Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths

Dinghao Liu
dinghao.liu@zju.edu.cn
Zhejiang University

Qiushi Wu
wu000273@umn.edu
University of Minnesota

Shouling Ji*
sji@zju.edu.cn
Zhejiang University & Binjiang Institution of Zhejiang University

Kangjie Lu
kjlu@umn.edu
University of Minnesota

Zhenguang Liu
liuzhenguang2008@gmail.com
Zhejiang University

Jianhai Chen
chenjh919@zju.edu.cn
Zhejiang University

Qinming He*
hqm@zju.edu.cn
Zhejiang University

Abstract

Missing a security operation such as a bound check has been a major cause of security-critical bugs. Automatically checking whether the code misses a security operation in large programs is challenging since it has to understand whether the security operation is indeed necessary in the context. Recent methods typically employ cross-checking to identify deviations as security bugs, which collects functionally similar program slices and infers missed security operations through majority-voting. An inherent limitation of such approaches is that they heavily rely on a substantial number of similar code pieces to enable cross-checking. In practice, many code pieces are unique, and thus we may be unable to find adequate similar code snippets to utilize cross-checking.

In this paper, we present IPPO (Inconsistent Path Pairs as a bug Oracle), a static analysis framework for detecting security bugs based on differential checking. IPPO defines several novel rules to identify code paths that share similar semantics with respect to an object, and collects them as similar-path pairs. It then investigates the path pairs for identifying inconsistent security operations with respect to the object. If one path in a path pair enforces a security operation while the other does not, IPPO reports it as a potential security bug. By utilizing on object-based path-similarity analysis, IPPO achieves a higher precision, compared to conventional code-similarity analysis methods. Through differential checking of a similar-path pair, IPPO eliminates the requirement of constructing a large number of similar code pieces, addressing the limitation of traditional cross-checking approaches. We implemented IPPO and extensively evaluated it on four widely used open-source programs: Linux kernel, OpenSSL library, FreeBSD kernel, and PHP. IPPO found 154, 5, 1, and 1 new security bugs in the above systems, respectively. We have submitted patches for all these bugs, and 136 of them have been accepted by corresponding maintainers. The results confirm the effectiveness and usefulness of IPPO in practice.

CCS Concepts

• **Security and privacy** → **Systems security**; **Software and application security**.

Keywords

Bug Detection; Similar Path; Missing Security Operation; Static Analysis

ACM Reference Format:
Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3485373

1 Introduction

Large-scale programs usually enforce various kinds of security operations (*e.g., security checks, locks,* and *reference counting*) to ensure the safety. Correctly using them greatly improves the efficiency and security of a complex system. However, missing a security operation is common in large programs, which may lead to severe security issues. Specifically, according to the statistics in [44], missing security operations is the cause of 61% vulnerabilities in the national vulnerability database (NVD). Consistent with the findings in [44], we empirically scrutinized recent vulnerabilities in Linux kernels and found that around 66% of them are caused by missing security operations. Existing works ([24, 26, 28, 36, 42, 50]) have also studied the security issues of such bugs in detail, including permission bypass [5], out-of-bound access [6], high power consumption [8], deadlock [7], system crash [2], *etc.*

Though missing security operations can lead to serious consequences, detecting them is difficult as it has to determine whether

*Shouling Ji and Qinming He are the co-corresponding authors.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485373

the missed security operations are indeed necessary in a specific context. This, however, requires not only elaborate checking rules, but also precise and scalable analysis of the complicated data flows and contexts. There is still a lack of oracles for detecting missed security operations. To address this problem, researchers have turned to cross-checking to automatically decide whether a security operation is needed. Specifically, a general cross-checking based method consists of two steps: (1) it first collects a substantial number of functionally or semantically similar code pieces, (2) then it checks the behaviors of security operations across these code slices. Once we find that the majority of the code pieces have enforced a security operation, we assume that the majority is correct and report the minority cases that miss the security operation as bugs. The main advantage of cross-checking is that we can avoid direct code semantic understanding. Previous works like Juxta [27], Crix [24], APISan [48], Engler [14], and EECatch [30] employed cross-checking to infer bugs.

However, the cross-checking technique would suffer from unavoidable false negatives due to the following facts. (1) Many code pieces may be unique, and thus we may not be able to find enough similar cases to enable cross-checking (*e.g.*, the *one-to-one inconsistency* mentioned in FICS [9]). In practice, most cross-checking tools will set a threshold to label the majority patterns, usually 0.8 (*e.g.*, 0.85 in Crix [24] and 0.8 in APISan [48]), which means that we need at least four similar correct samples to pick out one inconsistent bug effectively. Figure 1 shows a counterexample, where both allocation and release functions are only called once in the entire Linux kernel. As a result, we cannot infer it as a bug through majority-voting. (2) The granularity of code slicing is hard to control. In order to make cross-checking scalable enough to deal with large programs, existing methods usually abstract their code representation or use specific rules to limit slice generation (*e.g.*, Simplified Program Dependence Graph in FICS [9]). Such strategies make the code slicing coarse-grained and lose some valuable code snippets. (3) The hypothesis that the majority is correct might not always hold. For example, it is possible that some poorly documented APIs are often misused (*e.g.*, `pm_runtime_get_sync()` and `kobject_init_and_add()`) [10, 26]. In that case, cross-checking would not flag the common misuses as potential bugs.

In this paper, we present IPPO (Inconsistent Path Pairs as a bug Oracle), a security bug detection framework that requires only one pair of similar code paths to determine if a path misses a security operation. IPPO's detection is based on the observation that if a pair of paths are semantically similar with respect to an object, they are expected to enforce the same security operations against the object. Given a similar-path pair, if a path enforces a security operation while the other does not, IPPO reports it as a potential security bug. By introducing the *object-based* path-similarity analysis, IPPO achieves a higher precision than conventional code-similarity analysis. Meanwhile, unlike traditional cross-checking methods, IPPO conducts differential checking on each similar-path pair and on longer requires constructing a large number of similar code pieces.

A key challenge in realizing the idea of IPPO is to construct the object-based similar-path pairs (OSPP). On the one hand, the specific semantics and contexts of different paths are complex, and the usages of an object could be diverse. Thus, it is challenging to automatically understand the semantics and contexts of the paths. Moreover, it is

```
/* drivers/infiniband/hw/usnic/usnic_ib_verb.c */
static struct usnic_ib_qp_grp*
find_free_vf_and_create_qp_grp(...)
{
    ...
    if (usnic_ib_share_vf) {
        /* Try to find resouces on a used vf which is in pd */
        dev_list = usnic_uiom_get_dev_list(pd->umem_pd);
        ...
        if (!usnic_vnic_check_room(vnic, res_spec)) {
            ...
            qp_grp = usnic_ib_qp_grp_create(...);
            goto qp_grp_check;
        }
        ...
        usnic_uiom_free_dev_list(dev_list);
    }
    ...
    qp_grp = usnic_ib_qp_grp_create(...);
    goto qp_grp_check;
    ...
    return ERR_PTR(-ENOMEM);

qp_grp_check:
    if (IS_ERR_OR_NULL(qp_grp)) {
        usnic_err("Failed to allocate qp_grp\n");
        return ERR_PTR(qp_grp ? PTR_ERR(qp_grp) : -ENOMEM);
    }
    return qp_grp;
}
```

Figure 1: A memleak bug identified by IPPO. If usnic_ib_qp_grp_create() fails at the first call, dev_list will not be freed on error.

not easy to determine whether two code paths should be considered similar. On the other hand, analyzing and collecting OSPP is likely to encounter path explosion, especially in large functions. To address the first challenge, we develop multiple rules to characterize the features of code paths in an object-level granularity. As for the second challenge, we develop techniques to reduce path redundancy, *e.g.*, we partition the control-flow graph (CFG) and reduce redundant structures in similar paths to address the pair- and path-explosion problems. With the defined rules and techniques, IPPO is able to precisely and scalably construct OSPP in even large functions.

We have implemented IPPO as several LLVM static analysis passes. We chose four widely used open-source programs to extensively evaluate our method: the *Linux kernel*, the *OpenSSL library*, the *FreeBSD kernel*, and *PHP* . Two run in the kernel mode, and the other two run in the user mode. Each bug in them will influence a massive number of users and devices. IPPO finished the whole analysis for all programs within two hours and reported 754 missing security operation cases. By manually checking all of them, we finally confirmed 154, 5, 1, and 1 new security bugs in the above systems, respectively, including 82 refcount leak bugs, 57 memleak bugs, 10 missing check bugs, 7 use-after-free bugs, and 5 missing unlock bugs. We have submitted patches for the new bugs, and 136 of them have been accepted by community maintainers. The results confirm the effectiveness, scalability, and portability of IPPO. In summary, the key contributions of this work are:

- **A new system for detecting missed security operations.** We propose a new bug detection framework, IPPO, to address the important limitations of traditional cross-checking. The missed security operation detection requires only a pair of code paths rather than a substantial number of similar code pieces. We implemented IPPO and it supports further extension and flexible customization on specific kind of security operations. We will open source IPPO[1] to facilitate further researches.

- **New techniques for constructing object-based similar-path pairs.** An important technical challenge in IPPO is to construct path pairs that are semantically similar. To improve the precision, we develop the object-based similarity analysis. We also develop a set of rules to refine the similar path construction. In addition, we propose return value-based sub-CFG and reduced similar path to address the path-explosion problem.
- **Finding and fixing numerous new bugs.** With IPPO, we found numerous new bugs in the Linux kernel, the OpenSSL library, the FreeBSD kernel, and PHP, which could cause various security and reliability issues. We have reported these bugs, and most of them have been fixed by working with the community maintainers.

2 Background and Motivation

2.1 Missing Security Operation Bugs

In this paper, we focus on the missed security operations in similar-path pairs. Missing security operations introduces bugs when such security operations are indispensable in a specific context. Figure 2 shows a memory leak bug (CVE-2019-8980 [3]). variable `buf` allocated at line 6 is not freed on failure of `kernel_read()` at line 8, which allows attacks to cause a DoS by triggering vfs_read failures.

```
1  /* fs/exec.c */
2  int kernel_read_file(...)
3  {
4      ...
5      if (id != READING_FIRMWARE_PREALLOC_BUFFER)
6          *buf = vmalloc(i_size);
7      ...
8      bytes = kernel_read(file, *buf + pos, i_size - pos, &pos);
9      if (bytes < 0) {
10         ret = bytes;
11         goto out;
12     }
13     ...
14 out_free:
15     if (ret < 0) {
16         if (id != READING_FIRMWARE_PREALLOC_BUFFER) {
17             vfree(*buf);
18             *buf = NULL;
19         }
20     }
21
22 out:
23     allow_write_access(file);
24     return ret;
25 }
```

Figure 2: CVE-2019-8980, a missing null check vulnerability.

Though uncommon, *redundant* security operations are also possible to cause security issues. Figure 3 shows CVE-2019-12819 [1], where `put_device()` at line 8 is redundant because the caller of `__mdiobus_register()` (e.g., `xlr_setup_mdio()`) will call other functions to free the `bus` on its failure (`mdiobus_free()` at line 21). As a result, use-after-free occurs when `bus` is accessed again. This inconsistent case could also be caught by comparing the usage of `device_register()` in other paths. The two examples show that the inconsistency between two similar paths can reliably reveal the bugs. When we find a missing security operation case in a similar path pair, usually, there is a bug caused by the missed or redundant security operation.

[1] https://github.com/dinghaoliu/IPPO

```
1  /* drivers/net/ethernet/agere/et131x.c */
2  int __mdiobus_register(struct mii_bus *bus, struct module *owner)
3  {
4      ...
5      err = device_register(&bus->dev);
6      if (err) {
7          pr_err("mii_bus %s failed to register\n", bus->id);
8  -       put_device(&bus->dev); //Redundant release
9          return -EINVAL;
10     }
11     ...
12 }
13
14 /* drivers/staging/netlogic/xlr_net.c */
15 static int xlr_setup_mdio(struct xlr_net_priv *priv,
16                 struct platform_device *pdev)
17 {
18     ...
19     err = mdiobus_register(priv->mii_bus);
20     if (err) {
21         mdiobus_free(priv->mii_bus);
22         pr_err("mdio bus registration failed\n");
23         return err;
24     }
25     ...
26 }
```

Figure 3: CVE-2019-12819, a use-after-free vulnerability.

2.2 Impact of Missing Security Operations

We collected recently published Linux kernel vulnerabilities (published between January and August during 2019) in CVE Details [4] to analyze the impact of missing security operations. We finally screened out 121 vulnerabilities with certain security impacts and valid patches. Among these vulnerabilities, we found that 69 (57.0%) of them are fixed by adding missed security operations directly. Another 10 (8.3%) vulnerabilities are fixed by adjusting the position of security operations, which can also be regarded as a kind of missing security operations at a specific path location. Three (2.5%) vulnerabilities are caused by redundant security operations. The missed security operations including security checks, variable initiation/nullification, resource release, memory cleaning, refcount operations, unlock, and some other critical APIs. As for the impact of these missed security operations, they could lead to DoS, memory corruption, information leak, overflow, privilege gaining, and code execution. 25 of these vulnerabilities have a CVSS score more than 7, and six have a CVSS core of 10 (the highest security level).

2.3 Causes of Missing Security Operations

Based on our study of existing bugs, the causes of missing security operations can be roughly classified into two categories. (1) *Complicated program logic.* With the growth of the program scale, its execution paths increase exponentially, which makes it difficult for developers to carefully review all the paths. According to our analysis, even a single function in Linux kernel could have hundreds lines of source code. The longest bug function detected by IPPO contains 613 lines of code, which is difficult for manual review. Therefore, it is easy for developers to forget to apply necessary security operations or fixing-patches while developing such complicated program logic. (2) *Poorly designed security related APIs.* The misleading designs of security related APIs could easily make developers misuse them and introduce bugs due to missed security operations. Among the refcount leak bugs found by IPPO, 87% bugs are caused by poorly designed refcount APIs (mainly `pm_runtime_get_sync()`), even though such APIs have a sound document. We will discuss the details of this problem in §6.

2.4 Detecting Missed Security Operations

To determine whether a missed security operation is necessary, the easiest way is to compare the missed cases with existing bug samples, which is known as *bug localization* [34, 47]. Such approaches are mainly used to pick out bugs in different program versions and is incapable to detect new bugs. Collecting a large number of bug samples itself is also a challenging task.

Another method is to utilize statistical information to infer the necessity of the missed security operations [24, 48], where the majority cases are considered as correct, known as *cross-checking*. Usually this approach needs a large amount of use cases as its samples and may not work well when dealing with uncommon cases. Some other works [17, 26, 36, 50] direct their efforts at specific kinds of missed security operations, which, unfortunately, limits their scalability and portability.

3 Overview

The goal of IPPO is to identify the missed essential security operations in a target program as security bugs. The key challenge is to determine whether a security operation is really indispensable in the context, which requires the understanding of code semantics and contexts. IPPO addresses this challenge by modeling the similarity of different paths: if two paths share similar functionality with respect to a specific object, then their usages of security operations against that object are supposed to be consistent. Our approach could work under the scenarios where there is only a very limited number of code pieces available (*e.g.,* the code piece in Figure 1). The overall architecture of IPPO is outlined in Figure 4. At a high level, IPPO's workflow contains three phases:

In the first phase, IPPO generates a global call graph for the target program with the provided LLVM IR files, which is used to assist the security operation identification in the second phase. IPPO also builds control-flow graphs (CFGs) and unroll loops for one level for every function, which lays the base for path analysis.

In the second phase, IPPO analyzes the target program and accomplishes three tasks: (1) detecting all security operations in the target program; (2) extracting critical variables (objects) from the security operations; (3) identifying and collecting all similar-path pairs based on the critical objects in each function. After that, IPPO constructs similar-path pairs that may have inconsistent security operations. These paths should be similar with respect to the contexts and semantics. To this end, we present the idea of *object-based similar-path pair* (OSPP) to characterize the similarity of such paths. IPPO adopts several new path-sensitive and semantic-sensitive techniques to collect OSPP within a function effectively, which is discussed in detail in §4.2.

In the third phase, IPPO checks the missed security operations in the collected OSPPs, and generates bug reports for further manual confirmation.

4 Object-based Similar-Path Pairing

In this section, we present the design principles of object-based similar-path pairs (OSPP), together with the related analysis techniques. Other techniques and implementation details will be presented in §5.

4.1 Extracting Objects

One key component of constructing OSPP is object. In this paper, we extract critical variables from security operations as our target objects. Currently, we aim at four kinds of widely used security operations: *security checks*, *resource alloc/release*, *reference count operations*, and *lock/unlock*. They represent the most common cases and missing them has contributed to the most common and critical classes of bugs. The target critical objects extracted from these security operations are the *checked variables*, *resource variables*, *reference counters*, and *lock variables* respectively. The first kind of objects could be extracted from the `if` statements, while the last three kinds of objects could be extracted from the function arguments directly.

4.2 Design Principles of OSPP

Code representation. The granularity of code representation determines the upper bound of bug detection. We need to choose a fine-grained code representation to make up for lacking subsequent similar code slices. In this paper, we select the *control-flow path* as the basic unit while modeling similar code pieces. It contains relatively more abundant semantic information and makes our analysis path-sensitive. Specifically, a control-flow path consists of a series of coherent basic blocks in a control-flow graph (CFG).

Key insights. The most important component in IPPO is the construction of similar-path pairs. The path-pair construction is challenging for the following reasons. First, if the similarity analysis is too permissive, non-similar paths may be paired, resulting in many false positives; if the similarity analysis is too strict, valuable similar path pairs can be missed, leading to many false negatives in bug detection. Therefore, we need to design a new similarity analysis method that can precisely and broadly identify similar-path pairs. Our insight is that the ultimate goal of IPPO is to identify a missed security operation against an object, so *the similarity should be based on the particular object*. As long as two code paths have similar semantics for the object, object-irrelevant semantics in the code paths should not be considered while modeling the similarity. This way, we improve both the precision and coverage. Based on this insight, we propose *object-based similar-path pairs*. Second, we still lack a concrete criteria for determining whether two paths have the similar semantics and contexts for an object. To address this problem, we develop a set of rules for constructing OSPP.

Rules for constructing OSPP. Our intuition for determining the object-based path similarity is that *whether a security operation should be enforced against an object depends on (1) the semantics against the object and (2) the contexts of the semantics.* Therefore, we study existing bugs and empirically develop a set of rules that ensure the similarity of semantics and contexts of an object. Each rule is described and justified as follows.

For illustration purpose, we introduce a real bug as an example (Figure 5). In function `snd_echo_resume()`, there are four error handling paths (end at line 10, 17, 24, and 32 respectively) adopting inconsistent release operations against variable `chip`, a member of variable `dev`. Paths which return at line 17 and line 32 free `chip` while the other two paths do not. Actually, freeing a non-local variable in the resume error paths here is redundant because it could lead to double-free at the next time the system goes to resume and crash the system. From the perspective of a developer, the aforementioned four

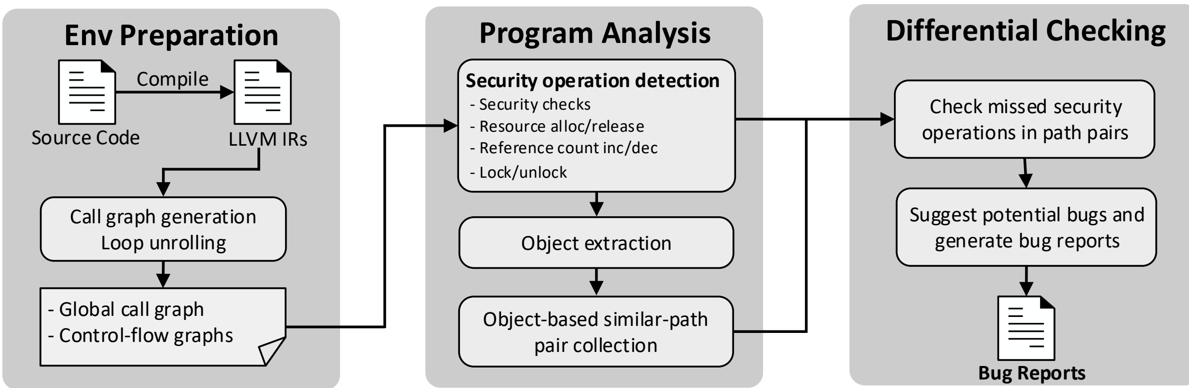

Figure 4: The overview of IPPO. IPPO has three analysis phases. It takes as input the LLVM IRs of the target program. It reports as output the potential bugs due to missing security operations.

```
1  /* sound/pci/echoaudio/echoaudio.c */
2  static int snd_echo_resume(struct device *dev)
3  {
4      struct echoaudio *chip = dev_get_drvdata(dev);
5      struct comm_page *commpage, *commpage_bak;
6      ...
7      commpage = chip->comm_page;
8      commpage_bak = kmemdup(commpage, sizeof(*commpage), GFP_KERNEL);
9      if (commpage_bak == NULL)
10         return -ENOMEM;
11
12     err = init_hw(chip, chip->pci->device, chip->pci->subsystem_device);
13     if (err < 0) {
14         kfree(commpage_bak);
15         dev_err(dev, "resume init_hw err=%d\n", err);
16         snd_echo_free(chip);
17         return err;
18     }
19     ...
20     err = restore_dsp_rettings(chip);
21     chip->pipe_alloc_mask = pipe_alloc_mask;
22     if (err < 0) {
23         kfree(commpage_bak);
24         return err;
25     }
26     ...
27     kfree(commpage_bak);
28     ...
29     if(request_irq(...)) {
30         dev_err(chip->card->dev, "cannot grab irq\n");
31         snd_echo_free(chip);
32         return -EBUSY;
33     }
34     ...
35     return 0;
36 }
```

Figure 5: A double-free vulnerability in the Linux kernel identified by IPPO.

paths are similar with respect to variable `chip` while determining if we should release it before returning. There is no special reason to enforce different treatments against `chip` in them, which could be used to detect security bugs. To model such object-based similarity, we extract four rules for constructing OSPP.

Rule 1: *The two paths start at the same block and end at the same block in CFG.* Put differently, the two paths share the same start point and end point. We observed that the closer two paths are, the more likely of existing one object be used similarly in them. Therefore, we aim at the nearest paths in a program: paths which share the same start and end blocks. In addition, the same start and end blocks also guarantee the semantic-integrity of the two paths. For example, in Figure 5, both error paths that start at line 22 and return at line 24 and line 32 (though the return line number is different, they exactly end at the same return block in CFG) have released `commpage_bak` and there is no bug here. However, if we consider these two paths having different start blocks (*e.g.*, one starts at line 22 and the other one starts at line 29), then we may lose important information (*e.g.*, releasing `commpage_bak` at line 27) and get wrong conclusion on bug analysis.

Rule 2: *The object has the same state in two paths.* Since the similarity we considered is object-based, we expect the object itself is equivalent across different paths. One of the most direct measurements is to evaluate the object's initialization point, namely, source. If the object's source is inside/outside both paths, we consider the state of the object in such paths are the same. The source of `chip` (sources from `dev` at line 4) in Figure 5 is outside all aforementioned error paths, which satisfies Rule 2. If the allocation (initialization) of `chip` is at line 26, then the error path returns at line 24 does not need to release it since the resource is even non-existent at that time.

Rule 3: *The two paths have the same SO-influential operations.* To further ensure that two paths have similar semantics with respect to both target objects and target security operations, we put forward the idea of *security operation-influential operations* (SO-influential operations): the operations that have an impact on whether we should enforce a specific security operation on a specific object. Table 1 shows the detailed definition of SO-influential operations. Specifically, security check is mainly used to terminate the execution flow on failure and returning an unchecked value usually does not constitute a bug. Therefore, we expect that there are other calculation tasks (function calls, arithmetic operations, memory operations, *etc.*) after the checked object. Resource alloc/release is under the influence of resource propagating operations (*e.g.*, the resource variable is propagated to global variables and there is a specialized callback function to release it). Refcount and lock/unlock is influenced by any other reference counter adjustment or lock state adjustment. SO-influential operations could determine whether a security operation is necessary in a path, thus are required to be used consistently in OSPP.

Table 1: SO-influential operations.

Security operation	SO-influential operation
Security check	Function calls, arithmetic and memory operations after the object (checked variable)
Resource alloc/release	Resource propagation
Refcount	Reference counter adjustment
Lock/unlock	Lock state adjustment

Rule 4: *The two paths have the same sets of pre- and post-conditions against the object.* Pre- and post-conditions are the assumptions before and after we executing a path, which are expected to be the same for OSPP. APISan [48] uses them to characterize API usage patterns. We redefine them in this paper to characterize the semantics of paths. The pre-condition of a path is its branch condition (*e.g.*, variable `err` is the pre-condition of the path starts at line 13 in Figure 5), while the post-condition is the path's impact on the return value. We require that the pre-condition of OSPP must be object-irrelevant, otherwise the semantic against the object is obviously diverse. For example, in Figure 5, there is another inconsistency besides releasing `chip`: releasing `compage_bak`. The error path ends at line 9 misses release against `compage_bak` while all the following paths contain such release. However, the release operation is not necessary in this error path since its pre-condition is the null check result against `compage_bak` (the target object), and a null pointer needs no further release. The same post-condition ensures two paths share the same functionality (either normal functionality or error handling).

Analysis challenges—path and pair explosion. While the rules define the object-based path similarity, checking against these rules is still hard for the following problems. (1) *Path explosion.* Analyzing OSPP requires us to collect paths as pairs first. A direct idea of path collection is to collect all paths start at the entry and end at the exit in a function (which satisfies Rule 1 of OSPP). Nevertheless, such an approach will result in path explosion in large functions and make further analysis impractical. There could also be a lot of redundant information if all paths are collected in this way. (2) *Pair explosion.* Some path pairs may only satisfy partial OSPP rules, but we could not simply discard them because it is possible that they can be paired with other paths. Given the large number of paths, it is hard to comprehensively analyze all possible valid OSPPs.

Our solution—path reduction and graph partitioning. To address the aforementioned challenges, we come up with the following techniques. (1) We observed that the main cause of path explosion is the redundant common messages. Hence, we collect path pairs that satisfy Rule 1 in a new way: we only collect paths that share no common basic blocks besides the start block and the end block, which is referred to as *reduced similar paths* (RSPs) in this paper. We design a two-phase RSP collection algorithm to collect RSPs efficiently. (2) We choose to divide the CFG of a function into different parts, and paths in each part share the same return value, which are referred to as *return value–based graphs* (RVGs) in this paper. Paths collected from RVGs inherently satisfy the post-condition of Rule 4. The two strategies can effectively address both the path explosion and the pair explosion.

OSPP construction flow. To construct OSPPs in a function, we need to construct path pairs that satisfy Rules 1~4 of OSPP. We first generate return value-based graphs (RVGs) from a given CFG. Then, we collect the reduced similar paths (RSPs) from RVGs and pair them. The definitions of RSP and RVG inherently guarantee Rule 1 and the post-condition of Rule 4. Finally, we check the rest OSPP rules (Rule 2, Rule 3, and the pre-condition of Rule 4) against the RSPs and extract valid OSPPs from them. The following sections will present the technical details of OSPP construction.

4.3 Generating Return Value-based Graphs

4.3.1 Identifying Error Edges. Error edges are the key components for return value-based graph (RVG) generation. In this paper, we mainly consider two kinds of return values as post-conditions: normal values and error values, which indicate two most common functionalities: normal functionality and error handling functionality. If a path returns an error code or calls an error handling function, it is considered as an error handling path. An error CFG edge should satisfy one of the following conditions: (1) it connects to an error return value or error handling function, or (2) all its following edges connect to some error return values or error handling functions. Existing researches (Crix [24], EECatch [30]) have studied error handling functions. We will discuss error return value in detail in §5.2.1. `IPPO` uses a backward data-flow analysis starting from the return instruction to find the source of error return values and then uses a forward data-flow analysis starting from the source to collect all error edges of a CFG. These error edges are recorded in a global set (`ErrEdgeSet`).

4.3.2 Generating Sub-CFGs based on Return Values. Since we only consider two kinds of return values, `IPPO` generates two RVGs for each function: one graph contains all error handling paths and the other contains all normal paths without any error handling path. Algorithm 1 shows how to generate these two RVGs. It takes as input the CFG of the target function and an error edge set (`ErrEdgeSet`). The algorithm generates two graphs as output: `NormalRVG` and `ErrRVG`, the normal RVG and error handling RVG, respectively.

Generating normal RVG. The strategy to generate a normal RVG is simple: pruning all error edges from the original CFG (lines 4-6) and the pruned graph is the normal RVG. Given an input edge, the `PruneEdge()` method breaks the connection between the two endpoints of this edge. If the tail end block of the edge has no predecessor after pruning, which means it cannot be accessed from the entry anymore, `PruneEdge()` recursively prunes its successor edges. Since this pruned graph contains no error edges, all paths in this graph must be normal paths.

Generating error handling RVG. The basic idea of this part is to generate a complete graph from a part of discrete error edges. For each edge in `ErrEdgeSet`, Algorithm 1 adds it to the output `ErrRVG` (lines 8-9) and checks if this edge has reached the entry block (line 12) or the end block (line 15) of the CFG. If not, `IPPO` selects one predecessor or one successor edge and adds it to the next loop (line 13 and line 16). The `ErrRVG` will be extended in this procedure and finally reaches both the entry block and the end block of the CFG.

Algorithm 1: Generate error RVGs

```
GenErrRVG(ErrEdgeSet);
Input: CFG: Control-flow graph of the target function;
       ErrEdgeSet: Error edge set of the target function
Output: NormalRVG: Normal RVG;
        ErrRVG: Error handling RVG
ErrRVG ← ∅;
NormalRVG ← CFG;
foreach edge ∈ ErrEdgeSet do
    NormalRVG.PruneEdge(edge);
end
while Is_Not_Empty(ErrEdgeSet) do
    CE ← pop top element from ErrEdgeSet;
    ErrRVG.ADD_EDGE(CE);
    FB ← front end block of CE;
    TB ← tail end block of CE;
    if ∃(edge ∈ CFG) ends at FB and edge ∉ ErrRVG then
        ErrEdgeSet.PUSH_BACK(edge);
    end
    if ∃(edge ∈ CFG) starts at TB and edge ∉ ErrRVG then
        ErrEdgeSet.PUSH_BACK(edge);
    end
end
return NormalRVG, ErrRVG;
```

Algorithm 2: Collect RSPs - Collecting phase

```
1  RecurCollect(EB, CP);
   Input: EB: Entry block of current analysis;
          CP: Current path on collection
2  EB ← Entry block of input RVG;
3  RSPGSet ← CPG ← CP ← ∅ (Init once on the first call);
4  if EB has multiple predecessors or EB has no successor then
5      if CP ≠ ∅ then CP.PUSH_BACK(EB) ;
6      return;
7  else if EB has one successor then
8      if CP ≠ ∅ then CP.PUSH_BACK(EB) ;
9      RecurCollect(successor, CP);
10     return;
11 else // EB has multiple successors
12     foreach successor of EB do
13         new_path ← ∅;
14         new_path.PUSH_BACK(EB);
15         RecurCollect(successor, new_path);
16         CPG ← {new_path} ∪ CPG;
17     end
18     reserved_path ← RecurGroup(CPG);
19     if CP ≠ ∅ then CP.PUSH_BACK(reserved_path) ;
20     EB ← end block of reserved_path;
21     RecurCollect(EB, CP);
22 end
```

4.4 Collecting Reduced Similar Paths

So far, we have obtained graphs whose paths all share the same kind of return values. Then, `IPPO` needs to collect and group the reduced similar paths (RSPs) in these graphs. Based on the definition of RSPs, we design a method to collect RSPs in an RVG. It takes as input the entry block of RVG and it could group all RSPs in the graph and produce a set of these groups as output. Specifically, the RSP-collection method consists of two phases: *collecting phase* and *grouping phase*.

In the collecting phase, `IPPO` collects paths that start from the same block and end at some potential merge blocks, as shown in Algorithm 2. In the grouping phase, `IPPO` reviews if the paths collected in the previous phase are valid (merged at the same block), groups valid paths, and prunes the RVG, as shown in Algorithm 3. The collection results will be recorded in a global path group set: `RSPGSet`, in which every element is a group of valid RSPs. Both of the two phases are implemented recursively. Then, we present the details of the two phases below.

Collecting phase. Given the entry block of a RVG (EB), Algorithm 2 firstly checks if EB is a merge block (lines 4-6), in which case we should terminate the collection. The key feature of a merge block is the number of its predecessors and successors. When the EB has multiple predecessors, it must be a merge block of some RSPs. When the EB has no successor, it is the return block of the RVG. In these two cases, we should terminate the collection and return. Then, we check the successors of the EB to determine whether it is a start block. If the EB has only one successor, we recursively collect its successor (lines 7-10) until we meet one of the two aforementioned termination conditions. If there is a path on collection in above cases (lines 5 and 8), which means the current path collection serves a top path collection, we add the EB to the tail of this path.

When the EB has multiple successors, which means it is a start block of some RSPs, we recursively collect the RSPs for all of its successors and call Algorithm 3 to record them (lines 12-21). In order to speed up the collection, Algorithm 3 will select only one `reserved_path` from this group to be reserved and prune all other paths from the CFG. Each call of this algorithm simplifies the function's CFG and makes the subsequent collection faster. Then, Algorithm 2 starts a new collection from the end block of `reserved_path` (lines 20-21). If there is a path on collection, we push `reserved_path` to its tail (line 19).

Grouping phase. Algorithm 3 checks if the collected paths merge at the same block. If it is the case (line 2), then these paths satisfy all the requirements to be RSPs and we group them into the output set (line 3). Another task here is to prune the RVG. Since we have traversed and collected a group of RSPs, we do not need to traverse them again in the subsequent collection. Only one RSP is enough to represent this group (line 4). Function `PrunePath(reversed_path, CPG, CFG)` prunes all paths in CPG from the RVG except for a `reserved_path` (line 5). More specifically, for each path to be pruned, `PrunePath()` pruned all edges of this path using the `PruneEdge()` method mentioned in §4.3.2.

However, if the collected paths merge at different blocks (line 8), Algorithm 3 will rearrange the collection pace to make sure the collected paths merge at one block. Firstly, the algorithm checks all the collected paths and picks out RSPs that have already merged at one block (lines 9-17). Secondly, for the paths end at different blocks, the algorithm selects a `Top_block` from these ending blocks (line 18) to start a new collection (lines 19-31). Function `GetTopBlock()` finds the topmost block as `Top_block` in the CFG from the input block set.

Algorithm 3: Collect RSPs - Grouping phase

```
RecurGroup(CPG);
Input: CPG: Path group that just finished collection
Output: reserved_path: Path that selected as reservation
if ∀ (paths ∈ CPG) merge at one mergeblock then
    RSPGSet ← {CPG} ∪ RSPGSet;
    reserved_path ← select one path in CPG;
    PrunePath(reserved_path, CPG, CFG);
    CPG ← ∅;
    return reserved_path;
else // Paths merge at different mergeblocks
    MBSet ← all different mergeblocks of CPG;
    foreach mergeblock ∈ MBSet do
        if more than one paths ∈ CPG end at mergeblock then
            new_CPG ← paths ∈ CPG end at mergeblock;
            RSPGSet ← {new_CPG} ∪ RSPGSet;
            reserved_path ← select one path in new_CPG;
            PrunePath(reserved_path, CPG, CFG);
        end
    end
    Top_block ← GetTopBlokc(MBSet);
    new_CPG ← ∅;
    foreach mergeblock ∈ MBSet do
        if mergeblock ≠ Top_block then
            new_CPG ← {path ends at mergeblock} ∪ new_CPG;
        else
            reserved_path ← {path ends at mergeblock};
            new_path ← ∅;
            new_path.push_back(mergeblock);
            RecurCollect(mergeblock, new_path);
            new_CPG ← {new_path} ∪ new_CPG;
        end
    end
    RecurGroup(new_CPG);
    return reserved_path;
end
```

We use a constructed CFG example to illustrate how the path explosion happens and how the idea of RSP resolves the path explosion problem while analyzing paths in the Appendix.

4.5 Checking against OSPP Rules

At this step, IPPO further analyzes if the collected pairs of RSPs satisfy the rest rules of OSPP. The previous analysis phases have generated path pairs which satisfy Rule 1 and the post-conditions of Rule 4. Therefore, we enforce the following analysis to check against Rule 2, Rule 3, and the pre-conditions of Rule 4.

Rule 2 checking. The core of Rule 2 is to collect the source of an object. For resource alloc/release operations, we regard the allocation of the resource variable (object) as its source, which will be discussed in §5.1.3. For security checks, we use the source collection algorithm of Crix [24] to collect the source of an object. For refcount and lock/unlock operations, we regard the reference counter increments and lock operations as their sources. In this paper, we regard the return values of function calls with the same name as the same objects. Thus, there could be multiple sources for an object in a function. Once we get the source(s) of an object, we check if both paths contain the sources and discard path pairs that only one path of it contains the source while the other does not.

Rule 3 checking. We identify SO-influential operations according to the type of the target objects in both paths, as shown in Table 1. The checking procedures for security checks, refcount, and lock/unlock operations are relatively direct: matching the qualified instructions and APIs. For resource alloc/release, we trace the use chain of the resource variables and analyze if they propagate to any function parameters, global variables or return values. To make our method robust, we do not require the SO-influential operations in two paths to be exactly the same, but only existent or nonexistent in both paths.

Pre-condition checking. Since all collected paths start from the same block, the branch conditions of them are the same. We only need to make sure that the branch condition is object-irrelevant. Towards this, we firstly extract the branch condition from the start block of RSPs, then we analyze if the condition variable is exact the object or propagated from the object. We also observed that it is common to use function parameters or global variables to balance the pairwise used security operations. Thus, for resource alloc/release, refcount inc/dec, and lock/unlock, we currently discard the path pairs whose condition variables are propagated from them.

4.6 Workflow of IPPO

In this section, we use the vulnerability in Figure 5 as an example to show the workflow of IPPO on picking out this vulnerability. The workflow contains seven steps, as shown in Figure 6. Given an LLVM IR file as input, IPPO firstly generates the CFG for function `snd_echo_resume()` and detects all security operations in it (❶ in Figure 6). The identified resource release operations are marked blue in the CFG. IPPO extracts variables `chip` and `commpage_bak` from the release function `kfree()` and `snd_echo_free()` as target objects. Secondly, IPPO analyzes and marks the error edges in the CFG, which are shown in red (❷ in Figure 6). Thirdly, IPPO generates the two return value-based graphs (RVGs) from the CFG, and outputs an error handling RVG and a normal RVG (❸ in Figure 6). Since the normal RVG contains only one path, we do not need to further analyze it and just stop here. For the error handling RVG, IPPO collects reduced similar paths (RSPs) from it, which outputs three RSPs: RSP 1, RSP 2, and RSP 3 (❹ in Figure 6). For each object (`chip` and `commpage_bak`), IPPO checks the OSPP rules against the three RSPs (❺ in Figure 6). RSP 3 fails on the pre-condition checking of Rule 4, and is pruned from the analysis flow. Other RSPs are considered as object-based similar path pairs (OSPP). Finally, IPPO checks if one path in an OSPP contains the security operation (resource release) against the object while the other path does not (❻ in Figure 6). For object `chip`, IPPO finds that RSP 1 and RSP 3 have such a pattern, and generates a bug report to suggest that the error paths at line 10 and line 24 miss a release against variable `chip` in Figure 5 (❼ in Figure 6).

5 IPPO Implementation

We have implemented IPPO on top of LLVM, including a pass for unrolling loops and constructing the global call graph, a pass for finding function wrappers, a pass for detecting security operations, and a pass for OSPP analysis. IPPO in total contains 10K lines of

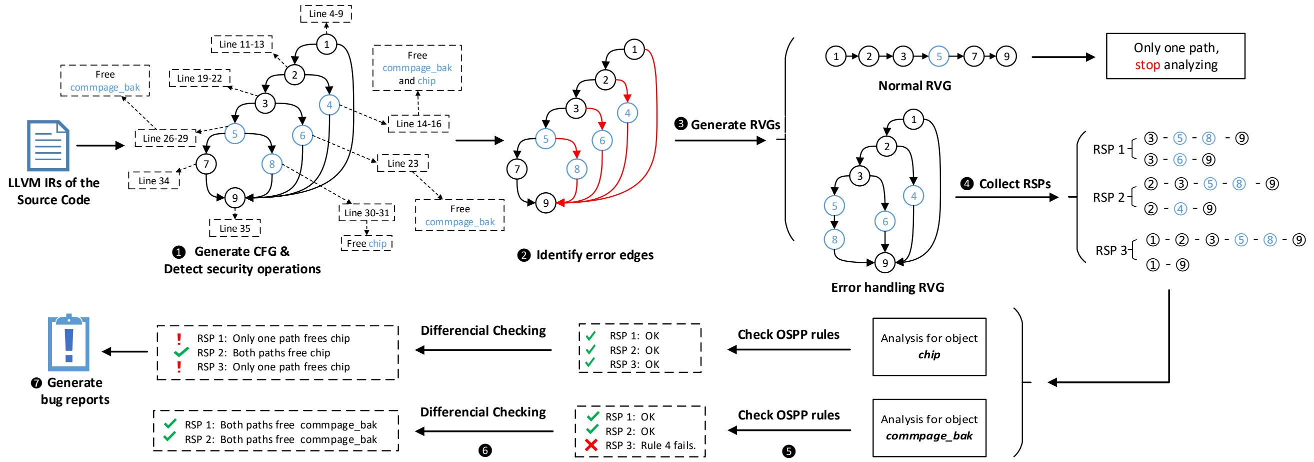

Figure 6: The workflow of IPPO while checking the double-free vulnerability in Figure 5.

C++ code. The rest of this section presents implementation details of IPPO.

5.1 Detecting Security Operations

To demonstrate how IPPO works, we experimented on detecting security checks, resource alloc/release operations, and reference count operations. Note that the security operation detection of IPPO is generic; once the patterns of security operations are provided, IPPO can be easily extended to detect other types of bugs.

5.1.1 Detecting Security Checks. Security check is a common used security operation and missing check causes a majority of recent security bugs [24, 44]. IPPO adopts the state-of-the-art security check detection method proposed in Crix [24]. Crix regards an `if` statement as a security check when one of its branch handles a failure and the other one continues the normal execution. We mainly consider the return value check, a subset of security check, in this paper. In particular, we found that Crix's security check detection method would miss some security checks when there exists a security check against a function call. We add an independent analysis flow to catch such cases and increase the total security check reports by around 20%. We also develop a more refined return value model while detecting security checks, which will be discussed in §5.2.

5.1.2 Detecting Reference Count Operations. We choose to detect inconsistent refcount operations in the PCI power management in the Linux kernel for the following reasons. (1) This set of APIs are most widely used [26] in subsystems of Linux kernel and could cause high power consumption and unexpected device suspending when used improperly. (2) We studied their documents and found that they are poorly designed: the reference counter will be changed even on failures, which seems counter-intuitive from the perspective of a developer. We manually collect its refcount APIs, which are shown in Table 5 in the Appendix. The table illustrates three sets of APIs: three refcout increment APIs, six refcount decrement APIs, and four refcount state description APIs. The first two sets of APIs are used in security operation detection. The last set of APIs is used in pre-condition checking of OSPP Rule 4 (we regard these APIs as object-relevant). Since the use of refcount in OpenSSL, FreeBSD, and PHP is limited, we do not identify refcount operations in it.

5.1.3 Detecting Resource Alloc/Release Operations. Improperly used resource allocation/release operations are the main cause of memory leak and could further cause DoS. Some could even directly lead to double-free/use-after-free [1]. A resource is usually complicated and represented as struct or pointer variable in practice. We mainly adopt the resource detection idea of Hector [36] in IPPO. Hector recognizes an allocation as a function call that returns pointer-typed values and a release as its last usage in a path of CFG, which is supposed to be a non-checked call. However, many release operations collected in this way are totally release-irrelevant. To improve the precision, we further require the release functions to contain *free* or *release* in their names. Since Hector is not open source, we implement this part as an LLVM pass independently.

5.1.4 Detecting Lock/unlock. Lock/unlock operations are widely used in large programs to manage shared resources and control concurrency. We observed that lock/unlock operations are usually carried out through function calls with *_lock* and *_unlock* key words in their names. Therefore, we heuristically collect such functions as lock/unlock operations. To further improve the accuracy, we require the unlock functions must be void functions and share the same parameters with the lock functions.

According to our observation, the resource and lock related issues are usually caused by missing resource release and unlock. Therefore, we focus on detecting missed release and unlock operations in OSPPs.

5.2 Path Analysis

5.2.1 Classifying Return Values. Both path analysis and security check detection in IPPO needs to classify function return values with high-precision. Typically, the Linux kernel has three types of return values for non-void functions: boolean value, integer value, and pointer value. Previous works mainly consider integer error return value, which is known as error code. For Linux kernel, returning 0 usually means success and non-zero values otherwise. The FreeBSD kernel and PHP perform similarly as the Linux kernel. However, for

OpenSSL library, a function will return 1 on success. We extend the idea of error code to boolean values and pointer values.

For all of the evaluated systems, a function with pointer type is expected to return a non-null pointer on success and a null on failure, which is easy to catch in LLVM IRs. A function with boolean type is expected to return $true$ on success and $false$ on failure. In LLVM IRs, boolean values and integer values all belong to $ConstantInt$ values. The only difference between them is their bit width: boolean values' bit width is 1 and integer values' bit width is a larger value. One interesting finding here is that the $true$ in LLVM IRs is -1 and $false$ is 0. If we treat boolean and integer return values equally, we will misclassify all boolean return values. There will also be no normal execution path in OpenSSL library under this circumstance.

5.2.2 Constructing RSPs. When we implement Algorithm 1, we carefully select edges to be added to the `ErrEdgeSet` at Line 13 and Line 16. When we select an edge from several candidates to extend the error RVG towards the entry or end block, we prefer the edge that could connect to the existing error RVG. To reduce the recursion depth, we combined the two-phase RSP collection algorithms into one function. The tail-recursions of Algorithm 2 and Algorithm 3 are all reconstructed as loops.

The path analysis phase of IPPO is relatively expensive. In order to speed up the analysis flow, we do not involve each function in the path analysis phase. When a security operation detection is completed, we discard the functions without any security operations before stepping into the path analysis phase. Hence, we could avoid analyzing unnecessary cases.

5.2.3 OSPP Rules Checking. In practice, we firstly collect and pair RSPs, then we execute differential checking before checking the rest OSPP rules to further speed up our analysis. IPPO checks against the rest OSPP rules (see §4.5) in order only when one path of RSPs contains a security operation and the other does not. We check OSPP rules for each kind of security operation independently because the definitions of objects are diverse in different security operations.

We observe that missing check usually occurs in normal RSPs, while missing release and unlock usually occurs in error RSPs. Hence, we only check OSPP rules in normal RSPs for security checks and error RSPs for resource release and unlock operations. Such pattern for refcount operation is not obvious. Thus, we carry out OSPP rules checking in both normal and error RSPs for refcount operation.

5.3 Differential Checking

Given an OSPP, IPPO checks the missed security operation and suggests a potential bug resulted from it. A potential security bug requires one path of OSPP to contain the security operation while the other one does not. Once a missed security operation is found in an OSPP, IPPO generates the detailed inconsistency information for further manual confirmation.

5.3.1 Bug Reports. For each missed security operation bug, IPPO records its top function, file location, and exact bug type (missing check, missing release, and refcount leak). In order to analyze such bugs easily from the perspective of a researcher, IPPO also locates the line number of branch instructions and security operations in the source code, together with the basic block chains that make up the OSPP in LLVM IRs. For the variable related security operations, IPPO locates the sources and security related usages of the critical variables both in source code and in LLVM IRs.

5.3.2 Reports Filter. Since one single path is possible to appear in multiple path pairs, one root cause of a missed security operation may lead to multiple reports. For example, Path 2 (a-c-f) appears in both Path pair 1 and Path pair 2 in **??**. If both Path 1 and Path 3 contain a refcount decrement while Path 2 does not, IPPO will report two potential refcount imbalance bugs for the contradiction is shown in two path pairs. Sometimes one function could also appear in multiple bitcode files, which introduces redundant bug reports, too. To address this problem, IPPO records the belonging function name and the corresponding missed security operation for each bug report. For each function, IPPO records one potential bug for each security operation type.

6 Evaluation

We evaluate the scalability and effectiveness of IPPO using the Linux kernel and OpenSSL library. The experiments were performed on a MacBook Pro laptop with 16GB RAM and an Intel(R) Core(TM) i7 CPU with six cores (i7-8850H, 2.60GHz). We tested the bug detection efficiency on the Linux kernel version 5.8, OpenSSL library version 3.0.0-alpha6, FreeBSD 12 and PHP 8.0.8 using LLVM of version 9.0. For the Linux kernel, we used $allyesconfig$ to compile as many kernel modules as possible, which generated 19,492 LLVM bitcode files. For the OpenSSL library, FreeBSD, and PHP, we used default compile options and generated 2,294, 1,483, and 371 LLVM bitcode files, respectively.

6.1 Overall Analysis Performance

Since the RAM size of our machine is limited, we batch Linux kernel bitcode files before analysis. Each batch contains 3,000 bitcode files. IPPO completed the analysis against the four systems in two hours. It reported 754 bugs in total. The detailed bug detection results are shown in Table 2.

Table 2: Bug detection results of IPPO in the four systems. The R and T in the table indicate the reported bugs and true bugs, respectively.

Bug type	Linux		OpenSSL		FreeBSD		PHP	
	R	T	R	T	R	T	R	T
Missing check	101	11	2	1	1	0	4	0
Missing release	244	68	13	6	1	0	11	1
Refcount leak	345	181	0	0	0	0	0	0
Missing unlock	29	6	0	0	2	1	2	0
Total	719	266	15	7	3	1	17	1

6.2 Bug Findings

We manually checked all the 754 reports generated by IPPO, taking about 20 man-hours in total. We finally confirmed 266, 7, 1, and 1 valid bugs from the Linux kernel, OpenSSL library, the FreeBSD kernel and PHP, respectively, including 181 refcount leaks, 68 memory leaks, 12 missing check bugs, 7 double-free/use-after-free bugs, and 7 missing unlock bugs. Among which, 2 missing check bug, 11 memleak bugs, 99 refcount leak bugs, and 2 missing unlock bugs have been fixed by other developers in the latest systems. We have submitted patches to fix the rest 161 bugs, and 136 of them have been accepted by corresponding maintainers. The detailed list of all bugs is available at Tables 5~11 in the Appendix.

One interesting finding of our bug analysis is that the refcount API `pm_runtime_get_sync()` is commonly misused, which has caused hundreds of bugs. This finding not only shows the limitation of cross-checking, but also reinforces the previous research against the same set of APIs ([26]). These refcount APIs will change the refcount even they return errors, which is counter-intuitive. In practice, it is common for developers to assume that the target task of a function call does not complete on failure. Therefore, many developers do not decrease the PM runtime counter on the failure of `pm_runtime_get_sync()`. Our patchwork aroused a discussion about the design of this set of APIs in the Linux community. Some Linux maintainers suggest that the right thing is to fix the misleading APIs to prevent misuse in the future rather than patch them one by one, which is reasonable. Therefore, we stopped submitting patches resulted in this issue. Fortunately, a new alternative refcount API has been released at the moment: `pm_runtime_resume_and_get()`, which will not modify the reference counter on failure. We believe this API will make future refcount development more stable and secure.

We also investigated the size of all bug functions except those caused by the misunderstanding of APIs (116 functions in total). 56 of them (48.3 %) contain more than 100 lines of source code, and 18 (15.5%) of them contain more than 200 lines. The longest bug function caught by IPPO possesses 613 lines of source code. This testifies IPPO's ability to detect bugs in complicated functions. Actually, 17 of the above long functions introduced bugs earlier than five years ago, and four bugs have existed for more than ten years.

6.3 Comparison with Other Tools

6.3.1 Comparison with Cross-checking Tools. In this subsection, we compare IPPO with three state-of-the-art bug detection tools: APISan [48], Crix [24], FICS [9]. APISan and Crix are based on cross-checking and FICS is based on machine-learning. All of these tools find bugs through differentially checking similar code slices. HERO could detect incorrect error handling through precise function pairing. Our modifications on security check detection have been synchronized in Crix before this experiment. We mainly focus on how many bugs found by IPPO could be caught by cross-checking methods. Thus we use the confirmed bugs found by IPPO as benchmark.

Table 3: Bug detection results of state-of-the-art tools.

Bug type	IPPO	FICS	Crix	APISan
Missing check	12	0	1	0
Missing release	75	0	0	0
Refcount leak	181	0	0	0
Missing unlock	7	0	0	0
Total	275	0	1	0

As shown in Table 3, almost all the bugs found by IPPO cannot be detected by the other three tools. FICS fails on analyzing Linux kernel because of the extremely huge RAM requirements (more than 200GB). Though FICS claims to be able to identify one-to-one inconsistency, its code representation (data dependence graph) and filter strategies are too coarse-grained to analyze path level difference. Crix is designed to detect missing check bugs, thus is incapable of finding other kinds of bugs. For missing check bugs, most bugs found by IPPO lack enough similar code pieces, thus cannot be detected by Crix neither. APISan considers all conditions in a path to construct semantic beliefs. However, as aforementioned in Rule 3 of OSPP, many intermediate conditions and operations have no impact on the usage of security operations, which makes APISan's similarity analysis have poor robustness. The comparison results not only show the limitation of cross-checking methods, but also reveal the effectiveness of IPPO.

6.3.2 Comparison with Pairing Analyses Tool. HERO [10] is the state-of-the-art pairing analysis tool, which could precisely detect functions used in pairs and bugs caused by disordered error handling (missing, redundant, and incorrect order of error handling). The bug types covered by HERO include refcount leak, memory leak, use-after-free/double-free, and incorrect lock/unlock, which are quite similar with the bug types supported by IPPO. Since HERO is not open-sourced yet, we manually checked the bugs found by IPPO and filter bugs that do not exist in the Linux kernel version of 5.3 (on which HERO is evaluated).

Table 4: Comparison with HERO.

Bug types	Bugs in v5.3	HERO Results	Recall
Memory Leak	55	2	3.6%
Refcount Leak	112	82	73.2%
Missing unlock	3	0	0%
UAF/DF	6	0	0%
Total	176	84	47.7%

As shown in Table 4, we finally checked out 176 valid bugs that exist in the Linux kernel of v5.3. HERO successfully detect 84 of them (47.7%). Among these bugs, HERO found most of refcount bugs that caused by misunderstanding of `pm_runtime_get_sync()` API, which is consistent with the conclusion of HERO paper. However, HERO still missed almost half of the bugs found by IPPO due to (1) many custom function pairs are still missed, and (2) HERO could not resolve bugs without leader functions.

6.3.3 Complementarity analysis. We further analyze whether the bugs found by other tools could be found by IPPO. We collected 560 bugs found by APISan, Crix, and HERO in the Linux kernel, and IPPO could detect 119 of them (the bug list of FICS is not released, thus is ignored). The above experiments show that IPPO shares very limited intersection with existing bug detection tools, which means IPPO is a promising complementation with them.

6.4 False Positives

The overall false positive rate of IPPO is 63.5%. The main causes are summarized below.

- **Unexpected pre-condition.** Though IPPO has considered this case, as described in §4.2, it cannot pick out all eligible cases. Developers often use some temporary variables to adjust the timing of using security operations, especially refcount operations.

These temporary variables may cause a missing case in a small part of code or a function, but finally they will balance again in the global context. Sometimes developers set call-back functions to auto-manage resources, which is hard to detect. IPPO needs a more powerful inter-procedure analysis flow to model and check against pre-conditions. Such cases account for 27% of the false positives.

- **Imprecise data-flow analysis.** The value escape methods while checking Rule 4 are diverse in practice. Some values escape through function calls, which needs expensive inter-procedure alias analysis and IPPO cannot tell which functions are designed for this. Complex value propagation also decreases the analysis precision. These cases lead to about 30% of the false positives.
- **Imperfect error path analysis.** Our error path analysis highly relies on return values, which is unreliable while analyzing void functions. Many error handling paths in void functions also have none error handling functions (*e.g.*, print error messages), thus are indistinguishable for IPPO by now. This contributes 14% of the false positives.
- **Imperfect security operation detection.** Currently the implementation of security operation detection cannot handle complex scenes. For example, some resources are released through refcount operations or other wrapper functions, which is missed by IPPO. This reason accounts for 6% of the false positives.
- **Other causes.** Other cases like special function logic could also cause false positives. Some missed security operations have no obvious security impact, thus are not counted as bugs. These cases contribute the rest 23% false positives.

False positive is always a key challenge in program static analysis, especially for complex analysis targets like OS kernels. We believe that the 63.5% false positive rate of IPPO is acceptable. In addition, the three state-of-the-art similar static analysis tools also suffer from this issue (65.4%, 99.8%, and 88.0% false positive rates for Crix [24], APISan [48], and FICS [9], respectively). On the other hand, manually analyzing bugs suggested by IPPO is easy because the correct path provides references. According to our statistics, it takes a non-expert researcher less than two minutes on average to check a bug report after analyzing several examples.

6.5 False Negatives

To evaluate the false negatives of IPPO, we constructed a testset by manually removing the security operations in normal functions. We randomly selected 40 functions in Linux kernel where security operations are shown in multiple paths. Then, we delete 10 resource release calls, 10 return value checks, 10 refcount decrements, and 10 unlocks from them, which results in 10 memleak bugs, 10 missing check bugs, and 10 refcount leak bugs, and 10 missing unlock bugs, respectively. We used IPPO to check against this benchmark and IPPO successfully detected 31 missed security operations (77.5% recall rate). Among the false negatives, two memleak bugs, three missing check bugs, and four missing unlock bugs are missed by IPPO. Two of them are filtered for the pre-condition containing function arguments, which breaks Rule 4 of OSPP. One security check in a function is not identified, which leads to a false negative. One checked function is defined inline, and removing security checks makes the function call instruction disappear in LLVM IRs. Four unlock calls does not share the same parameters with the lock calls, thus are filtered. The last missed memleak is caused by complex variable definition. Specifically, the resource variable has multiple definitions and the path where the resource release is removed has a wrapper function of release. However, this release wrapper should be paired with a previous definition, which is beyond the capability of IPPO.

6.6 Security Impacts of the Found Bugs

In this section, we discuss the security impacts of the bugs identified by IPPO. To this end, we first evaluate the potential reachability of these bugs. Furthermore, we illustrate the potential impacts of them.

6.6.1 Reachability Analysis. Understanding the reachability of bugs in complex programs is an open problem. Thus, in this evaluation, we are using the existence of shorter call-stacks, which are from system entry points to the vulnerable functions, to measure the reachability of bugs. Similar to previous works such as PeriScope [37] and SID [44], we chose the system calls, `ioctl` handlers, and IRQ handlers as user-controllable system entry-points to evaluate the reachability of the bugs identified by IPPO. Our evaluation result shows that 71.9% of the bugs identified by IPPO are reachable from at least one of these entry points, which means that these bugs are possible to be triggered by users.

6.6.2 Impact Analysis. As we discussed in §1, most of the bugs caused by missing security operations would cause security impacts such as memory corruption, DoS, and memory leak, when triggered. Considering the reachability evaluation, 71.9% of the bugs identified by IPPO would lead to at least one security impact. Specifically, 24.6% and 70% of the bugs would cause memory leaks and refcount leaks, respectively, which would further lead to deny-of-service if they are triggered repeatedly. Also, 5% of the bugs identified by IPPO would cause null-pointer-dereference, which may lead to memory corruption when triggered. For example, Figure 5 shows a potential use-after-free/double-free bug in the Linux kernel identified by IPPO. Function `snd_echo_resume()` in Figure 5 could be reached from the system call `sys_ioctl()`, which means that attackers could trigger it and cause security impacts to the kernel.

6.7 Scalability and Portability

The bug detection results on both Linux kernel and OpenSSL library have shown the scalability and portability of IPPO. The idea of OSPP and missed security operations are shared by various programs regardless of whether they run in kernel mode or user mode. Different programs may have their own preference in using security operations, but this is a pluggable analysis pass in IPPO and it supports further expansion on security operation types. IPPO is able to detect various missed security operations and infer their security impacts based on similar path pairs analysis in various programs that could be compiled into LLVM IRs.

7 Discussion

Security operation detection. IPPO detects security operations with a direct and simple analysis pass, as mentioned in §5.1. Since detecting security operations is not the main goal of IPPO, we only choose to implement three kinds of security operations in this paper to estimate the bug detection of IPPO. However, this part is pluggable and supports further extension. Here, three common types of security operations are sufficient to demonstrate the effectiveness of our approach. In the future, on the one hand, we plan to add more security operations (*e.g.*, variable initialization) to further improve the bug detection ability of IPPO. On the other hand, we will open-source IPPO and enable interested readers to extend IPPO and detect bugs according to their practical needs.

Inter-procedural analysis. IPPO is mainly designed based on static intra-procedural analysis. However, only considering the information within a single function could result in both false positives and false negatives. Meanwhile, Some missed security operations (*e.g.*, security checks) are more likely to show the difference in inter-procedural context. We believe the inter-procedural feature could be implemented by considering function calls, which is an interesting future work.

Precise data-flow analysis. Currently, we track all variables' sources and use flows directly through a data-flow analysis implemented by us, which may not be accurate enough and may cause false positives. To address this problem, it is interesting to introduce professional pointer analysis technology (*e.g.*, Andersen pointer analysis [15] or batch analysis [40]) to improve our analysis flows and the overall accuracy.

Exploitability analysis. We have analyzed the reachability of bugs found by IPPO in §6.6. To obtain the complete exploitability of a bug, we also need to analyze the accurate trigger condition, which could be accomplished through symbolic execution [35]. Actually, automatically determining the exploitability of a bug is a hot topic get with challenges, which deserves independent research (*e.g.*, AEG [11], EvilCoder [32], MAZE [43], and Coppelia [38]). In this paper, we mainly focus on detecting the existence of a potential security bug. We may adopt exploitability analysis in the future to reduce the false-positive rate of IPPO.

8 Related Work

Differential analysis in bug detection. Differential analysis against similar code snippets is a common practice to detect semantic bugs. FICS [9] uses machine learning to measure the similarity and difference among code pieces. Similarly, some research also uses machine learning to detect bugs in smart contracts [23, 51]. PISan [49] automatically infers the correct API usage-patterns and further detects API misuse bugs through cross-checking. Juxta [27] could infer high-level semantics from source code and pick out the implementations that are inconsistent with the implicit semantics. CRADLE [33] leverages inconsistency checking to detect bugs in deep learning libraries. Hector [36] and RID [26] detect inconsistent release and refcount operations through intra-procedural path analysis, while Pex [50] and Crix [24] identify missing checks in inter-procedural paths or slices. However, most of the previous static-analysis works would rely on cross-checking to eliminate false positives, which would further introduce false negatives, as we discussed in §1. Furthermore, some of these works, such as Crix[24], can only identify a specific type of bugs like missing check bugs. Unlike these existing works, IPPO is more generic and can identify multiple types of bugs without using the cross-checking technique.

Similarity analysis in bug detection. Besides analyzing inconsistent cases in the similar code pieces, one could also detect bugs by identifying the similarity between the target program and the existing bug samples (*e.g.*, bug reports), which is usually referred to as *bug localization* or *vulnerability extrapolation*. VulPecker [21] defines a set of patch features and detects bug source code through similarity analysis [22]. MVP [45] builds patch and function signatures at syntactic and semantic levels to locate bugs. DNNLoc [20] utilizes deep neural networks to relate the terms in bug reports to source files, while DrewBL [39] uses word2vec methods to localize faulty files. Pewny et al. [31] presented a method to find binary-level code parts that share similar I/O behavior with the bug code. These approaches are able to detect multiple kinds of security bugs but require existing bug knowledge and have limitations on detecting bugs that have not appeared before.

Detecting bugs in OS kernels. OS kernel is a typical large scale and widely used set of programs, which has been a hot research target in the domain of security. Defining and detecting bugs in OS kernels is of great significance while being challenging. Pallas [16] presented fast-path bugs and detected them in Linux kernel and Android kernel. LRSan [42] and Crix [24] detect bugs around security checks, while Deadline [46], Dftinker [25], and DFTracker [41] detect double-fetch bugs in OS kernels. DCNS [12] could identify conservative non-sleep defects in Linux kernel. These approaches are effective for their target bugs, but usually rely on precisely defined rules to detect the specific bugs, which needs a wealth of analysis experience. Recently, fuzzing is also applied to bug detection tasks gradually in OS kernels [13, 18, 19, 29]. However, the path explosion problem has not been resolved yet, which leads to a relatively low code coverage rate.

9 Conclusion

Missing security operations is common in large scale programs and could lead to various security issues. In this paper, we present IPPO, a security bug detection framework to automatically detect bugs caused by missed security operations. IPPO models the object-based similar-path pairs within a function and detects potential security bugs through differential checking, which makes the analysis fine-grained. We realize the efficient path analysis of IPPO with several new techniques, termed return value–based sub-CFG (RVG) and reduced similar-path (RSP), respectively. We evaluate IPPO on Linux kernel, OpenSSL library, FreeBSD kernel, and PHP. in which IPPO discovered 161 new bugs. We have submitted patches for these bugs and most of them have been fixed with our patches. The evaluation results show the effectiveness and portability of IPPO on bug detection.

10 Acknowledgment

This work was partly supported by NSFC under No. U1936215 and U1836202, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, and the Fundamental Research Funds for the Central Universities (Zhejiang University NG-ICS Platform). Qiushi Wu and Kangjie Lu were supported in part by the NSF awards CNS-1815621 and CNS-1931208.

References

[1] 2020. CVE-2019-12819, a use-after-free vulnerability in Linux kernel. https://www.cvedetails.com/cve/CVE-2019-12819/
[2] 2020. CVE-2019-15807, a memory leak vulnerability in Linux kernel. https://cwe.mitre.org/data/definitions/833.html
[3] 2020. CVE-2019-8980, a memory leak vulnerability in Linux kernel. https://www.cvedetails.com/cve/CVE-2019-8980/
[4] 2020. CVE Details. The ultimate security vulnerability datasource. https://www.cvedetails.com/
[5] 2020. CWE-285: Improper Authorization. https://cwe.mitre.org/data/definitions/285.html
[6] 2020. CWE-788: Access of Memory Location After End of Buffer. https://cwe.mitre.org/data/definitions/788.html
[7] 2020. CWE-833: Deadlock. https://cwe.mitre.org/data/definitions/833.html
[8] 2020. CWE-920: Improper Restriction of Power Consumption. https://cwe.mitre.org/data/definitions/920.html
[9] 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi
[10] 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi
[11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. (2011).
[12] Jia-Ju Bai, Julia Lawall, Wende Tan, and Shi-Min Hu. 2019. DCNS: automated detection of conservative non-sleep defects in the linux kernel. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 287–299.
[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.
[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.
[15] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–299.
[16] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 709–722.
[17] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. "Semantic-Aware Checking for Finding Deep Bugs in Fast Path". In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Xi'an, China.
[18] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768.
[19] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society.
[20] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 218–229.
[21] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 201–213.
[22] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. 2021. Multilevel Graph Matching Networks for Deep Graph Similarity Learning. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* (2021).
[23] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2021), 1–14. https://doi.org/10.1109/TKDE.2021.3095196
[24] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.
[25] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. 2018. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 780–785.
[26] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 531–544.
[27] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 361–377.
[28] KOSAKI Motohiro. 2020. A memory corruption by refcount imbalance. https://lore.kernel.org/patchwork/patch/331920/
[29] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 729–743.
[30] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1203–1218. https://doi.org/10.1145/3372297.3417256
[31] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy*. 709–724.
[32] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 214–225.
[33] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038.
[34] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 621–632.
[35] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 49–64.
[36] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.
[37] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary.. In *NDSS*.
[38] Cynthia Sturton. 2019. Hardware Is the New Software: Finding Exploitable Bugs in Hardware Designs. USENIX Association, Burlingame, CA.
[39] Y. Uneno, O. Mizuno, and E. Choi. 2016. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 183–190.
[40] Jyothi Vedurada and V Krishna Nandivada. 2019. Batch alias analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 936–948.
[41] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. 2019. DFTracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science* 13, 2 (2019), 247–263.
[42] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.
[43] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. 2021. MAZE: Towards Automated Heap Feng Shui. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-yan
[44] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*.
[45] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1165–1182. https://www.usenix.org/conference/usenixsecurity20/presentation/xiao
[46] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. 661–678.
[47] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Information and Software Technology* 82 (2017), 177–192.
[48] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378.

[49] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing {API} Usages through Semantic Cross-Checking. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 363–378.
[50] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel.. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1205–1220.
[51] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In *IJCAI*. 3283–3290.

A Appendix

Effectiveness in addressing path explosion. Figure 7 shows a potential path collection result on a target CFG after executing our two-phase RSP collection algorithm. The control flow structure in the dashed box of Figure 7 is very common in the real world, which is introduced by an `if` statement. This simple structure could double the total path number if we collect paths from the entry block (block a) to the end block (block l) directly. To make matters worse, the information of this structure (path b-d or b-c-d) is contained in every collected path, which brings huge redundancy. As a result, we will collect 10 paths with an average length of 7.1 blocks in this way. In order to check every path and make sure each path could be compared at least once with all other paths directly or indirectly, we have to construct at least 9 path pairs in this case.

However, the aforementioned control flow structure will introduce only one path pair (G1) when using RSPs to represent CFG paths, which makes the collection result increase linearly rather than exponentially. Additionally, The information of this structure does not appear in any other collected path group. Our path collection method finally collects 5 path pairs from this CFG with an average path length of only 3.1 blocks, which significantly reduces the workload of path comparison.

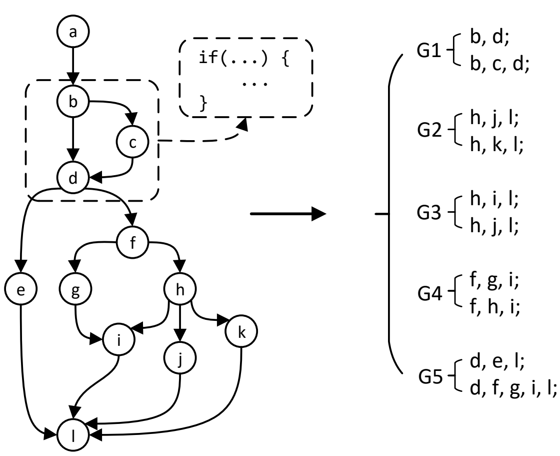

Figure 7: An example of RSP collection. IPPO finally collects five path groups (namely G1-G5) in the left CFG that satisfy Rule 1 of OSPP.

Table 5: Reference count APIs.

Function description	API
PM usage counter increment	pm_runtime_get pm_runtime_get_sync pm_runtime_get_noresume
PM usage counter decrement	pm_runtime_put pm_runtime_put_sync pm_runtime_put_noidle pm_runtime_put_autosuspend pm_runtime_put_sync_suspend pm_runtime_put_sync_autosuspend
PM usage counter state	pm_runtime_get_if_in_use pm_runtime_active pm_runtime_enabled pm_runtime_suspended

Table 6: List of bugs detected by IPPO in OpenSSL, FreeBSD, and PHP. The S, A, and F in the Status column indicate submitted, accepted, and fixed by other developers in the last version, respectively.

System	Buggy function	Impact	Status
OpenSSL	dtls1_buffer_message	Memleak	A
OpenSSL	newpass_bag	Memleak	A
OpenSSL	PKCS5_PBE_keyivgen	Memleak	A
OpenSSL	generate_cookie_callback	Memleak	A
OpenSSL	build_chain	Memleak	A
OpenSSL	cms_RecipientInfo_pwri_crypt	Memleak	F
OpenSSL	int_ctx_new	Reliability	F
FreeBSD	wpi_run	Deadlock	C
PHP	accel_preload	Memleak	S

Table 7: List of bugs (1-16) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
snd_intel8x0m_create	Null-pointer-dereference	C
dpot_read_spi	Reliability	A
sdhci_pci_o2_probe	Reliability	S
smtcfb_pci_probe	Null-pointer-dereference	F
i40e_vsi_open	Reliability	A
e1000_set_d0_lplu_state_82571	Reliability	A
ahc_handle_seqint	Null-pointer-dereference	S
ahd_handle_seqint	Null-pointer-dereference	S
sata_dwc_isr	Null-pointer-dereference	C
mpu3050_trigger_handler	Reliability	A
vadc_do_conversion	Reliability	C
snd_echo_resume	Double-free	A
qcom_snd_parse_of	Refcount leak	A
ide_pci_init_two	Memleak	S
rxe_mem_init_user	Memleak	A
subscribe_event_xa_alloc	Memleak	F

Table 8: List of bugs (17-73) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
hl_device_reset	Memleak	A
vmd_enable_domain	Memleak	F
fb_probe	Memleak	F
radeonfb_pci_register	Memleak	F
wilc_sdio_probe	Memleak	A
wilc_bus_probe	Memleak	A
rtl8192_usb_initendpoints	Null-pointer-dereference	A
fwserial_create	Memleak	A
ia_css_stream_create	Memleak	A
crc_control_write	Memleak	A
nv50_wndw_new_	Memleak	S
amdgpu_debugfs_gpr_read	Memleak	F
amdgpu_dm_mode_config_init	Memleak	A
vega20_setup_od8_information	Memleak	F
v3d_submit_cl_ioctl	Double-free	F
ethoc_probe	Memleak	A
ice_set_ringparam	Memleak	C
ixgbe_configure_clsu32	Memleak	A
gemini_ethernet_port_probe	Double-free	F
mvneta_probe	Memleak	A
bcm_sysport_probe	Memleak	A
mlx5e_create_inner_ttc_table_groups	Double-free	A
mlx5e_create_ttc_table_groups	Double-free	A
mlx5e_create_l2_table_groups	Memleak	A
hns_nic_dev_probe	Memleak	A
arc_mdio_probe	Memleak	A
_rtl_usb_receive	Memleak	F
prism2_config	Memleak	S
ttc_setup_clockevent	Memleak	S
fs_open	Memleak	A
scsi_debug_init	Memleak	A
pm8001_exec_internal_task_abort	Memleak	A
vnic_dev_init_devcmd2	Memleak	A
olpc_ec_probe	Memleak	A
ca91cx42_dma_list_add	Memleak	S
intel_ntb_pci_probe	Memleak	A
watchdog_cdev_register	Use-after-free	A
watchdog_cdev_register	Memleak	A
intel_irq_remapping_alloc	Memleak	A
st95hf_in_send_cmd	Memleak	A
qca_controller_memdump	Memleak	A
btusb_mtk_submit_wmt_recv_urb	Memleak	A
sun6i_rtc_clk_init	Memleak	A
adis_probe_trigger	Memleak	F
i5100_init_one	Memleak	A
extcon_dev_register	Memleak	A
empress_init	Memleak	A
dvb_register_device	Memleak	A
emmaprp_probe	Memleak	A
isp_probe	Memleak	A
em28xx_alloc_urbs	Use-after-free	A
tm6000_start_stream	Memleak	A
add_extent_data_ref	Memleak	C
dbAdjCtl	Memleak	A
ubifs_init_authenticatio	Memleak	A
add_new_gdb	Memleak	A
add_partition	Refcount leak	C

Table 9: List of bugs (74-130) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
init_desc	Memleak	A
nf_nat_init	Memleak	A
rxkad_verify_response	Memleak	A
krb5_make_rc4_seq_num	Memleak	F
wm8962_irq	Refcount leak	A
wm8962_set_fll	Refcount leak	A
tas2552_probe	Refcount leak	C
tas2552_component_probe	Refcount leak	A
img_spdif_in_probe	Refcount leak	A
img_i2s_out_probe	Refcount leak	A
img_spdif_out_probe	Refcount leak	A
img_i2s_in_probe	Refcount leak	F
omap2_mcbsp_set_clks_src	Refcount leak	A
bq24190_sysfs_show	Refcount leak	F
bq24190_sysfs_store	Refcount leak	A
bq24190_charger_get_property	Refcount leak	F
bq24190_charger_set_property	Refcount leak	F
bq24190_battery_get_property	Refcount leak	F
bq24190_battery_set_property	Refcount leak	F
sun8i_ce_probe	Refcount leak	S
sun8i_ce_cipher_init	Refcount leak	C
sun8i_ss_cipher_init	Refcount leak	S
sun8i_ss_probe	Refcount leak	A
rcar_pcie_probe	Refcount leak	A
rcar_pcie_ep_probe	Refcount leak	A
dra7xx_pcie_probe	Refcount leak	C
pex_ep_event_pex_rst_deassert	Refcount leak	C
qcom_pcie_probe	Refcount leak	A
cdns_plat_pcie_probe	Refcount leak	A
mipi_csis_s_stream	Refcount leak	C
atomisp_open	Refcount leak	S
atomisp_pci_probe	Refcount leak	A
tegra_vde_ioctl_decode_h264	Refcount leak	A
cedrus_start_streaming	Refcount leak	F
rkisp1_vb2_start_streaming	Refcount leak	F
etnaviv_gpu_init	Refcount leak	F
etnaviv_gpu_recover_hang	Refcount leak	F
etnaviv_gpu_bind	Refcount leak	F
cdns_dsi_transfer	Refcount leak	F
nouveau_drm_ioctl	Refcount leak	F
nouveau_drm_open	Refcount leak	F
nouveau_debugfs_strap_peek	Refcount leak	F
nouveau_connector_detect	Refcount leak	F
nouveau_gem_object_del	Refcount leak	F
amdgpu_driver_open_kms	Refcount leak	F
amdgpu_hwmon_get_pwm1	Refcount leak	F
amdgpu_hwmon_set_pwm1	Refcount leak	F
amdgpu_hwmon_get_pwm1_enable	Refcount leak	F
amdgpu_hwmon_set_pwm1_enable	Refcount leak	F
amdgpu_hwmon_get_fan1_input	Refcount leak	F
amdgpu_hwmon_get_fan1_min	Refcount leak	F
amdgpu_hwmon_get_fan1_max	Refcount leak	F
amdgpu_hwmon_get_fan1_target	Refcount leak	F
amdgpu_hwmon_set_fan1_target	Refcount leak	F
amdgpu_hwmon_get_fan1_enable	Refcount leak	F
amdgpu_hwmon_set_fan1_enable	Refcount leak	F
amdgpu_hwmon_show_power_avg	Refcount leak	F

Table 10: List of bugs (131-186) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
amdgpu_hwmon_set_power_cap	Refcount leak	F
amdgpu_hwmon_show_vddgfx	Refcount leak	F
amdgpu_hwmon_show_vddnb	Refcount leak	F
amdgpu_hwmon_show_mclk	Refcount leak	F
amdgpu_hwmon_show_temp	Refcount leak	F
amdgpu_hwmon_show_sclk	Refcount leak	F
amdgpu_set_dpm_state	Refcount leak	F
amdgpu_set_dpm_forced_performance_level	Refcount leak	F
amdgpu_set_pp_force_state	Refcount leak	F
amdgpu_get_pp_table	Refcount leak	F
amdgpu_set_pp_table	Refcount leak	F
amdgpu_set_pp_sclk_od	Refcount leak	F
amdgpu_set_pp_mclk_od	Refcount leak	F
amdgpu_set_pp_power_profile_mode	Refcount leak	F
amdgpu_set_pp_od_clk_voltage	Refcount leak	F
amdgpu_get_gpu_busy_percent	Refcount leak	F
amdgpu_get_mem_busy_percent	Refcount leak	F
amdgpu_set_pp_features	Refcount leak	F
amdgpu_debugfs_process_reg_op	Refcount leak	F
amdgpu_debugfs_regs_didt_read	Refcount leak	F
amdgpu_debugfs_regs_pcie_read	Refcount leak	F
amdgpu_debugfs_regs_smc_read	Refcount leak	F
amdgpu_debugfs_sensor_read	Refcount leak	F
amdgpu_debugfs_wave_read	Refcount leak	F
amdgpu_debugfs_gpr_read	Refcount leak	F
amdgpu_debugfs_sclk_set	Refcount leak	F
amdgpu_debugfs_gpu_recover	Refcount leak	F
amdgpu_connector_dp_detect	Refcount leak	F
amdgpu_connector_vga_detect	Refcount leak	F
amdgpu_connector_dvi_detect	Refcount leak	F
amdgpu_connector_lvds_detect	Refcount leak	F
kfd_bind_process_to_device	Refcount leak	F
panfrost_job_hw_submit	Refcount leak	A
v3d_job_init	Refcount leak	S
radeon_driver_open_kms	Refcount leak	F
radeon_dp_detect	Refcount leak	F
radeon_vga_detect	Refcount leak	F
radeon_dvi_detect	Refcount leak	F
radeon_tv_detect	Refcount leak	F
radeon_lvds_detect	Refcount leak	F
mock_gem_device	Refcount leak	S
flexcan_probe	Refcount leak	F
flexcan_open	Refcount leak	F
xcan_probe	Refcount leak	S
xcan_open	Refcount leak	F
fec_enet_open	Refcount leak	F
fec_enet_get_regs	Refcount leak	F
fec_enet_mdio_read	Refcount leak	F
fec_enet_mdio_write	Refcount leak	F
smsc911x_drv_probe	Refcount leak	A
wlcore_regdomain_config	Refcount leak	A
wl1271_recovery_work	Refcount leak	F
wl1271_op_add_interface	Refcount leak	F
wl1271_op_bss_info_changed	Refcount leak	F
exynos_trng_probe	Refcount leak	C
mtk_smi_larb_resume	Refcount leak	F

Table 11: List of bugs (187-243) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
rproc_fw_boot	Refcount leak	F
cyapa_update_rt_suspend_scanrate	Refcount leak	S
omap4_keypad_probe	Refcount leak	S
ina3221_write_enable	Refcount leak	F
ti_j721e_ufs_probe	Refcount leak	C
omap_i2c_probe	Refcount leak	C
sprd_i2c_master_xfer	Refcount leak	S
lpi2c_imx_master_enable	Refcount leak	C
img_i2c_init	Refcount leak	F
stm32f7_i2c_smbus_xfer	Refcount leak	F
stm32f7_i2c_reg_slave	Refcount leak	C
stm32f7_i2c_unreg_slave	Refcount leak	C
xiic_xfer	Refcount leak	F
qcom_slim_ngd_enable	Refcount leak	S
apple_mfi_fc_set_property	Refcount leak	F
xhci_histb_probe	Refcount leak	S
__cdns3_gadget_init	Refcount leak	C
fsl_lpspi_probe	Refcount leak	C
ti_qspi_setup	Refcount leak	A
zynqmp_qspi_probe	Refcount leak	A
rti_wdt_probe	Refcount leak	F
stm32_mdma_alloc_chan_resources	Refcount leak	F
tegra_dma_issue_pending	Refcount leak	A
tegra_adma_alloc_chan_resources	Refcount leak	A
stm32_dmamux_probe	Refcount leak	S
stm32_dmamux_route_allocate	Refcount leak	S
stm32_dmamux_resume	Refcount leak	S
sprd_dma_probe	Refcount leak	C
dw_probe	Refcount leak	A
rcar_dmac_probe	Refcount leak	S
usb_dmac_probe	Refcount leak	S
edma_probe	Refcount leak	C
am654_hbmc_probe	Refcount leak	F
gpmi_init	Refcount leak	F
gpmi_nfc_exec_op	Refcount leak	F
cqspi_probe	Refcount leak	C
sata_rcar_probe	Refcount leak	C
gp2ap002_probe	Refcount leak	C
arizona_gpio_direction_out	Refcount leak	F
rcar_usb2_clock_sel_probe	Refcount leak	S
arizona_clk32k_enable	Refcount leak	F
arizona_irq_thread	Refcount leak	C
venus_probe	Refcount leak	A
venc_open	Refcount leak	A
ispif_set_power	Refcount leak	A
csid_set_power	Refcount leak	A
vfe_get	Refcount leak	A
csiphy_set_power	Refcount leak	F
s3c_camif_open	Refcount leak	S
s3c_camif_probe	Refcount leak	A
dcmi_start_streaming	Refcount leak	F
fimc_is_probe	Refcount leak	A
fimc_isp_subdev_s_power	Refcount leak	F
fimc_md_register_sensor_entities	Refcount leak	F
fimc_lite_open	Refcount leak	F
isp_video_open	Refcount leak	A
fimc_capture_open	Refcount leak	A

Table 12: List of bugs (244-266) detected by IPPO in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Impact	Status
deinterlace_start_streaming	Refcount leak	A
cal_probe	Refcount leak	S
rvin_open	Refcount leak	F
s5p_mfc_power_on	Refcount leak	F
vpif_probe	Refcount leak	A
vsp1_probe	Refcount leak	A
bdisp_probe	Refcount leak	A
regs_show	Refcount leak	A
hva_hw_probe	Refcount leak	A
hva_hw_dump_regs	Refcount leak	F
coda_probe	Refcount leak	A
coda_open	Refcount leak	A
__s5k6a3_power_on	Refcount leak	A
smiapp_probe	Refcount leak	A
smiapp_pm_get_init	Refcount leak	F
pvrdma_register_device	Memleak	A
find_free_vf_and_create_qp_grp	Memleak	A
bnxt_re_dev_init	Deadlock	A
virtio_gpu_execbuffer_ioctl	Deadlock	F
qlcnic_pinit_from_rom	Deadlock	C
qlcnic_83xx_flash_read32	Deadlock	A
raid_ctr	Deadlock	C
idxd_config_bus_probe	Deadlock	F