null-ptr-deference [29], while incorrect return value checks can lead to unintended behaviors [15]. Missed paired function calls

[1] Available in https://github.com/Yuuoniy/APHP.

can result in memory leaks and deadlock [27], whereas incorrect paired function calls may cause use-after-free issues [3]. Therefore, detecting APH bugs is vital for security, but this task is difficult because it is often unclear which APIs require post-handling or what specific operations they require.

2.2 API Documents and Specifications

The API documents describe APH specifications, guiding developers in using APIs correctly. These specifications encompass critical elements and considerations for using APIs. In a document introducing an API, the API is the *target API*. The document details the target API's usage specification, among other information, like functionality, and parameters. For instance, the Linux kernel documents outline the specification of `kobject_init_and_add` (the target API): *"If this function returns an error, kobject_put() must be called to properly clean up the memory associated with the object"*. We underline key elements of the APH specification. "*kobject_put()*" is a post-operation to clean up the resource allocated by the target API. "*the object*" is a *critical variable*, which requires the post-operation. The clause "*if this function returns an error*" describes the API execution status's condition. Post-operations might have separate documents detailing usage conditions. For example, the `platform_device_put` post-operation document states: "*This function must _only_ be externally called in error cases. All other usage is a bug"*. The phrase "*in error cases*" refers to the path return status's condition. These conditions, affecting post-operation usage, are called *path conditions*. We define the key elements below.

• **Target API**: the function that affects variables and requires a post-operation. Specifically, the target API may return a value or change its argument.

• **Post-operation**: is performed after the target API to handle its effects thus avoid security issues. It includes error checks, memory releases, etc.

• **Critical variable**: the variable affected by the target API and requiring the post-operation. It can either be the return value or the argument of the target API.

• **Path conditions**: represent the conditions of applying a post-operation. As we know, a target API may contain different operations depending on the execution status, which also requires different post-operations. Also, the return status of the path affects the use of post-operation. These statuses can be characterized by path conditions.

Therefore, we present APH specifications as a four-tuple <*target API, post-operation, critical variable, path conditions*>, meaning the post-operation should be performed on the critical variable when the path calls the target API and path conditions are satisfied. Researchers have proposed methods for extracting specifications from API documents and detecting violations in code [30, 33]. However, missing critical information and the difficulty of automatic extraction from unstructured text make it necessary to explore alternative sources containing APH specifications.

2.3 Security Patches

Vulnerabilities can compromise software security, allowing attackers to exploit them, thereby threatening the system's integrity and security. To address this issue, developers release security patches to fix vulnerabilities. These patches are typically available as commits in open-source repositories and consist of patch code and patch descriptions[2] (Figure 2 (a)). Patch code represents the changes made to the code, including added or deleted statements marked with + or −. Patch descriptions provide crucial information related to the vulnerability, such as the reason for the patch, its behavior, and the security impact. As seen in Figure 2 (a), the patch description includes a summary in the first line, followed by a detailed explanation containing more information about the vulnerability, the modified function, the trigger conditions, and the patch behaviors.

Existing methods for using security patches to detect new bugs [16, 20, 53] focus on patch code only and employ techniques like code clone detection. However, functions with APH bugs that violate the same APH specification can vary significantly. As a result, these methods cannot detect APH bugs effectively due to code clone detection technique used and inaccurate patch characterization. Although patch descriptions contain valuable information about bugs, automatically utilizing them is challenging due to the lack of consistent templates or styles within these descriptions. In our research, we explore a way to utilize patch descriptions for bug detection.

3 Overview

Architecture. Figure 3 shows the overview of APHP. APHP aims to automatically obtain APH specifications from patches and use them to detect APH bugs in a target program. It consists of two main phases: *APH Specification Extraction* and *Bug Detection*.

In the APH Specification Extraction phase, APHP extracts the APH specification from a patch. It contains the following key elements: post-operation, target API, critical variable, and path conditions. Specifically, if the patch inserts the post-operation directly, APHP identifies it using code differences. Otherwise, APHP utilizes path differences to identify the post-operation. Then, APHP identifies the target API with code semantics derived from post-operation and textual semantics in the patch description. After that, APHP extracts the critical variable and path conditions via static analysis. The critical variable is operated by the target API and the post-operation. Path conditions are held by paths that pass through the target API and the post-operation.

During the Bug Detection phase, APHP identifies specification violations in the target program using path-sensitive analysis. To expedite detection, APHP first retrieves functions that call the target API. These functions are the targets for checking. Then, APHP constructs each function's

[2]In this paper, we refer to the commit message associated with the patch as the "patch description", based on previous research [46, 57].

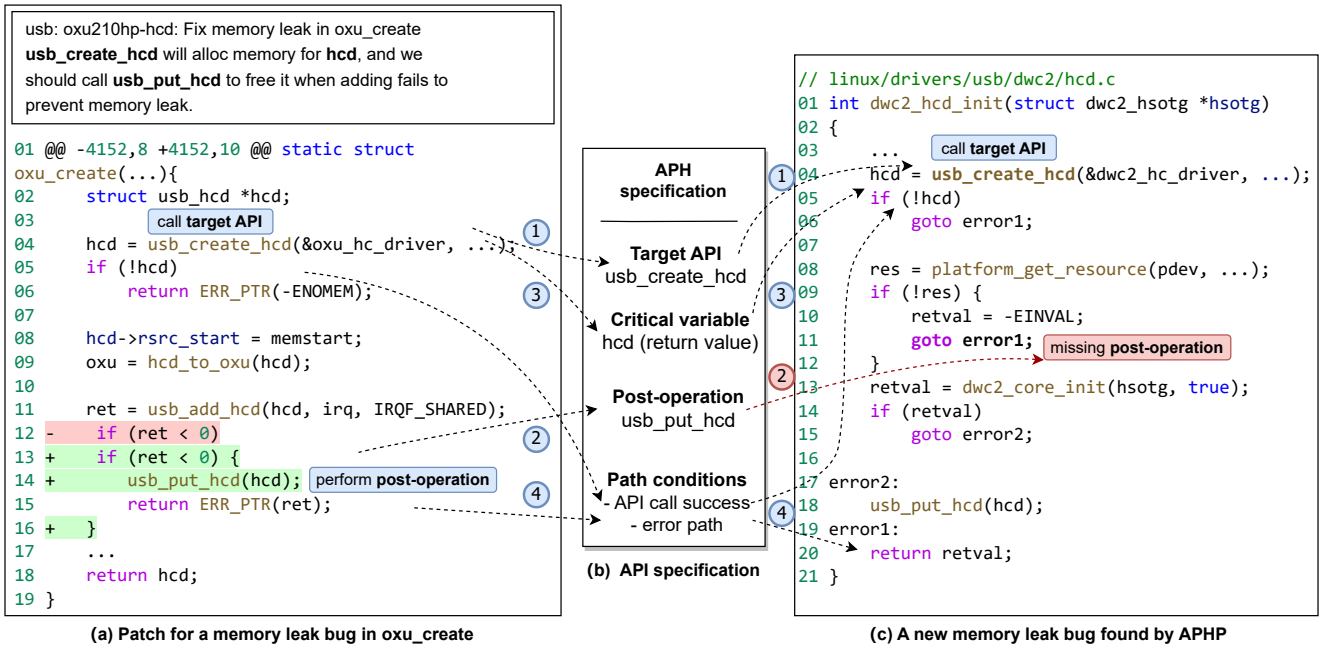

Figure 2: An example to illustrate detecting new APH bugs using patches.

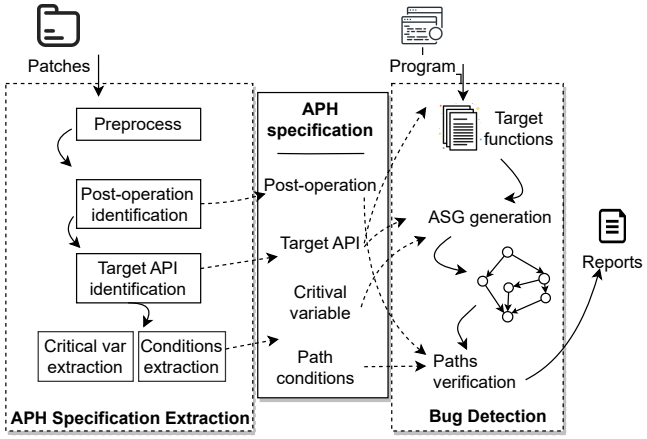

Figure 3: The overview of APHP.

CFG and generates the corresponding ASG with critical variable and target API in the specification. The ASG enables APHP to focus on API usage information, thereby preventing path explosion. Subsequently, APHP examines paths in the ASG to detect violations, identifying paths meeting conditions and verifying the presence of the post-operation. If the post-operation is absent or incorrect, APHP reports it as a potential bug and provides the violated specification for further manual verification.

Example. Figure 2 provides an example to show how APHP detects new bugs using patches. Figure 2(a) shows a patch for a memory leak bug, including the patch description and code differences. Before applying the patch, the API `usb_create_hcd` is misused because the allocated memory resource is not released when `usb_add_hcd()` fails. The patch adds the post-operation `usb_put_hcd` in the error path at line 14 to fix it. A similar bug exists in the `dwc2_hcd_init` function as is shown in Figure 2(c). After calling the API `usb_create_hcd`, the function misses the post-operation when `platform_get_resource()` fails. To detect the latter bug, APHP extracts the APH specification in Figure 2(a) and detects its violations. Specifically, APHP identifies the post-operation `usb_put_hcd` using code differences. The API `usb_create_hcd` is data-dependent with the post-operation and is also mentioned in the patch description. Thus APHP infers it as the target API. Furthermore, because `usb_put_hcd()` and `usb_create_hcd()` both operate on the variable `hcd`, which is the return value of `usb_create_hcd()`, APHP identifies it as the critical variable. Finally, APHP collects conditions in the paths going through `usb_put_hcd()` and `usb_create_hcd()`. The post-operation `usb_put_hcd` is applied in the subsequent error path after a successful call to the API `usb_create_hcd`.

After specification extraction, APHP obtains the APH specification shown in Figure 2(b). Then it investigates the program to detect violations of the specification. Specifically, APHP first obtains callers of `usb_create_hcd` and gets function `dwc2_hcd_init`. Then, it constructs the CFG of the function and generates the corresponding ASG with the specification. Subsequently, it verifies paths in the ASG to detect violations. For each path, APHP finds the path that meets the conditions and further checks whether the post-operation exists. The path 04-05-08-09-10-11-20 in Figure 2(c) is missing the post-operation `usb_put_hcd`, thus APHP reports it as a potential bug. Unlike previous approaches, such as MVP, the feature of APHP is to extract APH specifications using patches, rather than detecting similar vulnerable code.

4 APH Specification Extraction

For a given patch, APHP sequentially extracts the elements of the APH specification: post-operation, target API, critical variable, and path conditions. As patches that fix APH bugs usually have minor code differences and are closely tied to post-operations, APHP first identifies the post-operation, which is typically easier to identify. Next, APHP identifies the target API by performing relaxed data-flow analysis on the post-operation and leveraging the patch description. Finally, based on the post-operation and the target API, APHP extracts the critical variable and the path conditions.

4.1 Preprocessing

APHP begins by preprocessing the patch, converting it into a proper format for later analysis. Each patch commit is identified by a unique hash value. Given this hash value, APHP uses Pydriller [41] to parse the commit, extracting both the patch code and its description. Specifically, APHP first locates the modified file and obtains the modified function. From there, APHP obtains the fixed function from the modified file version after the patch, and the buggy function from the version before the patch. It then computes the differences between these two versions using AST difference, which is syntax-aware and more fine-grained than text-based differences. After parsing the patch, APHP constructs Code Property Graphs [54] (CPGs) for both the fixed and buggy versions of the function. CPGs offer a comprehensive view of the code, enabling analysis techniques such as backward slicing. Specifically, a CPG integrates properties from various code representations, including abstract syntax trees (AST), control flow graphs (CFG), and program dependence graphs (PDG), into a unified structure.

4.2 Post-Operation Identification

Following the preprocessing, APHP proceeds to identify the post-operation within the patch. Post-operations are necessary for fixing APH bugs, and their absence often signals the presence of such bugs. As discussed in Section 2.1, APH can be categorized into two types: return value checks and paired function calls. Accordingly, APHP focuses on extracting post-operations from conditional statements and call statements, aligning with the focus of other work [27]. This strategic focus filters out statements unrelated to post-operations.

Before identifying the post-operation, we divide patches into *direct fix* and *indirect fix*, depending on whether the code differences include the post-operation. In a direct fix, the code differences include post-operation, while in an indirect fix, they do not [55, 56]. For example, developers often use `goto` statements to avoid duplicating code in such fixes.

For a direct fix, APHP directly identifies the post-operation using the code differences. Specifically, APHP first obtains the added statements via AST-difference, and then uses AST parsing to extract the post-operation. For instance, in Figure 2(a), APHP identifies the newly inserted statement at line 14, and consequently obtains the post-operation `usb_hcd_put` from this statement.

As for an indirect fix, the post-operation cannot be directly extracted from the code differences. To address the problem, we consider path-level differences to identify it. Specifically, the *buggy path* denotes the path in the buggy function that lacks the necessary post-operation. To fix the buggy path, the patch modifies the function's control flow to include the missed post-operation. Therefore, the path difference between the buggy and patched paths allows APHP to identify the post-operation. Based on this idea, APHP obtains the patched and buggy paths using the code differences, then it compares the two to identify the post-operation. For a detailed description of this procedure, refer to Appendix A.

Although the patch may insert missed post-operations in multiple buggy paths[3], we do not need to consider every path's differences, as the post-operations are the same. Instead, we only need to consider the differences of a single path pair to identify the post-operation. Moreover, if the patch simultaneously contains a conditional statement and a call statement, APHP uses keywords in the patch description to infer the APH type, then focus on the corresponding statement type. For example, the keyword *"check"* indicates return value checks, while *"leak"* indicates paired function calls.

4.3 Target API Identification

After identifying the post-operation, APHP next determines the target API, which is the function that requires a follow-up post-operation when necessary. Identifying this API associated with the bug is crucial for detecting APH bugs. Intuitively, the post-operation operates the target API's return value. Thus we can find the function call that assigns its return value to the variable that the post-operation operates. However, functions can also pass data via parameter pointers, not just return values. For example, in the patch shown in Figure 4, the `kobject_init_and_add` function in line 6 is the target API, and `kobject_put` in line 10 is the post-operation. The variable involved is the first argument of the target API. In this case, solely performing return value-based analysis on line 10 would miss the function call in line 6, leading to inaccurate results. Besides, it is hard to model all functions invoked by the buggy function to figure out how they operate arguments and return values. Another possible way is to retrieve functions associated with the post-operation. However, considering all related functions introduces noise, resulting in inaccurate results.

We observe that developers often highlight the target API in patch descriptions. Based on this observation, we propose

[3] https://git.kernel.org/torvalds/c/b92675d998a9

```
int bond_sysfs_slave_add(struct slave *slave)
{
    const struct slave_attribute **a;
    int err;

    err = kobject_init_and_add(&slave->kobj, &slave_ktype,
                  &(slave->dev->dev.kobj), "bonding_slave");
-   if (err)
+   if (err) {
+       kobject_put(&slave->kobj);
        return err;
+   }
    ...
}
```

Figure 4: Patch for a reference count leak in Linux kernel.

using the patch description as auxiliary information to identify the target API. The idea is to use code semantics in patch code to identify all possible target API candidates, while utilizing textual semantics to infer the most likely one. The textual semantics within the patch description greatly facilitate identifying the target API, circumventing the need for inter-procedural data-flow analysis or modeling of all invoked functions. Below, we describe the details.

First, APHP utilizes code semantics to identify all potential candidates by performing a relaxed data-flow analysis on the post-operation. It considers all function calls that utilize variables, not just those that explicitly assign variables, thereby addressing the limitations of existing data flow analysis that may overlook possible candidates. Specifically, APHP performs backward slicing on the CPG of the fixed function, using the post-operation statement as the slicing criterion, and focusing on the variable within this statement (step 1). Consequently, APHP collects all statements that have a data relationship with the variable in the post-operation statement. Following this, APHP retrieves all variables from the previously identified statements via AST parsing and further obtains all function calls that assign a value to the variable or utilize the variable as an argument (step 2). These functions are candidates for the target API.

Following that, APHP leverages textual semantics to infer the target API, based on the observation that such information is often mentioned in the patch description. To do this, APHP removes less relevant content from the description and matches tokens to identify the target API. Specifically, we note that developers typically mention the target API in main clauses, while conditional clauses are usually less relevant. For example, in Figure 5, the clause *"when the call to platform_device_add_proprieties fails"* describes the condition triggering the bug, which is not relevant to the target API. Therefore, APHP enhances identification accuracy by removing clauses starting with *"if"* or *"when"*. After removing these clauses, if a function exists in both the target API candidates and the patch description, APHP infers it as the target API. Since function names in the patch description often appear as tokens, APHP tokenizes the description to match with target API candidates.

We illustrate the process with two examples. In the first example, as shown in Figure 2, after determining `usb_put_hcd` as the post-operation, APHP identifies the variable `hcd` from the statement `usb_put_hcd(hcd)` using AST parsing. APHP then conducts relaxed data-flow analysis on the variable `hcd` to obtain the target API candidates, which include `usb_create_hcd`, `hcd_to_oxu`, `usb_add_hcd`, etc. These candidates are then matched with tokens in the patch description. Since the API `usb_create_hcd` is present in both the candidate list and the description's token list, APHP infers it as the target API. For the second example, we turn to the patch description of CVE-2019-18813 in Figure 5, which mentions three functions. Among them, `dwc3_pci_probe` is not related to the post-operation, while `platform_device_add_properties` is mentioned within a conditional clause *"when the call to ... fails"*. As a result, APHP discards them during target API identification, ultimately inferring `platform_device_alloc` as the target API.

usb: dwc3: pci: prevent memory leak in dwc3_pci_probe

In **dwc3_pci_probe** a call to **platform_device_alloc** allocates a device which is correctly put in case of error except one case: when the call to **platform_device_add_properties** fails it directly returns instead of going to error handling. This commit replaces return with the goto.

Figure 5: Patch description of the patch for CVE-2019-18813.

4.4 Critical Variable Extraction

After extracting the target API, APHP extracts the critical variable. This variable, specifically affected by the target API and requires post-operation, forms the focus of APHP's analysis. APHP can thereby track its specific data flow and filter out extraneous information, such as operations related to other variables in the target API. The critical variable appears in both the target API call and the post-operation statement. However, directly recording the variable name is infeasible as it changes in various contexts.

Therefore, we represent the critical variable by its relative position in the target API (e.g., return value, first argument, or second argument). The variable is critical because it plays a specific role in the target API call. With the target API and relative position information, APHP can identify the critical variable consistently regardless of its naming in various contexts. Specifically, APHP employs AST parsing to find the common variable between the target API call and the post-operation statement. It then determines this variable's relative position in the target API call and identifies it as the critical variable. For example, as shown in Figure 2, APHP parses the target API call `hcd = usb_create_hcd(&oxu_hcd_driver, ...);` and the post-operation statement `usb_put_hcd(hcd);`. It identifies `hcd` as the common variable and, upon further analysis, recognizes it as the return value in the target API call. Therefore, APHP takes *return value* as the critical variable.

4.5 Path Conditions Collection

After obtaining the target API and the post-operation, APHP collects path conditions. These conditions capture the path-related semantics, providing a precise characterization of the specifications. We classify path conditions into two types: *pre-condition* and *post-condition*. Though previous research like Advance [30] uses pre- and post-conditions to characterize API usages, we refine them in this paper to represent the assumptions before and after the post-operation.

The pre-condition represents the target API's execution status, which can be either "`success`" or "`failure`". Different execution statuses of the target API produce different effects, which subsequently affect the post-operation. The post-condition represents the return status of the path, which can be either "`normal`" or "`error`". Post-operations differ according to the return status. For instance, as illustrated in Figure 2(a), the post-operation must be executed after the successful API call (pre-condition) and before the error path returns (post-condition). If either the pre-condition or the post-condition is not satisfied, the path does not require the post-operation `usb_hcd_put`.

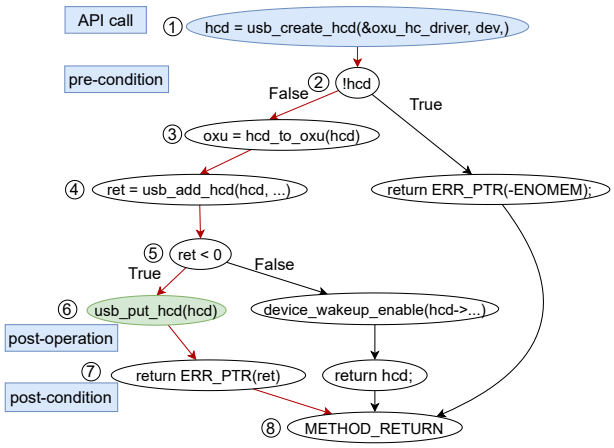

Figure 6: Simplified CFG of oxu_create function.

Specifically, APHP analyzes paths in the fixed function's CPG to collect path conditions. These paths go through the target API and the post-operation. To collect the pre-condition, APHP uses a method inspired by APEx [17] to infer the execution status of the target API. The basic idea is: if an API call fails, the caller will immediately return without executing the rest of the normal path. Therefore, the number of statements along the API call's error path is fewer than those along the corresponding success path. Additionally, the pre-condition might be empty if no condition imposed on the target API's execution status. For post-condition collection, APHP bases its analysis on the path's return value. This is because error return values are distinct from normal return values, mirroring the path's potential return status of error or normal. To distinguish these return values, APHP uses a similar method to that adopted in IPPO [27]. For instance, in the Linux kernel, negative values and error pointers with `ERR_PTR` represent error values.

We illustrate this process using Figure 6, which shows the sub-CFG of the `oxu_create` function. Here, APHP identifies the path ①-②-③-④-⑤-⑥-⑦-⑧ that includes the target API and the post-operation, as it is the only path that traverses both `usb_create_hcd` and `usb_put_hcd`. Further, APHP analyzes the path to collect the path conditions. Specifically, APHP parses the call statement in node ① to obtain the return value `hcd` and identify the condition applied to it. Node ② enforces a check on `hcd`, and this path takes the `False` branch. Unlike the other branch that returns immediately, this branch has more statements, indicating the successful execution status. Thus, the pre-condition is determined to be "`success`". Finally, APHP finds node ⑦ as the return node within this path. By observing the return value `ERR_PTR(ret)`, APHP classifies the path as an error path due to the presence of `ERR_PTR`. Thus, the post-condition is "`error`".

5 Bug Detection

Following the APH specification extraction, APHP proceeds to investigate the violations of these specifications. Each specification is structured in the following format: *<target API, post-operation, critical variable, path conditions>*. The aim is to find the path that calls the target API and meets the path conditions, then verify whether the post-operation is correctly performed on the critical variable. If the post-operation is missing or incorrect, APHP identifies it as a potential bug.

For precise bug detection, path-sensitive analysis is needed. However, this requires exploring all paths, which is cumbersome or even infeasible, especially for large-scale programs. A more efficient approach is to focus on a subset of paths by selecting bug-related code in advance. Intuitively, slicing on the critical variable can filter out irrelevant code. However, accurately obtaining the desired slice is challenging due to the need to model function effects [2]. Besides, it is difficult to strike the right balance between data-flow and control-flow analysis. Data-flow slicing may exclude essential statements, such as the crucial return statement on line 15 in Figure 2(a) when slicing the `hcd` variable, which is vital for path condition checking. While control-flow slicing would introduce unrelated statements. Therefore, we introduce the *APH specification-based graph* (ASG), a carefully designed sub-CFG that retains only code related to APH specifications, including code associated with critical variables and path conditions. By doing so, we reduce analysis complexity while retaining crucial information for bug detection.

Given a specification, APHP identifies functions that are related to the target API and checks selected paths within these functions. More specifically, APHP first identifies the target API's caller functions and constructs their CFGs, and subsequently generates an ASG for each function. Then, APHP

explores the ASGs to identify paths that satisfy the path conditions. In this way, APHP conducts partial path-sensitive analysis on the ASGs, focusing on specific critical paths and capturing path-related behaviors. This approach may introduce some false positives due to the absence of context outside of ASGs. We discuss this in Section 7.1. Finally, APHP determines whether the post-operation exists in the selected paths. Next, we describe the details below.

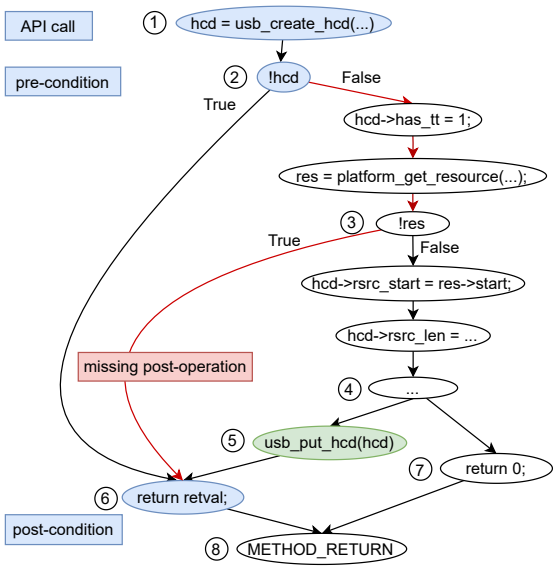

Figure 7: ASG of dwc2_hcd_init function.

ASG generation. APHP generates the ASG by preserving only APH-related semantics. Since effects occur after the target API call, the code range between the API call and function exit is most relevant. Within this range, only statements related to critical variables and path conditions are essential for checking, while others are considered irrelevant. Conditional statements regarding target API's execution status and return statements usually after the API call, making statements preceding it generally unimportant for bug detection.

Based on the idea, APHP generates the ASG with the following steps. It first removes the statements prior to the target API by topologically sorting the CFG nodes. Topological sorting [43] linearizes the nodes of a CFG, ensuring that each node appears before all the nodes it influences. To prevent failures in topological sorting due to CFG cycles, we unroll loops once, a technique used in previous research [27, 29]. As APH bugs are typically unrelated to the number of loop executions, this rarely impacts the correctness. Furthermore, APHP only keeps statements that are related to path conditions or critical variables, termed path condition related statements (PCSs) and critical variable related statements (CVSs). PCS is the same as explained in Section 4.5, and CVS uses the critical variable or assigns a value to it. Note that we take statements that refer to the aliases of the critical variable as CVSs, too. Specifically, APHP extracts the critical variable from the AST of the target API call statement using the APH specification. For example, if the critical variable specified is *return value*, APHP obtains the variable name corresponding to the return value in the target API call statement. Additionally, APHP performs data-flow analysis to identify aliases of the critical variable. Using these aliases, APHP obtains all CVSs.

Paths verification. After ASG generation, APHP significantly reduces the number of paths to be analyzed. APHP further validates paths in the ASG to detect APH bugs. Specifically, APHP collects path conditions for each path and determines whether it meets the required path conditions. The path conditions are collected using the method described in Section 4.5. Subsequently, APHP performs a direct literal comparison to determine if conditions are satisfied. Since different string values represent distinct conditions, the direct comparison is sufficient to determine their equivalence. APHP excludes paths that do not meet the conditions in the given specification. Finally, APHP determines whether the remaining path performs post-operation on the critical variable or its aliases. If the post-operation is missing or incorrect, APHP reports it as a potential bug.

paths	pre-condition	post-condition	post-operation
P1 ①②⑥⑧	✗	✓	—
P2 ①②③⑥⑧	✓	✓	✗ 🐞
P3 ①②③④⑥⑧	✓	✓	✓
P4 ①②③④⑦⑧	✓	✗	—

Figure 8: Path verification for dwc2_hcd_init function.

For example, referring to the specification in Figure 2(b), APHP first obtains callers of the target API `usb_create_hcd` and constructs their CFGs. The function `dwc2_hcd_init` in Figure 2 is one of the callers, with a total of 254 lines of code. APHP generates the ASG based on the API `usb_create_hcd` and its return value `hcd`, as shown in Figure 7. After that, only a few paths remain. Then, APHP investigates all possible paths in the ASG. As shown in Figure 8, there are four paths in the ASG. Since only P2 and P3 satisfy the conditions in the APH specification, P1 and P4 are ignored. When investigating P2 and P3, APHP finds that P2 misses the post-operation. Therefore, APHP reports it as a potential bug.

6 Implementation

We implemented an APHP prototype consisting of 3.5k+ lines of Python code and 200+ lines of Scala code. The implementation includes two modules: APH specification extraction and bug detection. The APH specification extraction module processes and analyzes patches to generate APH specifications, while the bug detection module utilizes these specifications to identify bugs and generate bug reports. We employ Joern [54], a widely-used tool in previous research [53, 59], for comprehensive code analysis. To handle loops, we unroll them by treating *for* and *while* statements as *if* statements, a common strategy used in previous work [29]. We provide more implementation details in Appendix B.

Table 1: Target program overview.

Program	Version	Lines of code	Num of patches	Category
Linux kernel	5.16-rc1	22.5M	14115	Operating System
QEMU	7.0.0-rc4	1.8M	1096	Emulator
Git	2.37.3	870K	363	Version Control
Redis	7.0.0	263K	142	Database

7 Evaluation

To extensively evaluate the effectiveness of APHP, we first evaluated its effectiveness in finding unknown bugs and runtime performance. We further compared APHP with three different types of bug detectors: patch-based [20, 53], document-based [30], and inconsistency-based [27]. Then, we evaluated the effectiveness of APHP in extracting specifications and examined the contribution of each component of APHP.

Dataset. We evaluated the effectiveness and performance of APHP on Linux kernel, QEMU, Redis, and Git, including a total of 25.43 million lines of code. The target programs cover different kinds of software. We collected APH-related patches by referring to the methods used in previous work [49]. In particular, patch descriptions of APH-related patches contain special keywords, and 96.15% of API misuse can be patched with fewer than ten lines of code [13]. Therefore, we filtered patches by matching APH-related keywords (e.g., "leak", "error handling", "return value"). Patches with no code changes or code changes exceeding ten lines were excluded. While APHP is capable of handling patches with more than ten lines of code changes, we focus on the most common cases based on previous research. Table 1 shows the overview of target programs and the number of patches collected.

As it is impractical to identify all APH bugs and confirm all specifications manually, we constructed a ground-truth dataset to evaluate the false negatives and the effectiveness of specification extraction in APHP. Specifically, we randomly selected 100 APH patches (D_{patch}) from Table 1 and manually summarized the correct specifications exhibited by the patches to create a ground truth. Based on these specifications, we constructed a dataset of 100 bugs (D_{bug}) to evaluate false negatives. Specifically, we randomly selected a target function to inject bugs for each specification by deleting the correct post-operation. In other words, the target functions violate the specifications.

Platform. All experiments were carried out on a single 64-bit server running Ubuntu 20.04 LTS and equipped with eight Intel(R) Xeon(R) Silver 4110 CPU cores running at 2.10GHz, 128GB of memory, and a 22TB hard drive.

7.1 Effectiveness on Bug Findings

Bug findings. APHP generated bug reports with the corresponding buggy functions and the violated APH specifications. We manually validated each bug report. Specifically, we took bug reports as true APH bugs when the API usage violated its specification. This indicates that the required post-operation of the target API call is either missing or incorrect, regardless of whether the bug is fireable or not. In addition, most APH bugs are confined to a single function, and the reports provide helpful information. We can typically analyze a bug report within a minute. We confirmed 410 true bugs from 667 bug reports generated by APHP. This took us about 12 person-hours, which we believe is manageable as a one-time effort. Until the submission of the paper, we have submitted patches to fix 274 bugs, and 216 new bugs have been confirmed where 201 patches have been accepted. Specifically, we found 402, 5, 2, and 1 bugs from the Linux kernel, QEMU, Git and Redis, respectively. APHP found proportionally more bugs in Linux kernel due to its extensive code reuse and widespread misuse of error-prone APIs. We discuss findings regarding the prevalence and causes of APH bugs in Section 7.5. The numerous APH bugs found show that developers easily make mistakes in API post-handling. The detailed list of partial bugs is shown in Table 8 and Table 9 in the Appendix. Many of our submitted patches have been backported to long-term stable versions of the Linux kernel, helping to improve the system security. In addition, we analyzed the latent period of the found bugs. The average time from bugs introduced by commits to detection is 1967 days, or about five years and four months. Moreover, 5% of bugs have a latent period of more than 12 years and ten months, and 50% of bugs exist longer than five years and seven months.

Security impact. Table 2 summarizes the major impacts of the confirmed bugs. Specifically, 14.9% of the bugs lead to memory leaks. 54.1% of the bugs lead to reference count leaks (refcount leaks). 20.2% of bugs lead to restricted use of resources due to missing resource releases. In addition, failures to check properly after API calls also lead to various problems, including NULL pointer dereference (null-ptr-deref, 1.5%) when triggered or unintended behaviors that affect the

Table 2: Major impacts of bugs found by APHP.

Types of bugs	Prop	Causes	CWE-ID
Refcount leak	54.1%	MFC	CWE-911
Memory leak	14.9%	MFC/WFC	CWE-401
NULL pointer dereference	1.5%	MRC	CWE-690
Restricted use of resources	20.2%	MFC	CWE-404
Reliability	8.8%	WRC/MRC	CWE-235

Note: CWE = common weakness enumeration. MFC = missing paired function call, IFC = incorrect paired function call, MRC = missing return check, IRC = incorrect return check.

Table 3: Execution time of APHP.

Program	Phase 1	Phase 2	Total
Linux	03h08m	04h50m	07h58m
QEMU	37m04s	38m30s	75m34s
Git	19m38s	26m55s	46m33s
Redis	10m43s	08m33s	19m16s

Note: Phase 1: APH specification extraction, Phase 2: bug detection.

Table 4: Accuracy of VUDDY, MVP and APHP

Program	Bugs	VUDDY					MVP					**APHP**				
		#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall
Linux kernel	405	3	103	402	0.03	0.01	8	64	397	0.11	0.02	402	246	3	0.62	0.99
QEMU	5	0	4	5	0.00	0.00	0	0	5	N/A	0.00	5	3	0	0.63	1.00
Git	3	0	9	3	0.00	0.00	1	0	2	1.00	0.33	2	6	1	0.25	0.67
Redis	1	0	1	1	0.00	0.00	0	0	1	N/A	0.00	1	2	0	0.33	1.00
Total	414	3	117	411	0.03	0.01	9	64	405	0.12	0.02	**410**	**257**	**4**	**0.61**	**0.99**

system's reliability (8.8%). The remaining bugs (0.5%) cause problems such as memory corruption. Two CVEs related to APH were assigned in the process: CVE-2022-29156 in Linux kernel and CVE-2022-33105 in Redis. Both of them are evaluated as of high severity.

False positives. We manually confirmed 667 bug reports, of which 402 are true bugs and the rest are false positives. The false positive rate of APHP is 39%, which is acceptable for static analysis-based detection and lower than related tools, such as IPPO [27], which has a 63.5% false-positive rate. We identify three main causes of false positives. (1) Imprecise data flow analysis causes 55% of false positives. APHP's intra-procedural analysis struggles to track data flow when post-operations and target APIs are in different functions. (2) Incorrect path condition analysis accounts for 33% of false positives. APHP employs methods from prior studies [17, 27], such as Apex [17], to extract path conditions. However, these methods can be imprecise due to inaccurate assumptions. For instance, Apex assumes that successful API calls contain more statements than their corresponding fail paths, but this assumption is not always true, leading to false positives. (3) The remaining 12% of false positives stem from insufficient contextual information and unusual code patterns. In some cases, target APIs are executed under specific conditions (e.g., when a local variable `flag` is set). However, ASGs may filter out this context, causing false positives.

False negatives. We evaluated APHP's false negative rate by running it on D_{bug} using D_{patch}. APHP detected a total of 84 bugs, with a false negative rate of 16% ((100-84))/100). We identify four main reasons for these false negatives. First, six bugs are missed due to incorrect specification extraction. We evaluate specification extraction and analyze the failures in Section 7.3. Second, four bugs are missed because the path conditions of the buggy paths fail to meet the specifications' path conditions, causing them to be filtered out. Third, another four bugs are missed due to imprecise path conditions collection, such as failing to recognize error paths for void functions. Finally, one bug is missed due to inaccurate CFG generation by Joern [54]. We note that the specification extraction process may produce partially correct specifications. In some cases, these specifications can still facilitate bug detection. For example, suppose the inferred API is not the correct API that should be identified. If the correct and inferred APIs are frequently used together, bugs may still be identified using the inferred API. Such situation happens for several times in our evaluation.

Run-time performance. We evaluated APHP's run-time performance, as shown in Table 3. For the most time-consuming analysis process against the Linux kernel (23.5 million lines of code), APHP completed the analysis in 8 hours. For other programs, APHP finished in about an hour. These results illustrate that APHP performs bug detection in an acceptable time for large programs like Linux kernel. APHP tackles scalability issues by employing intra-procedural and paritial path-sensitive analysis on ASGs. Specifically, APHP focuses only on the target API's callers to reduce analyzed functions. ASGs help minimize the amount of code analyzed. Nevertheless, APHP's strategies, such as ASGs and intra-procedural analysis, may lead to false positives and negatives due to missed critical intra- and inter-procedural context.

7.2 Comparison with the State-of-the-Art

Comparison with patch-based detectors. To the best of our knowledge, limited work specifically targets detecting unknown APH bugs or API misuses using patches. Thus, we compared APHP with state-of-the-art patch-based bug detectors, VUDDY [20] and MVP [53], which are not limited to particular bug types. While both VUDDY and MVP detect same or similar bugs using bug patches as inputs, APHP extracts API specifications from patches and identifies buggy code violating these specifications. We compared these tools to explore their respective performances in detecting APH bugs. We evaluated these tools using the patches listed in Table 1 as inputs to identify new bugs in the same program. This comparison method is similar to the evaluation method used in MVP [53]. To evaluate the accuracy of various approaches, we used five standard metrics, true positives (TPs), false positives (FPs), false negatives (FNs), precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$). However, obtaining all the bugs in a program is impossible. As a result, we only measured verifiable bugs. To ensure fairness, we considered all bugs found by APHP, VUDDY, and MVP as ground-truth when calculating false negatives (FNs). For instance, FNs of APHP refer to bugs (i.e., not FPs) detected by VUDDY and MVP but not identified by APHP. In particular, we manually examined reports detected by each tool to determine whether they were true bugs. The results of the comparison are shown in Table 4.

False positives analysis for VUDDY and MVP. VUDDY generated 120 reports, including 3 TPs and 117 FPs, primarily due to abstraction technique limitations. Over-abstraction causes VUDDY to fail in distinguishing buggy and patched functions. MVP generated 73 reports, including 9 TPs and 64 FPs. The false positives result from inaccurate vulnerability signatures (41 FPs) and patch signatures (23 FPs). MVP's signatures are too general and can produce incorrect patch signatures, leading to false positives.

False negatives analysis for VUDDY and MVP. Existing patch-based approaches, VUDDY and MVP, have limitations in detecting APH bugs. VUDDY misses 99% of these bugs due to including bug-irrelevant statements in signatures, while MVP fails to detect 98% of APH bugs because of noise introduced by its slicing algorithm and the lack of path-level differences consideration. Additionally, MVP's failure to perform path-sensitive detection results in missed bugs when target functions already have a post-operation.

False negatives analysis for APHP. VUDDY and MVP found 12 true bugs, 8 of which can also be detected by APHP. Two missed bugs resulted from failures in specification extraction. Another two were missed because the target APIs in the buggy code differed from those in the original patches. We discuss the limitations of APHP in handling target API aliases in Section 8.

Comparison with Advance. We compared APHP with Advance [30] to demonstrate the ability of document-based approaches in detecting APH bugs. Other studies are excluded from comparison because they are either not open-sourced (e.g., [61] and [36]) or have inferior performance compared to Advance (e.g., [33]). Advance requires API documents as inputs. To accommodate this, we provide it with documents collected from comments above functions in the analyzed programs, which often contain useful documentation-like descriptions, and from third-party library websites [44] used in these programs. Among the 410 bugs discovered by APHP, only six bugs were detected by Advance. Three factors contribute to Advance's false negatives: (1) 50.4% of cases due to missing post-operation information in API documents; (2) 46.7% of cases result from Advance overlooking sentences lacking strong sentiments, leading to missed specification extraction; (3) 2.9% of cases occur due to Advance's verification code generation missing critical information from documents, resulting in false negatives. The results show that APHP effectively compensates for Advance's shortcomings, detecting bugs that Advance cannot find.

Comparison with IPPO. Existing approaches like FICS [1], Crix [29], and IPPO [27] detect bugs using inconsistency checking, with IPPO being a state-of-the-art tool. We are interested in how many bugs discovered by APHP can be discovered by IPPO, as well as acknowledging that IPPO may identify bugs outside APHP's scope, as it is not limited to detecting APH bugs. Since IPPO has achieved good results in the Linux kernel, 96.7% (266/275) of the bugs found in the Linux kernel, we use the 402 bugs discovered by APHP in the Linux kernel as a benchmark. Results show IPPO fails to detect any bugs in the benchmark due to two factors: *(1) inadequate security operations identification (392 FNs)*, where IPPO struggles to accurately identify custom security operations in programs, and *(2) similar paths requirement (10 FNs)*, where IPPO fails to detect bugs because of missing reference paths and similar path pairing issues. The evaluation shows that APHP can effectively compensate for inconsistency checking.

7.3 Effectiveness of Specification Extraction.

In the Appendix, Table 10 presents the top 50 API pairs sorted by the number of associated patches. Each row details the program, target API, post-operation API, whether the API pair is documented, and the number of corresponding new bugs identified by APHP. The identified API pairs cover various functionality categories, such as memory management, device management, file operations, etc. Furthermore, they exhibit a diverse distribution of bug frequency, with some pairs correlating to more new bugs. We inspected the corresponding API documentation to verify the presence of these post-operations. The results show that 56% (28/50) of them are undocumented but can be extracted from patches. This highlights that patches can provide valuable information not always found in documentation. API documentation and patches serve as complementary sources for specification extraction.

The effectiveness of specification extraction was evaluated on D_{patch}. Results show an overall correctness of 89% for inferred specifications, with a precision and recall of 89%. Each patch has its specification, incorrect extraction of a specification could cause a false positive while simultaneously missing the correct specification, resulting in a false negative. We identify 11 false positives, which stem from incorrect identification of target API, post-operation, and path conditions, resulting in 7, 1, and 3 false positives, respectively. The most challenging aspect is identifying the target API, leading to the majority of failures. Three factors contribute to this issue. First, one failure is caused by inaccurate variable alias analysis. APHP identifies target API candidates using data flow analysis, which involves imprecise alias analysis of variables. This imprecision leads to the omission of some candidates, including the correct target API. Second, imprecise description analysis leads to four failures. As APHP is unable to fully understand text semantics, it misidentifies target APIs when multiple APIs are mentioned in the diverse and complex patch descriptions, failing to identify which API is the target API. Third, missing target API information in patch descriptions results in two failures, as APHP mistakenly infers the target API when it is not mentioned in descriptions. Moreover, APHP fails to identify post-operation in cases of unusual fixing patterns, such as modifying branching conditions to fix reference count leaks [55]. Finally, imprecise

Table 5: Comparison with APHP– on D_{patch} and D_{bug}.

	Specification extraction		Bug detection	
Approach	**Precision**	**Recall**	**Precision**	**Recall**
APHP	89%	89%	45%	84%
APHP–	26.5%	94%	6%	88%

Table 6: Comparison of scale of ASG and CFG.

	Num of nodes	Num of paths	Avg. path length
ASG	14.4	45.4	8.7
CFG	106.0	2942.2	61.6
% Reduction	86.4%	98.5%	85.9%

value analysis causes failures in identifying post-conditions due to inaccurate return value analysis.

7.4 Ablation Study

Contribution of patch descriptions. To evaluate the contribution of patch descriptions, we compared APHP's performance with and without them on D_{patch} and D_{bug}. Without patch descriptions, we treated all target API candidates as the target API and extracted specifications for each. This approach is referred to as APHP–. Table 5 displays the results for precision ((i.e., TP / (FP + TP)) and recall (i.e., TP / (TP + FN) in each phase. During the specification extraction, APHP, which considers patch descriptions, achieved a precision of 89% and a recall of 89%, whereas APHP– had a lower precision of 26.5% and a slightly higher recall of 94%. This suggests that APHP obtains a more accurate target API and, in turn, higher-quality specifications. In bug detection, APHP achieved a precision of 45% and a recall of 84%, while APHP– had a significantly reduced precision of 6% and marginally higher recall of 88%. It is worth noting that APHP– attains higher recall in specification extraction by considering all target API candidates, resulting in a more exhaustive search. However, this leads to a significant decrease in precision and a high number of false positives. In contrast, APHP utilizes patch descriptions for more accurate target API identification, improving overall performance. These results highlight the value of patch descriptions in attaining high-quality specifications and effective bug detection.

Contribution of ASG generation. We evaluated the impact of ASG generation on the amount of code analyzed by comparing the metrics of generated ASGs with those of the original control flow graphs (CFGs). Specifically, we randomly selected 500 target functions and used their original CFGs as baselines. The statistics for ASGs and CFGs are displayed in Table 6. The average reduction in the number of nodes, number of paths, and the average length of paths achieved by ASGs are 86.4%, 98.5%, and 85.9%, respectively, compared to the original CFGs. These results suggest that ASG generation can substantially reduce the amount of code analyzed, consequently enhancing bug detection efficiency.

7.5 Findings and Suggestions

Our investigation yields several key findings that shed light on the prevalence and causes of APH bugs. Specifically, we discover that widespread use of error-prone APIs and developers' limited knowledge of them could result in numerous bugs. Additionally, API implicit usage specifications derived from program-specific conventions and non-standard API designs also contribute to the occurrence of bugs. In the following sections, we first elaborate on our findings and then provide suggestions to prevent and fix APH bugs based on the findings and patch investigations.

Widespread use of error-prone APIs leads to numerous bugs. We find that error-prone APIs frequently misused by developers, leading to bugs. For instance, Open Firmware (OF) Device Tree APIs [18] in Linux kernel, which provide access to hardware devices, are often misused. Certain APIs can retrieve the desired device node, returning a node pointer with a reference count increment. Part of them are shown in Table 7 in the Appendix. Developers should perform `of_node_put()` on the pointer after use to prevent reference count leaks. However, they frequently overlook this specification, resulting in numerous bugs. Table 10 shows additional error-prone APIs like `clk_prepare_enable` and `pm_runtime_enable`.

API misuse due to implicit usage specifications derived from program-specific conventions. APIs may have implicit usage specifications tied to program-specific conventions, causing mistakes by inexperienced developers. For example, functions in the Linux kernel may use `ERR_PTR()` to encode error numbers into pointers. To catch errors in this case, developers should use `IS_ERR()` instead of null checking. However, developers often confuse the two, leading to repetitive bugs.

API designs deviating from default conventions may lead to bugs. Some API designs deviate from default conventions, causing confusion and bugs. For example, the `pm_runtime_get_sync` API has both 0 and 1 as successful return values, contrary to the Linux kernel convention of returning 0 for success and non-zero for failure. This deviation can lead to incorrect error handling. Additionally, APIs `kobject_init_and_add` and `pm_runtime_get_sync` require post-handling for cleanup even when API calls fail, which can be counter-intuitive and cause bugs.

Suggestions for avoiding APH Bugs. Based on our findings, we suggest several ways to avoid APH bugs. First, API designers could follow general conventions and provide clear documentation, such as Javadoc-style annotations [32]. Additionally, documentation for error-prone APIs could include proper usage examples and potential pitfalls. Second, API developers may provide helper functions to simplify API usage, like the `devm_pm_runtime_enable` for the `pm_runtime_enable` API, which performs post-handling automatically. Third, API users should carefully review documentation, paying attention to API usage specifications in

addition to API functionality. Utilizing CI pipeline regression tools [45] can also help identify potential issues.

Suggestions for fixing APH bugs. We propose several suggestions for bug fixers when patching APH bugs: *(1) Investigate patches for effective fixing patterns.* Bug fixers may explore patches using tools like Pydriller [41] and GitPython [12] to discover diverse ways of fixing bugs. For instance, when addressing a memory leak caused by `g_strdup_printf()`, fixers may insert `g_free()` or use the macro `g_autofree` on the pointer variable for automatic cleanup. *(2) Search for similar bugs with tools.* To avoid incomplete fixes, bug fixers may consider examining and addressing instances of similar bugs in other code paths or functions as thoroughly as possible. Tools such as vgrep [38] and weggli [50] assist in quick pattern matching. *(3) Write clear and detailed patch descriptions.* Following community guidelines [19], bug fixers may strive to write clear and detailed patch descriptions, like the one in Figure 2(a), to help other developers understand bugs and learn from past mistakes.

8 Discussion

Identifying APH patches. It is worth mentioning that our method's primary contribution is not identifying APH patches. We employ simple heuristics to find patches potentially related to APH bugs, which may have certain limitations. Upon random inspection of the identified patches, we found that 62% are relevant to APH bugs, while the remaining patches address issues such as code refactoring or fixing other types of bugs. To enhance the accuracy of patch identification, more advanced methods could be considered, such as incorporating text semantics [60] and code semantics [48].

Generalizability. We conducted an empirical study of APH patches to assess the generalization of APHP. In this study, we measured the prevalence of structured patch descriptions among APH patches. The percentages of patch descriptions, including target API, post-operation, critical variable, and path condition, are 88%, 62%, 24%, and 86%, respectively. APHP primarily uses patch descriptions for target API identification and can handle unstructured or incomplete descriptions. In Linux kernel, QEMU, Git, and Redis, 12%, 56%, 61%, and 86% of APH patch descriptions lack mention of the target APIs, with the higher percentage in Redis due to its less standardized descriptions. Additionally, our analysis reveals that 96% of APH patches are confined to a single function, making intra-functional analysis sufficient for most cases. The remaining 4% of patches that modify multiple functions are more complex, such as patches that address multiple bugs simultaneously and patches involving code refactoring. To better handle these cases, APHP could be extended with techniques like inter-procedural analysis [37] and modular analysis [28].

Identifying target API across different code patterns. APHP detects bugs for callers of target APIs using specifications extracted from patches. However, programs may use different code patterns, such as API wrappers and aliases, which can lead to missed bugs. To address this, we can extend APHP using techniques such as code clone detection [40] and call graph analysis [47] to identify and link different code patterns, including wrappers and aliases, to the original target APIs. This allows us to expand the specifications and detect more bugs.

Mitigating false negatives and false positives. To enhance APHP's effectiveness, we can adopt several approaches to reduce both false negatives and false positives. First, we can adopt advanced NLP techniques [8] to better utilize patch descriptions. If patch descriptions do not mention target APIs, we can supplement them with information from other sources, such as the corresponding bug reports. Additionally, as an auxiliary measure, human-built specifications adhering to APHP's format can be employed for bug detection, albeit at the cost of increased manual effort. Next, we can improve path conditions collection by incorporating strategies from other work. Specifically, we can use error-only functions [7] in addition to error codes to better identify error paths. Furthermore, inaccurate data flow analysis is a significant reason for false positives. We can use advanced analysis techniques like alias analysis [34] and escape analysis [5] to address this. Finally, we can employ symbolic execution [35] to eliminate false bug reports that arising from infeasible paths.

9 Related Work

Bug detection using historical bugs. Our work is closely related to previous studies that detect bugs using historical bugs, which fall into two main categories: clone-based and learning-based approaches. Clone-based approaches [16, 20, 25, 51, 53] employ code clone detection. CP-Miner [25] utilizes token-based matching, while VUDDY [20] leverages vulnerability-preserving abstraction. MVP [53] generates signatures for known vulnerabilities. Learning-based approaches [21–24, 59] use vulnerability features to train detection models, often requiring large and clean datasets. Different from them, APHP combines the code and descriptions of patches to identify critical information.

Bug detection using documents. Previous studies focus on detecting bugs using documents, primarily for API misuse detection [30, 33, 36, 61]. For instance, Advance [30] extracts specifications from API documents using sentiment analysis, while Ren et al. [36] build API constraint knowledge graphs. Unlike these studies, APHP mines knowledge from previous bugs to obtain correct APH specifications.

Bug detection using source code. Many studies use source code to detect bugs through inconsistency checking, such as IPPO [27], APIsan [58], and FICS [1]. These approaches identify common API usage patterns or mine function pairs from code [3, 9, 26, 39, 52]. They then check whether the API usage meets the requirements of the identified patterns or function pairs. Unlike these studies that use source code, APHP mines patterns from bug patches.

10 Conclusion

APH bugs are a common type of API misuse bugs, resulting in various security issues. In this paper, we propose APHP, a novel framework to detect APH bugs using patches. APHP extracts APH specifications using code differences and patch descriptions, then uses these specifications to detect bugs. Furthermore, we design the ASG (APH specification-based graph) and perform partial path-sensitive analysis to improve detection accuracy and efficiency. Evaluations on four open source programs show APHP outperforms state-of-the-art approaches, detecting 410 new APH bugs, with 216 confirmed and 2 CVEs assigned.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their insightful comments. We are also grateful to the maintainers of the open source software for their invaluable feedback during our patch submission. The authors are supported in part by NSFC (92270204, U1836211, 62202462), and Youth Innovation Promotion Association CAS.

References

[1] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In Proceedings of the 30st USENIX Security Symposium (Security), 2021.

[2] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A Comparative Study of Two Whole Program Slicers for C. 2001.

[3] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed. In Proceedings of the 28th SIGSOFT Foundations of Software Engineering (FSE), 2020.

[4] Steven Bird, Ewan Klein, and Edward Loper. Natural Language Processing with Python. O'Reilly Media, Inc., 1st edition, 2009.

[5] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.

[6] The MITRE Corporation. CVE-2019-18813: a memory leak vulnerability in Linux kernel. https://www.cvedetails.com/cve/CVE-2019-18813, 2022.

[7] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. Effective Error-Specification Inference via Domain-Knowledge Expansion. In Proceedings of the 27th SIGSOFT Foundations of Software Engineering (FSE), 2019.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. ArXiv, abs/1810.04805, 2019.

[9] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. In Proceedings of the 28th Network and Distributed System Security Symposium (NDSS), 2021.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and Accurate Source Code Differencing. In Proceedings of the 29th International Conference on Automated Software Engineering (ASE), 2014.

[11] Python Software Foundation. difflib. https://docs.python.org/3/library/difflib.html, 2022.

[12] GitPython-developers. GitPython. https://github.com/gitpython-developers/GitPython, 2022.

[13] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In Proceedings of the 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019.

[14] Aric A. Hagberg, Daniel A. Schult, and Pieter Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. 2008.

[15] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In Proceedings of the 25th USENIX Security Symposium (Security), 2016.

[16] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), 2012.

[17] Yuan Jochen Kang, Baishakhi Ray, and Suman Sekhar Jana. APEx: Automated Inference of Error Specifications for C APIs. In Proceedings of the 31st International Conference on Automated Software Engineering (ASE), 2016.

[18] The kernel development community. DeviceTree Kernel API. https://docs.kernel.org/devicetree/kernel-api.html, 2022.

[19] The kernel development community. Doc for submitting patches. https://docs.kernel.org/process/submitting-patches.html, 2023.

[20] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P), 2017.

[21] Z. Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC), 2016.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. IEEE Transactions on Dependable and Secure Computing (TDSC), 2021.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. IEEE Transactions on Dependable and Secure Computing (TDSC), 2021.

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), 2018.

[25] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2004.

[26] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In Proceedings of the 13th SIGSOFT Foundations of Software Engineering (FSE), 2005.

[27] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. In Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS), 2021.

[28] Kangjie Lu. Practical Program Modularization with Type-Based Dependence Analysis. In Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P), 2023.

[29] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In Proceedings of the 28th USENIX Security Symposium (Security), 2019.

[30] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, Xiaofeng Wang, Peiwei Hu, and Luyi Xing. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. In Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS), 2020.

[31] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics ACL), 2014.

[32] Oracle. Javadoc Tool. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html, 2023.

[33] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring Method Specifications from Natural Language API Descriptions. In Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012.

[34] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient Field-Sensitive Pointer Analysis of C. ACM Transactions on Programming Languages and Systems (TOPLAS), 30:4 – es, 2007.

[35] David A. Ramos and Dawson R. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In Proceedings of the 2015 USENIX Annual Technical Conference (ATC), 2015.

[36] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In Proceedings of the 35th International Conference on Automated Software Engineering (ICSE), 2020.

[37] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In Proceedings of the 22nd ACM-SIGACT Symposium on Principles of Programming Languages (POPL), 1995.

[38] Valentin Rothberg. vgrep. https://github.com/vrothberg/vgrep, 2022.

[39] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In Proceedings of the 43rd Annual International Conference on Dependable Systems and Networks (DSN), 2013.

[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal Kumar Roy, and Cristina V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016.

[41] Davide Spadini. Pydriller. https://github.com/ishepard/pydriller, 2022.

[42] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In Proceedings of the 30th USENIX Security Symposium (Security), 2021.

[43] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM J. Comput., 1:146–160, 1972.

[44] GTK Development Team. GLib Documentation. https://docs.gtk.org/glib/, 2023.

[45] The Travis CI team. Travis CI. https://www.travis-ci.com/, 2023.

[46] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawend'e F. Bissyand'e. Is this Change the Answer to that Problem?: Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In Proceedings of the 44th International Conference on Automated Software Engineering (ICSE), 2022.

[47] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In Proceedings of the 15th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2000.

[48] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P), 2022.

[49] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A Large-Scale Security Patch Dataset. In Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021.

[50] weggli rs. weggli. https://github.com/weggli-rs/weggli, 2022.

[51] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In Proceedings of the 31st USENIX Security Symposium (Security), 2022.

[52] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In Proceedings of the 30th USENIX Security Symposium (Security), 2021.

[53] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In Proceedings of the 29th USENIX Security Symposium (Security), 2020.

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P), 2014.

[55] Xiyu Yang. Linux kernel commit 62b4011fa7be. https://git.kernel.org/torvalds/c/62b4011fa7be, 2023.

[56] Yang Yingliang. Linux kernel commit 739752d655b3. https://git.kernel.org/torvalds/c/739752d655b3, 2023.

[57] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS), 2017.

[58] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and M. Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. In Proceedings of the 25th USENIX Security Symposium (Security), 2016.

[59] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS), 2019.

[60] Yaqin Zhou and Asankhaya Sharma. Automated Identification of Security Issues from Commit Messages and Bug Reports. In Proceedings of the 25th SIGSOFT Foundations of Software Engineering (FSE), 2017.

[61] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs Documentation and Code to Detect Directive Defects. In Proceedings of the 39th International Conference on Software Engineering (ICSE), 2017.

A Appendix: Details of Path Difference-Based Identification

This appendix provides a detailed description of the path difference-based identification used in APHP for indirect fixes. APHP identifies the buggy and patched paths by analyzing added and deleted statements in the code differences, and then compares these paths to identify the post-operation. Specifically, APHP obtains the patched paths (PPs) by finding paths that go through added statements based on the CPG of the fixed function. Correspondingly, APHP obtains the buggy paths (BPs) by finding paths that go through deleted statements based on the CPG of the buggy function. If there is only one patched path and one buggy path, APHP directly compares the difference between these two paths to get statements that only exist in the patched path. Otherwise, APHP finds the corresponding buggy path based on path similarity for a given patched path. Specifically, a path is a sequence of statements. Of all the paths in BPs, the corresponding buggy path has the most statements in common with the patched path. As a result, the corresponding buggy path is the most similar to the patched path. The similarity score of the two paths is calculated by the `SequenceMatcher` method in difflib [11].

Consider the indirect fix shown in Figure 1 as an example. The patch removes `return ret` in line 9 and adds `goto err` in line 10. APHP obtains the patched path: 03-04-07-08-10-13-14 using the added statement and gets the buggy path: 03-04-07-08-09 using the deleted statement. Next, APHP compares the two paths to obtain the statement that only appears in the patch path (`goto` statements are ignored). Then APHP obtains the statement `platform_device_put(dwc-dwc3)` in line 13. Further, it gets the post-operation `platform_device_put`.

B Appendix: Implementation Details

This appendix provides implementation details of the APH specification extraction and bug detection modules.

APH Specification Extraction. We use the third-party library Pydriller [41] to process patch commits. Gumtree [10] is a state-of-the-art AST difference tool, and we use it for AST difference. For the text analysis, we used dependency analysis provided by Stanford CoreNLP [31] to remove conditional clauses and NLTK [4] to tokenize patch description. During patch analysis, we filter out obvious noise in the patch code. Specifically, code differences also contain modifications to debug information that are unrelated to the specification, such as function calls `dev_err`, `pr_debug`, etc.

Bug Detection. In the detection phase, we use weggli [50] to obtain callers of specific target functions. Weggli is a very fast and robust semantic search tool that can quickly get the callers of target APIs. We use NetworkX [14] to analyze the CFGs exported from Joern and perform ASG generation and path verification. We perform topological sorting using the NetworkX `topological_sort` interface, which employs a non-recursive depth-first search algorithm. APHP performs intra-procedural analysis and may result in false positives due to missing inter-procedural context. To mitigate this, we employ a one-layer inter-procedural analysis, as used in prior work [42]. As for bug reports, we report bugs and give the APH specifications they violate. It can significantly reduce the time required for manual confirmation. In addition, we provide reference patches. Even if developers are not familiar with the target API, they can quickly capture critical information from the patches. Moreover, these patches also contain descriptions, such as security implications. It assists developers in identifying security threats and prioritizing patching.

Table 7: Device Tree APIs for get device node.

API Description	API
OF device node getter	of_parse_phandle
	of_find_matching_node
	of_find_compatible_node
	of_find_node_by_name
	of_find_node_by_path
	of_find_node_by_phandle
	of_get_child_by_name
	of_find_matching_node_and_match
	of_get_next_parent
	of_graph_get_remote_node
	of_get_next_child
	of_cpu_device_node_get

Table 8: Bugs detected by APHP in Qemu, Git and Redis, Col Status indicates the status of the patch with S,C,A,indicating submitted, confirmed, accepted, respectively.

System	Buggy function	Impact	Status
Qemu	mft_qom_set	memleak	A
Qemu	mft_qom_set	memleak	A
Qemu	ubi_get_volnum_by_name	memleak	S
Qemu	chip_init	memleak	S
Qemu	chip_init	memleak	S
Git	cmd_clean	memleak	C
Git	expire_commit_graphs	memleak	A
Redis	streamGetEdgeID	memleak	A

Table 9: 100 confirmed bugs in the bugs detected by APHP in Linux kernel, Col Status indicates the status of the patch with S,C,A,indicating submitted, confirmed, accepted, respectively

Buggy func	Impact	Status
mtk_mipi_tx_probe	null-ptr-deref	A
aspeed_vuart_probe	null-ptr-deref	A
__create_synth_event	memory leak	A
mlxbf_pmc_map_counters	reliability	A
anx7625_register_i2c_dummy_clients	reliability	A
dmi_sysfs_register_handle	refcount leak	A
vmbus_add_channel_kobj	refcount leak	A
pdcs_register_pathentries	refcount leak	A
ab8500_fg_sysfs_init	refcount leak	A
meson_smp_prepare_cpus	refcount leak	A
axxia_boot_secondary	refcount leak	A
ehci_hcd_ppc_of_probe	refcount leak	A
cns3xxx_init	refcount leak	A
zynq_get_revision	refcount leak	A
xive_spapr_init	refcount leak	A
xen_dt_guest_init	refcount leak	A
omap_gic_of_init	refcount leak	A
dpaa2_ptp_probe	refcount leak	A
tegra210_clock_init	refcount leak	A
tegra20_clock_init	refcount leak	A
tegra114_clock_init	refcount leak	A
versatile_reboot_probe	refcount leak	A
rockchip_grf_init	refcount leak	A
realview_gic_of_init	refcount leak	A
of_flash_probe_versatile	refcount leak	A
aspeed_adc_set_trim_data	refcount leak	A
scmi_regulator_probe	refcount leak	A
imx_sc_thermal_probe	refcount leak	A
spufs_init_isolated_loader	refcount leak	A
xive_get_max_prio	refcount leak	A
bcm4908_partitions_fw_offset	refcount leak	A
parse_redboot_of	refcount leak	A
of_get_devfreq_events	refcount leak	A
mv88e6xxx_mdios_register	refcount leak	A
mtk_pcie_init_irq_domains	refcount leak	A
max77620_initialise_fps	refcount leak	A
am65_cpsw_nuss_probe	refcount leak	A
am65_cpsw_init_cpts	refcount leak	A
aries_audio_probe	refcount leak	A
bcma_mdio_mii_register	refcount leak	A
meson_encoder_hdmi_init	refcount leak	A
meson_encoder_cvbs_init	refcount leak	A
mdp4_modeset_init_intf	refcount leak	A
ti_dra7_xbar_route_allocate	refcount leak	A
qcom_smd_parse_edge	refcount leak	A
omap2430_probe	refcount leak	A
of_get_ocmem	refcount leak	A
of_get_dram_timings	refcount leak	A
brcm_pcie_probe	refcount leak	A
xemaclite_of_probe	refcount leak	A

Buggy func	Impact	Status
snd_proto_probe	refcount leak	A
smsm_parse_ipc	refcount leak	A
smp2p_parse_ipc	refcount leak	A
sdma_event_remap	refcount leak	A
rockchip_pinctrl_probe	refcount leak	A
q6v5_alloc_memory_region	refcount leak	A
nmk_pinctrl_probe	refcount leak	A
mxs_sgtl5000_probe	refcount leak	A
mxs_saif_probe	refcount leak	A
mtk_pctrl_init	refcount leak	A
imx8m_probe	refcount leak	A
fsl_rio_setup	refcount leak	A
dvic_probe_of	refcount leak	A
atmel_ebi_probe	refcount leak	A
ath10k_setup_msa_resources	refcount leak	A
a6xx_gpu_init	refcount leak	A
sysfb_create_simplefb	memory leak	A
pcm030_fabric_probe	memory leak	A
dwc3_qcom_acpi_register_core	memory leak	A
softingcs_probe	memory leak	A
ti_dra7_xbar_route_allocate	memory leak	A
octeon_cf_probe	memory leak	A
mtk_smi_device_link_common	memory leak	A
meson_encoder_hdmi_init	memory leak	A
tegra_smmu_find	memory leak	A
tegra_dsi_ganged_probe	memory leak	A
sun8i_hdmi_phy_get	memory leak	A
qmp_get	memory leak	A
of_get_ocmem	memory leak	A
msm_hdmi_get_phy	memory leak	A
ingenic_ecc_get	memory leak	A
imx_hdmi_probe	memory leak	A
fman_port_probe	memory leak	A
emc_ensure_emc_driver	memory leak	A
dsi_get_phy	memory leak	A
coda_get_vdoa_data	memory leak	A
imx_sgtl5000_probe	refcount leak	A
vc4_dsi_encoder_enable	refcount leak	A
rti_wdt_probe	reliability	A
omap4_keypad_probe	reliability	A
imx_clk_scu_probe	reliability	A
ssc_probe	reliability	A
wkup_m3_ipc_probe	reliability	A
sni_82596_probe	reliability	A
qcom_slim_probe	reliability	A
mpc8xxx_probe	reliability	A
meson_spicc_probe	reliability	A
idt_gpio_probe	reliability	A
dwc2_hcd_init	memory leak	A
tw686x_video_init	memory leak	A

Table 10: List of 50 API pairs identified by APHP. Each row displays the program name, target API, post-operation API, whether the API pair is documented, and the number of corresponding new bugs identified by APHP.

Program	Target API	Post-operation API	InDoc?	#New bugs
Kernel	of_parse_phandle	of_node_put	✓	82
Kernel	of_find_compatible_node	of_node_put	✓	40
Kernel	of_find_device_by_node	put_device	✗	26
Kernel	kobject_init_and_add	kobject_put	✓	11
Kernel	of_get_child_by_name	of_node_put	✓	17
Kernel	clk_prepare_enable	clk_disable_unprepare	✗	39
Kernel	of_find_node_by_path	of_node_put	✗	19
Kernel	of_find_node_by_name	of_node_put	✓	4
Kernel	of_find_matching_node	of_node_put	✗	25
Kernel	pm_runtime_enable	pm_runtime_disable	✗	44
Kernel	pm_runtime_get_sync	pm_runtime_put_sync	✗	6
Kernel	ioremap	iounmap	✗	4
Kernel	opendir	closedir	✗	4
Kernel	strdup	free	✓	2
Kernel	of_get_next_parent	of_node_put	✓	2
Kernel	argv_split	argv_free	✗	2
Kernel	of_find_node_by_phandle	of_node_put	✓	3
Kernel	of_graph_get_remote_node	of_node_put	✓	3
Kernel	of_find_matching_node_and_match	of_node_put	✓	6
Kernel	platform_device_alloc	platform_device_put	✗	8
Kernel	of_iomap	iounmap	✗	8
Kernel	edac_mc_alloc	edac_mc_free	✓	1
Kernel	of_graph_get_remote_port_parent	of_node_put	✓	1
Kernel	of_cpu_device_node_get	of_node_put	✗	1
Kernel	crypto_alloc_shash	crypto_free_shash	✓	1
Kernel	of_phy_find_device	put_device	✗	1
Kernel	device_register	put_device	✓	1
Kernel	of_get_next_child	of_node_put	✓	1
Kernel	device_initialize	put_device	✓	1
Kernel	usb_create_hcd	usb_put_hcd	✗	1
Kernel	video_device_alloc	video_device_release	✗	1
Kernel	sp_get_irqs	sp_free_irqs	✗	1
QEMU	opendir	closedir	✗	2
QEMU	g_strdup_printf	g_free	✓	1
QEMU	g_new0	g_free	✗	1
QEMU	qtest_qmp	qobject_unref	✗	1
QEMU	asprintf	free	✓	1
Git	opendir	closedir	✗	1
Git	parse_pathspec	clear_pathspec	✗	1
Redis	streamIteratorStart	streamIteratorStop	✓	1
Kernel	kmalloc	kfree	✗	-
Kernel	pci_enable_pcie_error_reporting	pci_disable_pcie_error_reporting	✗	-
Kernel	nand_scan	nand_cleanup	✗	-
Kernel	clk_get	clk_put	✗	-
Kernel	class_find_device	put_device	✓	-
Kernel	pci_get_slot	pci_dev_put	✓	-
Kernel	regulator_enable	regulator_disable	✓	-
Kernel	d_find_alias	dput	✗	-
Kernel	kstrdup	kfree	✗	-
Kernel	kasprintf	kfree	✗	-