Understanding and Detecting Disordered Error Handling with Precise Function Pairing

Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu
University of Minnesota

Abstract

Software programs may frequently encounter various errors such as allocation failures. Error handling aims to gracefully deal with the errors to avoid security and reliability issues, thus it is prevalent and vital. However, because of its complexity and corner cases, error handling itself is often erroneous, and prior research has primarily focused on finding bugs in the *handling* part, such as incorrect error-code returning or missing error propagation.

In this paper, we propose and investigate a class of bugs in error-handling code from a different perspective. In particular, we find that programs often perform "cleanup" operations before the actual error handling, such as freeing memory or decreasing refcount. Critical bugs occur when these operations are performed (1) in an incorrect order, (2) redundantly, or (3) inadequately. We refer to such bugs as *Disordered Error Handling* (DiEH). Our investigation reveals that DiEH bugs are not only common but can also cause security problems such as privilege escalation, memory corruption, and denial-of-service. Based on the findings from the investigation, we then develop a system, HERO (Handling ERrors Orderly), to automatically detect DiEH. The core of HERO is a novel technique that precisely pairs both common and custom functions based on the unique error-handling structures, which allows us to infer expected cleanup functions. With HERO, we found 239 DiEH bugs in the Linux kernel, the FreeBSD kernel, and OpenSSL, which can cause security and reliability issues. The evaluation results show that DiEH is critical and widely exists in system software, and HERO is effective in detecting DiEH. We also believe that the precise function pairing is of independent interest in other research areas such as temporal-rule inference and race detection.

1 Introduction

A program may encounter various errors at runtime, including hardware errors (e.g., disk corruption), software errors (e.g., an unlock without a lock), and invalid inputs. To avoid crashes and insecure operations, the error-handling mechanisms capture and gracefully deal with errors. As such, error handling plays a key role in ensuring the security and reliability of programs. Also, error-handling code is very prevalent; for example, according to our study, there are more than 400K occurrences of error handling in about 18K source files in the Linux kernel.

Unfortunately, error-handling code itself is often erroneous. In particular, EIO [18] even shows that error-handling code is "occasionally" correct. After checking the latest 100 CVE-assigned vulnerabilities [41] in the Linux kernel, we also found that at least 34% of them are related to incorrect error handling. Critically, erroneous error handling may result in many security issues such as use-after-free [11], information leakage [10], and denial-of-service [9].

The error-prone nature of error-handling code stems from several reasons. First, the error-handling code often deals with corner cases that are less likely to occur during normal execution. This results in two problems: Bugs in the error-handling code are often not triggered or noticed, and developers tend to overlook such rare cases. We argue that, in adversarial scenarios, attackers can intentionally trigger error-handling code through techniques like memory exhaustion [60] and fault injection [44]. Thus the bugs can be equally critical to the ones in normal code. Second, traditional dynamic testing, such as fuzzing, cannot adequately cover the majority of error-handling code because errors are hard to trigger in fuzzing. Third, error handling often involves special and complicated logic, which poses significant challenges to correct implementation.

Developers in low-level languages mainly use four error-handling primitives. (1) Terminating execution. When an error is critical, the error handling terminates the execution to avoid attacks or data/file corruption. (2) Printing error messages. The code prints out the details about the error for users to investigate further. In this case, the error is less critical, so that the execution can continue. (3) Passing error upstream. The function encountering the error passes the error back to the callers and expects the callers to handle it further. (4) Fixing errors. When the error is fixable, the error handling can directly fix it (e.g., resetting the size value) and continue.

Prior research thus has primarily focused on detecting bugs only in the "handling." For example, Rubio-González et al. [45] and EIO [18] proposed static-analysis approaches to detect error-propagation bugs in file systems, i.e., if error codes are passed correctly. APEx [20], ErrDoc [53], and EPEx [19] also check if errors are identified and handled in the callers. Although a few previous works attempted to check the operations before the handling, they are limited to only missing-operation cases. For example, Hector [48] detects missing memory release, and RID [28] detects missing refcount decrease. To the best of our knowledge, none of them could detect other bugs associated with the operations in error paths before handling, such as cases in which the operations are present but in an incorrect order or redundant. To fill this blank area, in this work, we aim to systematically study and detect the bugs of problematic operations in error paths.

1.1 Contributions

In this paper, we first propose a class of error-handling bugs from a different perspective and then develop an effective detection system with multiple new techniques.

Proposing DiEH bugs. While prior research primarily focused on the "handling" part, we find that, in the error paths, programs often perform "cleanup" operations before actually handling the errors. For the example shown in Figure 1, when the function `video_register_device()` (line 13) encounters an error, the code releases the pointer `vfd` (line 24) and unregisters the device (line 26) before passing the error to its caller. As the cleanup operations, these functions must be called correctly; otherwise, the program is vulnerable to bugs such as use-after-free. Buggy cases include calling cleanup functions (1) in an incorrect order, (2) redundantly, and (3) inadequately. We refer to such bugs as *Disordered Error Handling* (DiEH). While prior research studied inadequate error handling such as missing memory release [48] and missing refcount release [28], redundant and incorrect-order error-handling problems are unexplored.

An in-depth study of DiEH. Although the impacts and results of DiEH are known types of bugs like double-free and memory leak, it represents the root causes of a wide class of error-handling bugs from a different perspective, so we first define it and conduct an in-depth study of DiEH in multiple aspects: causes, commonness, categories, and criticalness. Specifically, DiEH is hard to avoid because (1) cleanup functions are often custom and are hard to use correctly, and (2) the error-handling code can be highly complex and involve corner cases. As a result, DiEH bugs are a common occurrence in complex programs like OS kernels. DiEH bugs can cause multiple types of security impacts, such as privilege escalation, memory corruption, information leakage, and denial-of-service, as will be detailed in §2.2.5.

Precise function pairing analysis. Our study also shows that the key to detecting DiEH is to precisely determine which cleanup functions should be called to handle the corresponding functions in the normal paths, i.e., to identify function pairs. However, function pairing is a challenging problem because such functions are abundant, diverse, and highly customizable. Moreover, the pairing rules are typically undocumented, so pairing is hard for even manual analysis. To address this problem, we propose a new technique so-called *delta-based pairing* (see §4) that precisely identifies both common and custom functions that should appear pairwise by exploiting unique error-handling structures. We believe our pairing analysis is of independent interest in other research areas such as temporal-rule inference and race detection.

An effective detection system—HERO. Based on our empirical study of DiEH bugs, we identify three challenges in their detection. First, DiEH represents the root causes of a wide class of semantic bugs in error-handling code from a different perspective, so the detecting rules are undefined yet. Second, a DiEH case may not be harmful, so we need to distinguish and remove harmless cases. Third, by nature, code paths containing DiEH bugs often involve path conditions (e.g., return-value checks), so path-feasibility testing is required to ensure that the paths are valid. To address these problems, we model DiEH and propose HERO ((Handling ERrors Orderly)). HERO is equipped with multiple techniques such as *scalable symbolic summaries* for eliminating infeasible paths and *dependency reasoning* for removing harmless incorrect-order DiEH cases. HERO also provides rankings to facilitate the final manual confirmation for DiEH bugs.

Open-source implementation and new bugs. We implemented HERO on top of LLVM-10 and plan to open-source it. HERO is scalable and effective. By applying it to the Linux kernel, the FreeBSD kernel, and the OpenSSL library, we found 239 new DiEH bugs, most of which can cause critical security issues to billions of devices running these applications. We reported these bugs and fixed most of them by working with the maintainers. The results confirm that DiEH bugs are indeed common and security-critical.

2 Background and Study

In this section, we discuss the unique structures of error handling and present our study of DiEH.

2.1 Error handling and function pairs

In case of an error, functions usually first clean up or handle the previous operations, e.g., releasing memory, before actually handling the error (e.g., returning an error code to their callers). Unwinding previous operations is however

```
1  /* drivers/media/platform/s5p-g2d/g2d.c */
2  static int g2d_probe(struct platform_device *pdev) {
3      ...
4      ret = v4l2_device_register(&pdev->dev, &dev->v4l2_dev);
5      if (ret)
6          goto unprep_clk_gate;
7      vfd = video_device_alloc();
8      if (!vfd) {
9          ret = -ENOMEM;
10         goto unreg_v4l2_dev;
11     }
12     ...
13     ret = video_register_device(vfd, VFL_TYPE_VIDEO, 0);
14     if (ret)
15         goto rel_vdev;
16     ...
17     dev->m2m_dev = v4l2_m2m_init(&g2d_m2m_ops);
18     if (IS_ERR(dev->m2m_dev))
19         goto unreg_video_dev;
20     ..
21 unreg_video_dev:
22     video_unregister_device(dev->vfd);
23 rel_vdev:
24     video_device_release(vfd);
25 unreg_v4l2_dev:
26     v4l2_device_unregister(&dev->v4l2_dev);
27 unprep_clk_gate:
28     ...
29 }
```

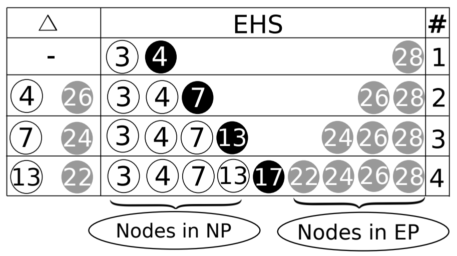
Figure 1: Example of the error-handling structure.

error-prone. To understand the characteristics of the handling of previous operations, we introduce the idea of leader and follower functions and use an example to describe the error-handling structure.

Leader and follower functions. Resources such as memory and locks are limited. As such, an operation against a resource, such as memory allocation, is typically accompanied by another operation that balances or recovers the resource. We define a function as a leader function if it initiates an operation against a resource. The operation typically either acquires or changes the state of the resource. Correspondingly, we define a function as a follower function if it recovers the resource. The leader function and the corresponding follower function constitute a *function pair*. Common function pairs include allocation/deallocation, lock/unlock, refcount increase/decrease, etc. As an example, Figure 1 shows three pairs of functions. The first pair is `v4l2_device_register()` and `v4l2_device_unregister()`, which initializes and cleans up the related objects such as refcounts and locks. The second pair is `video_device_alloc()` and `video_device_release()`, which allocates and releases the memory for video devices. The third pair is `video_register_device()` and `video_unregister_device()` whose functionality is similar to the first function pair.

Unique error-handling structure—EH stacks and deltas. We identify a unique and common error-handling structure and refer to it as *EH stacks and deltas*. We use the example in Figure 1 to illustrate the structure. In the error paths, follower functions are called to handle leader functions in a "stack" manner (i.e., the last follower corresponds to the first leader). In EH stacks, we use unfilled circles to represent the functions in the normal paths, gray-filled circles to show the functions in the error paths, and black-filled circles to indicate the errors or error checks. In the example, `v4l2_device_register()`, `video_device_alloc()`, and `video_register_device()` are leader functions and are called sequentially: 4–7–13. In case of an error in `v4l2_m2m_init()`, 17, the error path is 22–24–26. In the path, the corresponding follower functions are called in reverse order, hence we call the structure *EH stack*. Due to the complexity of error handling and the poor design of certain follower functions, in practice, the structure may not be honored, leading to DiEH.

In this example, there are multiple EH stacks, two of which are: 4–7–26 (path 1) and 4–7–13–24–26 (path 2). When we compare the unfilled lines and gray-filled lines in these two EH stacks, we can obtain the difference which is 7–24. We call the difference an *EH delta*. In this particular case, the delta consists of only one leader function and one follower function. As such, we can infer that functions `video_device_alloc()` and `video_device_release()` are a *function pair*. The inference does not require any domain knowledge or understanding semantic structure, thus it can be automated. In HERO, we will leverage EH stacks and deltas to precisely pair functions.



△	EHS	#
-	3 4 28	1
4 26	3 4 7 26 28	2
7 24	3 4 7 13 24 26 28	3
13 22	3 4 7 13 17 22 24 26 28	4

Figure 2: The EH stacks of the function in Figure 1. EHS = EH stack, EP = error path, NP = normal path, Δ = the EH delta of EHS_i and EHS_{i-1} where the EH stacks are numbered in the "#" column.

2.2 Disordered Error Handling

In this subsection, we present the definition, categorization, causes, and security impacts of DiEH.

2.2.1 Definition of DiEH

DiEH represents cases in which the follower functions are called in an incorrect order, redundantly, or inadequately. Thus, a DiEH case occurs if it satisfies the three conditions: (1) a function contains at least one error paths, (2) the function has at least one leader functions, and (3) in some error paths, the corresponding follower functions are not called in order, exactly once, or adequately. Informally, we define a DiEH case as follows.

Definition 1 *Let EP be an error path in a function, [LD] be the list of leader functions in EP, [FL] be the actual list of*

follower functions in EP. Suppose [FL]′ is the expected list of follower functions to appear in EP based on the foreknowledge of function pairs, then:
$\exists DiEH \in EP$, *if* $[FL] \neq [FL]'$

Specifically, $[FL] \neq [FL]'$ can occur due to three situations. (1) [FL] and [FL]′ contain the same set of follower functions but in different orders. (2) One or more follower functions are in [FL]′ but not in [FL]. (3) One or more follower functions are in [FL] but not in [FL]′. Based on the definition, we identify the key challenge in detecting DiEH as collecting [FL]′, which requires the foreknowledge of function pairs. In §4, we describe our new technique, which precisely identifies function pairs.

```
1  /* drivers/media/platform/rockchip/rga/rga.c */
2  static int rga_probe(struct platform_device *pdev) {
3    ...
4    pm_runtime_enable(rga->dev);
5    ...
6    ret = v4l2_device_register(&pdev->dev, &rga->v4l2_dev);
7    if (ret)
8      goto err_put_clk;
9    vfd = video_device_alloc();
10   if (!vfd) {
11     ...
12     goto unreg_v4l2_dev;
13   }
14   ...
15   rga->vfd = vfd;
16   ...
17   rga->m2m_dev = v4l2_m2m_init(&rga_m2m_ops);
18   if (IS_ERR(rga->m2m_dev)) {
19     ...
20     goto unreg_video_dev;
21   }
22   ...
23   /* Create CMD buffer */
24   rga->cmdbuf_virt = dma_alloc_attrs(...);
25   rga->src_mmu_pages = (unsigned int *)__get_free_pages(...);
26   rga->dst_mmu_pages = (unsigned int *)__get_free_pages(...);
27
28   ret = video_register_device(vfd, VFL_TYPE_VIDEO, -1);
29   if (ret) {
30     v4l2_err(&rga->v4l2_dev, "Failed to ...");
31     goto rel_vdev;
32   }
33   ...
34   return 0;
35
36 rel_vdev:
37     video_device_release(vfd);
38 unreg_video_dev:
39     video_unregister_device(rga->vfd);
40 unreg_v4l2_dev:
41     v4l2_device_unregister(&rga->v4l2_dev);
42 err_put_clk:
43     pm_runtime_disable(rga->dev);
44     return ret;
45 }
```

Figure 3: An example showing various new DiEH bugs, found by HERO, in a single function in the Linux kernel.

2.2.2 Classification of DiEH bugs

In §2.2.1, we present three situations that result in $[FL] \neq [FL]'$. In this section, we present concrete cases for them.

Incorrect-order follower functions. Using correct follower functions but in an incorrect order can cause security bugs. For example, Figure 3 contains a use-after-free bug caused by using the follower functions in an incorrect order. The function `video_device_alloc()` is called in line 9, which is before the function call of `video_register_device()` in line 28. Thus, the corresponding follower function `video_device_release()` should be called after `video_unregister_device()`. However, the error path starting from line 31 calls `video_device_release()` before `video_unregister_device()`. This incorrect-order DiEH results in a use-after-free because `rga->vfd` is an alias of `vfd` (line 15), and line 39 uses `rga->vfd` which uses the memory freed by line 37.

Redundant follower functions. Follower functions of a leader function might be called redundantly. This can happen when either multiple follower functions are called by mistake, or a follower can actually correspond to multiple leader functions, which confuses developers. For example, in Figure 3, the follower function `video_unregister_device()` (line 39) is called even when the call of its leader function `video_register_device()` (line 28) returns an error, which is unnecessary, leading to a redundant DiEH bug. A correct case is to call `video_unregister_device()` *only* when its leader function `video_register_device()` succeeds. Common issues resulting from redundant DiEH include double free, double unlock, double refcount, etc.

Inadequate follower functions. This situation refers to that necessary follower functions are missing. Common cases include missing release, missing unlock, missing refcount decrease, etc. For example, Figure 3 also contains several missing-release bugs. When the call of the function `video_register_device()` failed (line 28), pointers `rga->cmdbuf_virt`, `rga->src_mmu_pages`, and `rga->dst_mmu_pages` are not released, which are allocated in lines 24, 25, and 26. These bugs are common, and the Linux kernel has more than two thousand patches to fix inadequate follower functions. Prior research has studied such inadequate follower functions like missing resource release [48]; however, the other two situations, incorrect-order and redundant follower functions remain unexplored.

2.2.3 Causes of DiEH

In this section, we summarize three major causes of DiEH based on our empirical analysis, which are hard to avoid.

Poor design of follower functions. Different programmers have various programming habits. Some follower functions are hard to use if they do not follow the programming convention. For example, functions `pm_runtime_get_sync()` and `kobject_init_and_add()` are called many times in the Linux kernel, but they are actually poorly designed. Both of these functions would increase the kernel refcount, even when they failed, violating good design practice. Some Linux maintainers we interacted with even complained that *"if you follow the common convention, you will get it wrong."* Though patterns and anti-patterns in API design are widely discussed [46], factors such as a need for backward compatibility and a large developer base makes API design a challenge.

Complexity and dependency of cleanup operations. Error paths are prevalent in a large program, and each may contain various cleanup operations (follower functions). Our analysis shows that, in the Linux kernel, there are more than 120K intra-procedural error paths, and `61.6%` of them include at least one follower function, and on average, there are 2.46 follower functions per error path. The most complex error path contains 143 follower functions. More critically, some follower functions are dependent on each other, e.g., a parameter of a memory-release function is a nested field of a parameter of another function. The dependency requires the follower functions to be called in a specific order.

Custom follower functions. Different modules employ different leader and follower functions. We determined that about 80% of function pairs in the Linux kernel are custom (§7.1). These function pairs are defined and used within a specific module such as a driver. Avoiding DiEH bugs requires programmers to be knowledgeable about all the custom functions, which is a burden.

2.2.4 Prevalence of DiEH

It is hard to avoid DiEH due to the causes mentioned in §2.2.3. After manually checking 100 CVE-assigned vulnerabilities in 2019 from the Linux kernel, we found that DiEH causes 22 of them. Further, after checking the patches over the past two years from the Linux kernel, we found 42% of memory leaks and 45% of double-free bugs are due to DiEH. These results indicate that DiEH bugs are prevalent in the OS kernels, and can cause a wide range of security impacts. By employing a systematic detection, we expect to find many DiEH bugs.

2.2.5 Security Impacts of DiEH

Most DiEH bugs can cause severe security impacts, depending on their contexts. Common security impacts of DiEH include use-after-free, double-free, NULL-pointer dereference, deadlock, memory leak, refcount leak, etc. In the following, we showcase how DiEH leads to critical security issues.

Memory corruption. DiEH bugs often cause critical memory corruption such as use-after-free, double free, and NULL-pointer dereference. In Figure 3, we have shown how an incorrect-order DiEH leads to a use-after-free. Also, redundant and inadequate DiEH can lead to memory corruption. For example, CVE-2019-15504 [32] is a double-free vulnerability in the Linux kernel caused by redundant DiEH. This vulnerability has the highest CVSS score (10), which may be exploited remotely to compromise the system. CVE-2019-15292 [31] is a use-after-free vulnerability in the Linux kernel caused by inadequate DiEH. This vulnerability also has the highest CVSS score (10), which can compromise the confidentiality, integrity, and availability of the system. Further, DiEH is a source for NULL-pointer dereference. For example, CVE-2019-15923 [34] is a NULL-pointer dereference vulnerability in the Linux kernel, which is caused by inadequate DiEH.

Privilege escalation. DiEH can cause privilege escalation, which is considered one of the most critical security problems. CVE-2019-5607 [37] and CVE-2016-0728 [30] are refcount-leak bugs found in FreeBSD and the Linux kernel. Both bugs can cause privilege escalation because an overflowing reference count triggers a use-after-free. Similarly, CVE-2019-0685 [29, 39] is a refcount-leak vulnerability in Windows, which can be exploited to launch privilege-elevation attacks. These results show that DiEH can also compromise the confidentiality and integrity of OS systems.

Denial-of-Service. The most common security impact of inadequate follower functions is resource leak such as memory leak and refcount leak. Memory leaks in the OS kernels are considered critical because they can crash the whole system and lead to Denial-of-Service (DoS) [33, 35]. Figure 3 is vulnerable to a memory leak in case `video_register_device()` fails.

3 Overview

Based on the study, we develop an effective detection system for DiEH bugs. In this section, we first discuss the challenges in identifying DiEH, and then outline our solutions.

3.1 Challenges in Identifying DiEH

While prior research [28, 48] attempted to detect cases of inadequate follower functions, cases of incorrect-order and redundant follower functions remain unexplored. Systematically detecting DiEH bugs involves three major challenges.

Analysis of error-handling structures. HERO first needs to analyze the error-handling structures, so as to extract EH stacks and deltas, which will be leveraged to identify function pairs. In particular, HERO needs to: (1) identify the normal paths (e.g., ④, ⑦, and ⑬ in Figure 1) and error paths (e.g., ㉒, ㉔, and ㉖ in Figure 1) in a function, (2) identify the leader and follower functions in the normal and error paths.

Normal and error paths are often interleaved in the program. Thus, to identify and distinguish them, we need to know their demarcation points, which is a non-trivial task. In a function, there may be many normal and error paths, but only some of them should be associated together. Thus, we should map the normal paths to their corresponding error-paths. Further, numerous functions are called in normal and error paths, but not all of them should be called in pairs. Therefore, we need to extract the leader functions from normal paths and follower

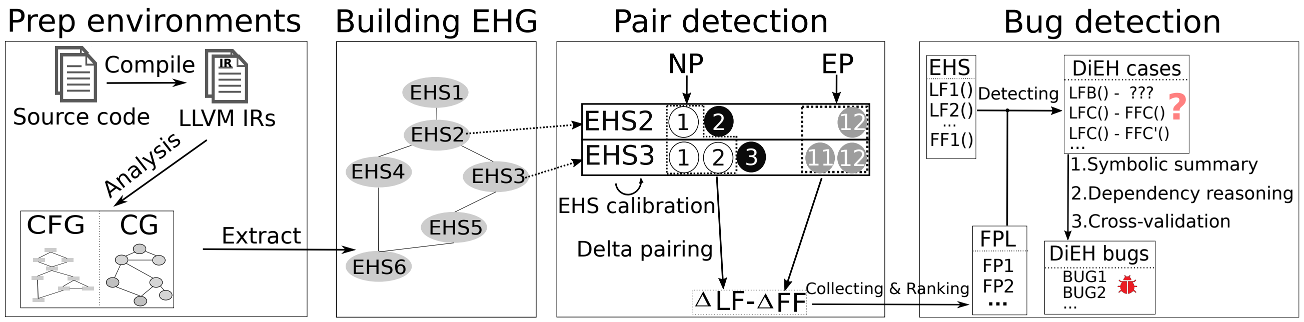


Figure 4: An overview of HERO. It has four steps; by taking the source code of a program, it automatically reports ranked DiEH bugs. CFG = Control flow graph, CG = call graph, EHG = error-handling graph, EHS = error-handling stack, LF = leader function, FF = follower function, FPL = function-pair list, Sym sum = symbolic summary.

functions from the corresponding error paths. More importantly, as we will describe in §4, the pairing of a leader and a follower function can be either conditional or unconditional. A precise pairing analysis requires distinguishing them, which is hard.

Function-pair identification. According to our definition of DiEH (§2.2.1), the detection of DiEH is essentially checking [FL] $\neq$ [FL]$'$, which requires the foreknowledge of leader-follower function pairs. This would previously require domain knowledge or manual efforts, and its automation is a significant challenge. In particular, programs tend to extensively use custom functions—defined and used in a specific module. Such functions have a limited number of uses, so existing mining-based inferences may not work. In fact, our study estimates that 80% of follower functions in the Linux kernel are custom. Moreover, there are a number of different classes of leader-follower pairs, such as allocation/deallocation, lock/unlock, getter/putter, and register/unregister. As a result, previous works (e.g., [15, 61]) either assume that function pairs are provided or only target a specific class of common pairs.

Elimination of harmless DiEH cases. The checking of "[FL] $\neq$ [FL]$'$" returns DiEH cases which may not be harmful, i.e., false positives. There are two major causes of harmless DiEH cases. First, by nature, the path of an EH stack often involves path constraints (i.e., return-value check). A path is infeasible if conflicting constraints exist. The intuitive solution, symbolic execution, may not work in complex programs. Second, for the incorrect-order DiEH cases if the follower functions are independent, their order does not matter. Therefore, for these incorrect order DiEH cases, we need to understand the potentially complicated data dependencies among different follower functions to determine potential bugs. Note that redundant and inadequate DiEH cases do not have this challenge because they are independent of ordering.

3.2 HERO Techniques

To address the challenges, we propose multiple new techniques. In this section, we briefly introduce them.

Understanding error-handling structures. To identify function pairs, HERO first automatically understands the error-handling structures and represents them with a graph. This technique starts with identifying *error checks*. An error check is basically an `if` statement that checks whether a function returns an error code. For example, lines 5, 8, 14, and 15 are error checks in Figure 1. With the identified error checks, we can identify normal paths and error paths—the code path prior to the error check is the normal path, while the taken path (as opposed to the fall-through) of the error check is the error path. This technique also identifies leader and follower functions on the normal and error paths by removing irrelevant functions (e.g., via dependency analysis), and stores all the information in a graph, referred to as the error-handling graph or *EHG*. We will present details of the technique in §4.1.

Pairing functions with EH deltas. To identify function pairs in a program, we propose delta-based analysis, which can precisely pair functions even when they are custom (i.e., only with a small number of occurrences). The key insight is that follower functions in the error path are called in a specific (reverse) order, corresponding to the leader functions in the normal path, which constitutes EH stacks, as shown in Figure 2. More importantly, when we compare two *adjacent* EH stacks, we naturally obtain the EH delta, which oftentimes has only one leader function and one follower function—therefore, we can infer that this follower function is paired to the specific leader function. For example, by comparing EH stacks 1 and 2 in Figure 2, we obtain the EH delta, ④–㉖, which constitutes a function pair. Similarly, EH stacks 2 and 3 generate the EH delta, ⑦–㉔, forming another function pair. To further improve the pairing precision, we propose *EH-stack calibration* to distinguish conditional and unconditional pairs. Details are presented in §4.

Detecting DiEH bugs with identified pairs. To detect DiEH bugs, we first detect DiEH cases, and then remove infeasible and harmless cases to report DiEH bugs. HERO detects DiEH cases by comparing the follower function list `[FL]` with the expected follower function list `[FL]`′. To remove infeasible DiEH cases, we propose a scalable *symbolic summary* for conflicting constraints, which helps eliminate infeasible paths. In addition, to remove harmless incorrect-order DiEH cases, we propose *follower-dependency reasoning*, which finds independent follower functions whose order does not matter. Finally, we provide a ranking of detected DiEH bugs to facilitate manual confirmation. More design details will appear in §5.

3.3 The HERO Framework

We now briefly introduce the workflow of HERO, shown in Figure 4. HERO consists of four steps. (1) *Preparing the analysis environment.* HERO first prepares the analysis environments by compiling the source code to LLVM IRs (bitcode files), and building the control-flow graph (CFG) and call-graph (CG) for the program. (2) *Constructing error-handling graph.* Second, HERO analyzes the unique error-handling structures to extract errors and EH stacks for each function. After that, the HERO constructs an EHG to record all the information. (3) *Leader-follower pairing.* Third, based on the EH stacks, the HERO computes the EH deltas and leverages them to pair functions. (4) *DiEH detection.* Finally, based on function pairs and the EHG, HERO detects DiEH bugs in the program. As a result, HERO reports the DiEH bugs. The reports include details such as disordered situations and suggested fixes.

4 Delta-Based Precise Function Pairing

A key challenge to detect DiEH bugs involves identifying function pairs including custom ones. We propose a novel technique that leverages the unique error-handling structure—EH stacks and deltas—to precisely pair functions. In this section, we present the design of the pairing analysis.

4.1 Extracting Error-Handling Structures

Identifying error checks, normal and error paths. To extract EH stacks of a function, we first identify error checks to collect normal paths and error paths. To identify error checks, we collect common error codes such as `ENOMEM`, and common error-handling functions such as `pr_err()` and `panic()`; §6 presents more details. Such error codes and error-handling functions are typically uniformly defined in dedicated header files. HERO regards a path as an error path if it returns an error code or a NULL pointer, or calls at least one error-handling functions. This design is consistent with existing works [19, 20, 27, 48].With the identified error checks, we naturally collect both normal paths and error paths of each error check. A path is represented with a list of code blocks, and a function can potentially contain a large number of paths.

Filtering follower functions by removing noises. Not all the functions in the normal and error paths should be paired, e.g., `kprintf()`. Therefore, we want to remove irrelevant functions. We first remove noisy functions in the error paths, i.e., filtering follower functions. We observe that unlike normal paths, error paths tend to be much simpler, in which irrelevant functions are typically commonly used error-messaging (e.g., `dev_err()`) and exiting (e.g., `panic()`) functions. Therefore, we remove such functions, and details are presented in §6.

Filtering leader functions through data dependency. Compared to error paths, normal paths are more complicated, which call diverse functions. As such, we instead employ data dependency to filter potential leader functions, given that we have already selected potential follower functions mentioned above. The insight is that follower functions clean up resources obtained by or operations performed by leader functions; a leader function and the corresponding follower function should be connected through variables. For example, `kfree()` takes the pointer returned by `kmalloc()` as the parameter. With the insight, we select potential leader functions based on data dependencies on the selected follower functions. Specifically, if the return value or a parameter of a function is used by a follower function, we select it as a potential leader function. To be conservative, our dependency analysis is field-insensitive. That is, different fields of an object are also considered dependent.

Constructing EH stacks. With the potential leader and follower functions collected, we next construct EH stacks. An EH stack consists of three parts: `<ERROR, [LD], [FL]>`. Here, [LD] is a non-empty list of leader functions, which are in the normal path; [FL] is a non-empty list of follower functions, which are in the error path; ERROR is the call-site to the error-generating function corresponding to the error check. We bypass the path-explosion problem by collecting EH stacks using intra-procedural analysis.

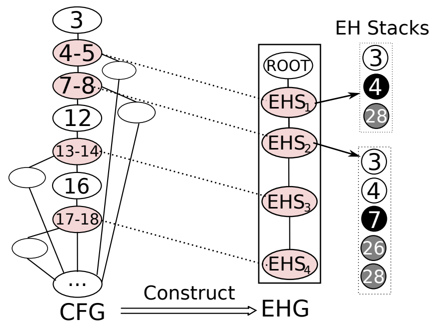


Figure 5: Constructing the EHG for the function in Figure 1. EHS = EH stack. With the EHG, we can quickly find adjacent EH stacks to compute EH deltas.

Building error-handling graph. To record all the identified error-handling information for a function, we then build an error-handling graph (EHG). Another purpose of building EHG is to also capture the adjacency of EH stacks, which facilitates the pairing analysis. The nodes of the EHG are EH stacks. Edges are added to connect the EH stacks based on the control-flow dependencies of the error checks associated with the EH stacks.

Specifically, given a function, to build the EHG, HERO first constructs the nodes by identifying all the basic blocks that include an error check. Then from a selected basic block and its error check, HERO collects all the EH stacks associated with this error, and further records these EH stacks into the nodes of EHG. After that, HERO traverses the CFG and connects these nodes in the EHG based on their control-flow relationship. Figure 5 shows an example of creating the EHG based on the control-flow graph (CFG) of the function in Figure 1. Four shadow nodes, which mark lines 4-5, 7-8, 13-14, and 17-18 in the CFG, indicate the code blocks containing error checks.

4.2 Delta-Based Pairing Analysis

In §4.1, we extract EH stacks and build the EHG. In this section, we present how we perform the delta-based pairing analysis, which computes EH deltas by comparing two adjacent EH stacks to precisely identify function pairs.

Computing EH deltas. As already described in §3.2, we leverage EH deltas to precisely identify function pairs because EH deltas often precisely capture an extra leader function and the extra follower function. To compute the EH deltas, we pick each two adjacent EH stacks from the EHG and compare them to generate the delta. In less than 5% of cases, an EH delta contains more than one leader or follower functions; in this case, we still try to pair them but in reverse order with the help of data-dependency analysis. That is, for the last follower function, it will be paired to the first leader function if they have data dependencies; otherwise, we would try to pair it with the second leader function. Following this order, and if finally, this follower cannot be paired with any leader function, we would further calibrate the EH stack (shown in the next paragraph) and try to pair it with the error-generating function. HERO would drop the leader or follower functions if they eventually cannot be paired, which is uncommon. Note that HERO may pair one leader to multiple follower functions, and vice versa, which means that the pairing output is "many-to-many" mapping between leader and follower functions.

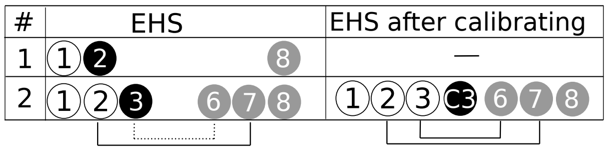


Figure 6: Calibrating EH stack. EHS = EH stack.

Calibrating EH stacks. Before we present the pairing algorithm, we first describe the challenge. We divide function pairs into two categories – *conditional pair* and *unconditional pair*. In most cases, function pairs are conditional. That is, a follower function is necessary only when the leader function succeeds. For example, if `kmalloc()` fails, `kfree()` is unnecessary. However, there are also some unconditional pairs. That is, despite the failure of the leader function, the corresponding follower function is still required. For example, as mentioned in [6], when `kobject_init_and_add()` fails, its follower function `kobject_put()` is still required to clean up the related objects. To correctly construct EH stacks, we must distinguish conditional and unconditional function pairs; otherwise, the pairing results would be unreliable.

We propose to calibrate the EH stacks, which identifies unconditional pairs and adjusts EH stacks. The idea is based on an observation that an unconditional pair will result in an extra follower function in the EH delta of two adjacent EH stacks. Therefore, we detect unconditional pairs based on such extra follower functions. Once unconditional pairs are detected, we adjust EH stacks by extending their normal paths to contain the error-generating functions. We use Figure 6 to illustrate the calibration. EHS_1 and EHS_2 are adjacent in the EHG. By comparing them, we find that the leader function ②, and two follower functions ⑥ and ⑦ show up on the EH delta, in which an extra follower exists. After checking the data dependencies between ❸ and ❻, we deem that ❸ is an unconditional leader function, and functions ③ and ⑥ constitute an unconditional pair. We thus calibrate EHS_2 by including ③ in its normal path. This way, HERO effectively eliminates noises introduced by unconditional pairs.

The pairing algorithm. Putting the steps together, HERO first traverses the EHG to get each EH stack and its successor in the EHG; these are two adjacent EH stacks. Specifically, HERO analyzes every path and differentiates error paths from normal ones to collect adjacent EH stacks. As such, HERO handles conditionals—if there is a conditional statement, HERO will simply collect two paths. Then, HERO calculates their EH delta. If the EH delta indicates an unconditional pair, HERO calibrates the EH stack and re-calculates the EH delta. Using the EH deltas, HERO collects the function pairs. The output of this algorithm is a list of potential function pairs. Note that this algorithm also includes a ranking mechanism that will be presented in §6.

5 Detection of Disordered Error Handling

With the identified function pairs and constructed EHG, HERO automatically detects DiEH bugs. The detection works with two phases: detecting DiEH cases, and reporting DiEH bugs by removing infeasible and harmless cases.

Detecting DiEH cases. HERO employs an intra-procedural, flow-sensitive static analysis to check each path and its corresponding EH stack in functions. At a high level, each EH stack contains a list of leader functions `[LD]` as well as a list of follower functions `[FL]`; after that, HERO computes the expected list of follower functions `[FL]`$'$ and compares it with `[FL]`. HERO reports cases in which $[\text{FL}] \neq [\text{FL}]'$ as DiEH cases. HERO also categorizes the DiEH into incorrect-order, redundant, and inadequate cases based on the classification rules presented in §2.2.3.

5.1 From DiEH Cases to DiEH Bugs

A DiEH case can be infeasible or harmless. In this section, we present our techniques for eliminating such cases to confirm DiEH bugs. We also provide a ranking mechanism to prioritize DiEH cases.

Eliminating infeasible paths by detecting conflicts. HERO statically finds normal and error paths to detect DiEH. If a path is infeasible (i.e., containing conflicting path constraints), the detected DiEH would be a false positive. To remove such false positives, we aim to eliminate infeasible paths. An intuitive strategy is to employ traditional symbolic execution, which is not scalable and can easily lead to path explosion, not to mention that our target programs are complex. To address this problem, we propose a scalable *symbolic summary* for each function, which intra-procedurally captures *conflicting* constraints among the variables such as, conditional variables and return values. When a path contains such conflicting constraints, we deem it infeasible.

Specifically, the symbolic summary consists of two steps: (1) collecting constraints from the path under analysis, (2) checking the existence of conflicting constraints. In the first step, HERO analyzes the current path and collects constraints from every conditional statement, such as `if (flag == True)`. Further, HERO extracts changes against the variables of collected constraints that we are certain about, such as constant assignment like `flag = false`. If a change is uncertain, e.g., assigned with an unknown variable, we regard the case as an uncertain constraint. In the second step, HERO checks collected constraints, and treats the path as infeasible if it has conflicting constraints. (e.g., the first constraint is `flag == false` and then the second constraint is `flag == True`.) The symbolic summary conservatively regards all the uncertain constraints as solvable, ensuring the precision of the removal of infeasible paths. This simple approach can quickly and reliably (i.e., the infeasibility is determined) remove infeasible paths without handling complicated uncertain constraints, which is a lightweight version of under-constrained symbolic execution.

Figure 7 shows an example of conflicting constraints causing a false positive in detecting DiEH. For this case, without the symbolic summary, HERO would detect a missing-follower DiEH case—the release function, `kfree(max3421_hcd->rx)`, is missing in path ③–④–⑦–⑪–⑫–⑬–⑮–⑯–⑲. This is however a false positive because constraints `if(!hcd)` (line 4) and `if(hcd)` (line 16) are conflicting in the path. With the symbolic summary, when analyzing this path, HERO will first collect the constraint `hcd != NULL` from line 4 and the constraint `hcd == NULL` from line 16. Then, HERO determines that the constraints are conflicting, and thus the path is infeasible. In addition to checking conflicting constraints from a called function, our technique will check the ones from the current function and use them to eliminate infeasible paths. To collect more conflicting constraints, we also employ alias analysis, which is based on the LLVM alias analysis infrastructure [43] to map the variables involved in the constraints.

```
1  /* drivers/usb/host/max3421-hcd.c */
2  static int max3421_probe(struct spi_device *spi) {
3      hcd = usb_create_hcd( ... );
4      if (!hcd)
5          goto error;
6      ...
7      max3421_hcd->rx = kmalloc( ... );
8      if (!max3421_hcd->rx)
9          goto error;
10
11     max3421_hcd->spi_thread = kthread_run(...);
12     if (max3421_hcd->spi_thread == ERR_PTR(-ENOMEM))
13         goto error;
14     ...
15 error:
16     if (hcd)
17         kfree(max3421_hcd->rx);
18     ...
19     return retval;
20 }
```

Figure 7: Example of the conflicting constraints.

Our evaluation shows that our solution is effective, and it reduces about half of the false positives cases without introducing additional false negatives, which makes the results manageable for manual analysis. Nevertheless, our symbolic summary is based on intraprocedural analysis and only considering the most intuitive conflict constraints, and thus it still cannot handle the false positives caused by complicated conditions. The evaluation results in §7.3.1 show that, finally, for bug detection, 23% of false positives are caused by complex conditions, which cannot be handled by the symbolic summary. However, our intra-procedural symbolic summary and feasibility testing are highly scalable, with no noticeable slowdown in the analysis.

In general, we can compare the symbolic summary with the symbolic execution from the following aspects: (1) both do not have false-positive in theory, (2) the symbolic summary has false-negatives due to the intraprocedural analysis and also missing handling complex constraints, and (3) the symbolic summary performance is much better than symbolic execution because the front one would not suffer from complex constraint solving or copying state for the forked process, which only simply compares the must conflict constraints in a given path.

Eliminating harmless cases via dependency reasoning. HERO reports any incorrect-order follower functions as potential DiEH. However, we observe that if two follower functions are independent, it is typically harmless to call them in staggered order. Therefore, we eliminate such independent cases. Specifically, we employ dependency reasoning to find independent follower functions. To be precise, we employ `MustAlias` analysis [43] and field-sensitive analysis. We apply the data-dependency analysis to the parameters and return values of the follower functions. If data dependency is found, we keep the DiEH cases. This technique can effectively remove the harmless incorrect-order DiEH cases.

Ranking reported bugs through cross-validation. To alleviate the manual effort in confirming DiEH bugs, HERO further ranks reported cases by employing cross-validation [14] across the cases. HERO calculates the percentage of error paths that encounter this problem. A lower percentage indicates that the DiEH case is an outlier and is more likely a bug. HERO then ranks the bugs based on the percentage in ascending order, for each category.

6 Implementation of HERO

We implement HERO based on LLVM-10 as multiple passes that identify error-handling structures, construct the EHG, perform delta-based pairing analysis, and detect DiEH bugs. We also implement multiple Python scripts for pairing and bug ranking. HERO is implemented with 5.5K lines of code in C++ and 800 lines of code in Python. In this section, we present some interesting implementation details.

Removing irrelevant functions in error paths. Compared to normal paths, error paths are often simple. Typically, irrelevant functions can be either (1) error-logging functions (e.g., `dev_err`), which log error messages, or (2) exit functions (`panic`), which terminate the execution. We employ two methods to eliminate such functions. First, we find that error-logging functions have clear patterns, e.g., having variadic and format parameters. We identify such functions by using pattern-matching. Second, to collect terminating functions, we identify wrapper functions that internally call primitive ones like `panic()`, `abort()`, and `exit()`. In total, we collect 537 irrelevant functions that are excluded from the pairing.

Ranking function pairs. The pairing analysis is precise for most cases but still has some false positives (see Figure 8) due to limitations with static analysis. We thus also provide a ranking mechanism against the pairs. The key insight is that for a true function pair, the occurrences of the leader function should close to the occurrences of the follower function. Given a function pair, we count the total occurrences of a leader function as `LT` and the total occurrences of its follower function as `FT`. Then, we count the frequency of function pair occurrence in the program as `PT`. Finally, we define the *paired rate (PR)* as $PR = \frac{PT^2}{FT*LT}$ and use it to rank the pairs in descending order. If PR approaches one the leader function and follower function are always used together; on the other hand, if PR approaches zero, the leader and the follower are rarely paired. Our evaluation (see §7.2) shows that such ranking can effectively squeeze most of the false positives into the bottom of the list, which can be eliminated easily.

7 Evaluation

We conduct our experiments on an Intel Xeon CPU server that has 48-cores and 256GB RAM, and runs Ubuntu-18.04 OS. All experiments use `-O2` optimization to generate bitcode (LLVM IR) files. We evaluate HERO on both system and application software, including Linux (commit #: 4d856f72c10) and FreeBSD (commit #: c54c07625bd) kernels, and OpenSSL library (commit #: 7821585206).

Analysis time and program complexity. Table 1 shows the analysis time for each component across different systems. Even for the Linux kernel, which has 17.7 million lines of code, the pairing finishes within one hour, and the detection finishes in about 10 hours. The results confirm that HERO is efficient and can scale to large programs. Note that HERO is currently single-threaded; multithreading can further improve its efficiency.

Target program	Lines of Code	IR files	Time for pairing	Detection time
Linux kernel v5.3	17.7M	18,071	48 min	10 h 16 min
FreeBSD v12.1	4.8M	1,483	10 min	2 h 28 min
OpenSSL	450K	1,902	53 sec	11min

Table 1: Analysis time of HERO and the complexity of programs.

Preparing pair sets. To evaluate our delta-based pairing, we prepare two sets of function pairs. The first set is the *reported pair set*, which includes 150 randomly selected unranked functions pairs identified by HERO. As will be detailed in §7.2, 89 of them are true pairs, while 61 are false pairs. The second set is the *ground-truth pair set*, which includes 86 function pairs of various types. We collected this set from 15 random source files across different subsystems of the Linux kernel; these files contain 26K lines of source code.

7.1 Characteristics of Identified Pairs

HERO detects more than 7.5K, 416, and 323 potential function pairs in the Linux kernel, OpenSSL, and FreeBSD, respectively. To further characterize these pairs, we pick the Linux kernel because it is the most complex. We first use script code to statistically select common keywords in the names of paired functions, and use the keywords to empirically classify pairs. The common keywords and the classification are summarized in Table 2. Interestingly, the keywords of a pair usually have the opposite meaning, indicating the paired operations, e.g., alloc/dealloc and increase/decrease.

Classes (Proportion)	LF Operations	FF Operations
Resource acquisition (50.2%)	alloc, new, request, create init	free, release, erase, destroy, remove fini, finish, deinit, uninit
Lock (4.4%)	lock, down	unlock, up
Refcount (12.5%)	get, inc	put, dec
Device related (18.2%)	register charge, on, enable	unregister, deregister uncharge, off, disable
Bit operation (0.7%)	set	clear
Others (33%)	apply, pin, assert join, add, map reserve begin, start, open setup	revert, unpin, deassert leave, remove, unmap delete, del end, finish, stop, exit, close clean, cleanup

Table 2: Common classes of function pairs in the Linux kernel. LF and FF are leader and follower functions, respectively.

Custom function pairs. A strength of our delta-based pairing analysis is that it does not require a large number of occurrences of pairs for inference or mining. As a result, HERO is capable of identifying function pairs that are composed of custom leader and follower functions, and thus it can identify a significantly larger number of pairs. To confirm that, we identify custom pairs from the 89 true pairs in the aforementioned reported pair set. We find that 71 are defined and used in specific modules, thus are custom. Therefore, the result shows that `79.8%` of them are custom.

7.2 Precision and Recall of Delta-Based Pairing

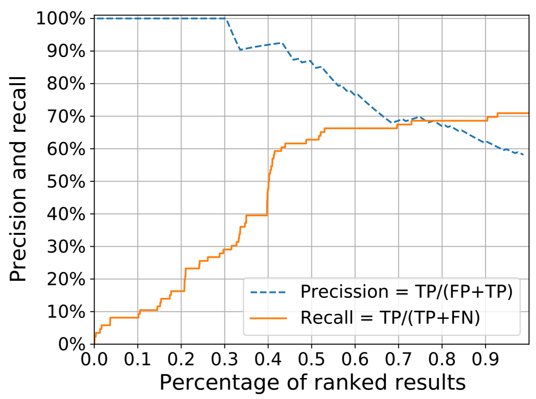


Figure 8: Precision rate and recall rate of pairing results. TP = True positives, FP = False positives; FN = False negatives.

Precision of the pairing. We first evaluate the precision rate (i.e., TP / (FP + TP)) of the pairing and the ranking mechanism. Manually confirming all the detected pairs is impractical, so we reuse the "reported pair set" which contains 150 unranked function pairs. In particular, we manually confirm the pairs through their names, semantics, functionalities, and usage by reading the code and comments. We found that functions names are very helpful in the confirmation because they contain opposite keywords (e.g., alloc/dealloc) and follow similar structures. Our confirmation shows that 89 are true function pairs. Automatically pairing the functions in large programs like Linux, where custom functions are prevalent, is very challenging. We believe the precision is already promising. However, to further improve the precision, we also provide a ranking mechanism, as shown in §6. The evaluation results show that the ranking mechanism can help exclude most of the false positives caused by irrelevant functions. As we discussed in §6, besides the 537 error-handling functions such as `warn()` , HERO treats all other functions as *potential* leader/follower functions. Thus other irrelevant functions can still incur noises. However, the false positives caused by these irrelevant functions can be further filtered out by the "paired rate," which is based on the fact that the irrelevant functions are often not paired to its leader or follower functions. For example, the function `__memcpy()` is an irrelevant function for functions pairs, but HERO still paired `__memcpy()` and `kfree()` twice in the Linux kernel. Nevertheless, `__memcpy()` is called more than 27K times, and `kfree()` is called more than 30K times in the linux kernel. Thus, the paired rate for function `__memcpy()` and `kfree()` is nearly 0, which means that they are not really a function pair. Figure 8 shows the precision evaluation for ranked pairs: For the top 30% of ranked pairs, the precision is 100%, and even for the 75% of the ranked pairs, the precision is about 70%.

We summarize three major causes of false positives. First, irrelevant functions still exist in EH stacks, introducing noises in the pairing. Second, function pairs may not appear in the same function but across different functions. The current implementation of HERO employs an intra-procedural analysis which would miss such pairs. Third, the detection of error paths, which is based on error codes, may misidentify normal paths and error paths of custom error codes are involved, leading to false positives as well.

Recall of the pairing. We also evaluate the recall rate (i.e., TP / (TP + FN)) of the pairing and the ranking mechanism using the aforementioned ground-truth pair set. The set contains 86 true function pairs; we find that HERO can detect 61 of them, leading to a recall rate of 0.71. Furthermore, Figure 8 shows the recall rate for the ranked pairs. Similar to the causes of false positives, false negatives are also mainly caused by (1) incorrect error-path identification and (2) noises in delta analysis.

	PF-Miner	PairMiner	HERO	HERO (30%)
Linux	-	94.7	303.3	128.2
Android	50.5	-	-	-

Table 3: Comparison with the closest pairing tools PF-Miner [23] and PairMiner [24]: Number of function pairs per million lines of source code. The top 30% of pairs identified by HERO are precise.

7.2.1 Comparison with Previous Pairing Analyses

We aim to compare HERO with related works on function pairing. We identify the following most relevant and recent works: PairMiner [24] and PF-Miner [23]. RID [28] also pairs functions; however, it focuses only on refcount-related ones and uses simple string matching (e.g., `*_inc/*_dec`), so we exclude it from the comparison. PF-Miner [23] first employs string matching (e.g., `new/delete` and `alloc/free`) to collect functions. Then, equipped with a mining algorithm, it statistically pairs the functions that often show up pairwise in the normal and error paths. After analyzing the C source code of the Android kernel, PF-Miner identifies 546 paired functions. PairMiner [24] shares similar approaches because it is built on top of PF-Miner. PairMiner identifies 1023 paired functions in the Linux kernel.

We compare HERO with these tools in how many function pairs are identified. Unfortunately, we cannot compare precision values because neither tool provided such numbers. Note that PF-Miner and PairMiner both employ simple mining (i.e., statistical counting) to collect pairs, we believe they inherently suffer from precision issues and could not support custom functions. Table 3 presents the details of the comparison. Specifically, HERO is agnostic to types of function pairs and supports custom functions. HERO also identifies significantly more pairs. Even if we select the top 30% of ranked function pairs, the number is significantly higher compared to either PF-Miner or PairMiner. We attribute HERO's effectiveness to its delta-based pairing analysis, which is precise and can support custom functions.

Evaluation against dependency-based pairing. WYSIWIB [22] employs data dependencies to pair alloc/dealloc function pairs. To compare HERO to such pairing, we extend the dependency analysis to all functions in the normal and error paths. As a result, such pairing reports about 200% more function pairs; however, we found the majority of them are false positives (wrong pairs), disqualifying it for the DiEH detection. This result shows that delta-based analysis can significantly reduce false function pairs and make results more precise.

7.3 Bug Detection

Based on the precision and recall trade-off shown in Figure 8, we choose the top 43.2% of function pairs for detecting DiEH bugs because it achieves a high precision (92.5%) and a reasonably good recall (60.4%). We then apply HERO to three target programs, the Linux kernel, the FreeBSD kernel, and the OpenSSL library, with corresponding 3276, 94, and 123 function pairs detected.

Based on these function pairs, HERO finally identifies 234, 2, and 3 DiEH bugs from the Linux kernel, FreeBSD, and OpenSSL library. The details of the identified bugs are shown in Appendix A. Among these detected DiEH bugs, 72% are caused by inadequate follower functions, 25% are caused by incorrect-order follower functions, and 3% are caused by redundant follower functions. Further, we found that the drivers of the Linux kernel are buggier than its core kernel. In the Linux kernel, the driver code accounts for 62% of the whole code-base; however, 87.6% of the found DiEH bugs come from the driver code, which means the bug density of the driver code vs. the core kernel is 4.3 : 1. We believe this is due to the following reasons: (1) drivers contain more custom functions, which are harder to be analyzed by previous static-analysis approaches; (2) many functions in drivers are used to support outdated devices and thus infrequently used or tested, and (3) compared to the core kernel, the drivers are less tested because existing dynamic-analysis tools require hardware devices or their emulation [4]. In the rest part of this section, we will present the causes of false positives and some interesting findings. For simplicity, we focus on the Linux kernel because it is the largest and the most complex.

7.3.1 False-Positive Analysis

HERO in total reports 454 potential DiEH cases in the Linux kernel, with 170 for incorrect-order, 40 for redundant, and 244 for inadequate DiEH cases. We manually check all these cases and regard a case as a true bug if it meets both of the following conditions: (1) the case is an actual DiEH case, and (2) the case would introduce at least one security issue. We confirmed 234 (thus, the false-positive rate is 48%) of them as true positives, with 58, 7, and 169 for incorrect-order, redundant, and inadequate DiEH bugs, respectively. To manually confirm these bugs, three researchers spent about a total of 16 man-hours. We believe the precision is reasonably good for static analysis–based detection against complex programs, and the manual effort for the confirmation is very manageable. Further, we patched and reported 230 bugs to the maintainers. The remaining 4 cases are removed in the latest version of the kernel. As of the submission of this paper, 125 of them have been accepted, and 105 have not received a response yet. We further analyzed the major causes of false positives.

First, we find that 23% of false positives are caused by complex path conditions that were missed by our under-constrained path-feasibility testing. We can mitigate these false positives by collecting more constraints from the complex path conditions.

Second, although some DiEH cases indeed exist, their impacts are prevented by some security operations such as enforcing a NULL check for a released pointer. Such cases contribute about 7% of false positives, and removing such false positives requires understanding the security operations. Third, our pairing analysis still misses the follower functions for some leader functions. This causes 18% of false positives. The remaining false positives are caused by other issues such as the aliasing problem in the static analysis, or incorrect detection of error paths.

7.3.2 Maintainer Feedback

During the bug confirmation and reporting, we found that function pairs are often used incorrectly. First, 8.2% of DiEH bugs are introduced by previous patches that incorrectly fixed error-handling bugs. For example, the patch (`6e5da6f7d824` [2]) in the Linux kernel fixed a DiEH bug caused by inadequate follower function. However, when this patch calls function `pm_runtime_get_sync()`, it still misses `pm_runtime_put()` when the call of function `pm_runtime_get_sync()` fails, which results in the bug. Second, even experienced Linux maintainers are not familiar with some follower functions, particularly custom ones. For example, few maintainers were aware that `kobject_put(P->kobj)` releases pointers `P` and `P->kobj`. These results are consistent with our previous findings in §2.2.3—cleanup operations are common, complex, and difficult to get right.

7.4 Security Impact Analysis

We not only confirm DiEH bugs but also empirically determine the impact of confirmed bugs. The impact is based on the involved variables and the contexts of each bug. Our determination is conservative—if a case is too complicated to analyze, we exclude it from the bugs. We reported the rest of the bugs to maintainers.

Type of bugs	Prop	Causes	CWE-ID [8]
Refcount leak	85.8%	IFL (75.6%), IOF (24.4%)	CWE-911
Memory leak	9.2%	IFL (77.3%), IFO (22.7%)	CWE-401
UAF/DF	1.7%	RFL	CWE-416, CWE-415
Double unlock	1.3%	RFL	CWE-765

Table 4: Most common security impacts of bugs found by HERO. CWE = common weakness enumeration. IOF = incorrect order of follower function, IFL = Inadequate follower functions, RFL = redundant follower function.

We summarize the impacts of the confirmed bugs in Table 4. 98.0% of the bugs would cause at least one of the security impacts mentioned in the table. Specifically, 3.0% of DiEH bugs would lead to use-after-free, double-free, or double-unlock, and all of them are caused by redundant follower functions. As we discussed in §2.2.5, these DiEH bugs can lead to critical security issues like memory corruption, DoS, privilege escalation.

Further, 85.8% of DiEH bugs would lead to refcount leak, with 75.6% of them caused by inadequate DiEH and 24.4% caused by incorrect-order DiEH. People often regard refcount leaks as general bugs but not security-critical ones. However, we argue that refcount leaks can also cause memory corruption. When a refcount field, especially the one with only 16 or less bits, is repeatedly incremented, it will finally overflow to zero, triggering a free and finally causing a use-after-free. As we discuss in §2.2.5, CVE-2016-0728 [30] is such an example. Moreover, there are many examples of exploiting refcount leaks for privilege escalation (e.g., CVE-2016-0728, CVE-2014-2851) and DoS (e.g., CVE-2019-9857). DoS, like crashing in the kernel, is security-critical for long-running servers.

Also, 9.2% of DiEH bugs would lead to memory leaks, with 77.2% of them caused by inadequate DiEH and 22.7% caused by incorrect-order DiEH. Memory leaks in the kernel can also be critical because they may result in DoS of the whole system. Assigned CVEs of kernel memory leaks include CVE-2020-15393 [40], CVE-2019-8980 [38], CVE-2019-5023 [36].

Type of entry points	Number of reachable bugs
System calls	180 (76.9%)
`ioctl` handlers	190 (81.2%)
IRQ handlers	185 (79.1%)
Total	199 (85.0%)

Table 5: The numbers of DiEH bugs that can be triggered from system calls, `ioctl` handlers, and IRQ handlers.

Triggerability analysis for detected bugs. To further understand the security impacts of bugs identified by HERO, we also tested the triggerability of them. Automatically confirming the triggerability of kernel bugs is still considered a challenging research problem. Dynamic analysis tools like OS fuzzers [7, 49, 51] have a low false-positive rate but suffer from performance issues and many false negatives. Therefore, similar to previous works such as SID [55], this evaluation focuses on identifying triggerable call stacks from the adversary-reachable entry points (e.g., system calls, `ioctl` handlers, and IRQs handlers) to the functions containing DiEH bugs. More details about the entry points are shown in Section VI.D of the SID paper [55]. Specifically, we analyze all the call instructions in the Linux kernel and leverage the state-of-the-art technique MLTA [25, 27] to handle the indirect calls, and finally build a complete call graph of the Linux kernel. Based on this call graph, given a vulnerable function that includes a DiEH bug, we traverse every entry-point function and extract the shortest path from each of them to the vulnerable function. If there is no path between a vulnerable function to all the entry points, we will mark the bug as non-reachable.

Table 5 shows the results of our triggerability analysis. 85.0% of DiEH bugs identified by HERO can be reached from at least one of the entry points, which means that it is possible for adversaries to intentionally trigger these bugs by constructing a specific input. Among these cases, 76.9% of them can be triggered through system calls, which means that they are relatively easier to be triggered by attackers and thus have a higher impact. The last column in Table 7 shows the specific triggerability information for each bug.

8 Discussion

Flow-sensitive vs. Path-sensitive. HERO is flow-sensitive and partially path-sensitive. Being path-sensitive can significantly improve the precision in both pairing and bug detection. However, full path-sensitive analysis cannot scale to large programs such as OS kernel yet. To eliminate the infeasible paths, §5.1 showed that HERO employed the symbolic summary to scalably identify conflicting path conditions, and further remove infeasible paths.

Generality. In the evaluation, we applied HERO to both kernels and a userspace program. The evaluation shows that applying HERO to a new program does not require extra manual effort. However, the precision of pairing analysis and DiEH detection slightly varies on different programs. In general, the detection precision for the Linux kernel is better than it for the FreeBSD and the OpenSSL library. We believe this is due to the reason that the error codes in the Linux kernel are well defined and used. Thus, HERO can better identify error paths and build the EHG.

HERO can be potentially extended also to analyze programs written in other languages or using other error-handling mechanisms. HERO detects DiEH bugs based on two factors (1) capturing errors and (2) analyzing the error-handling code. The logic of developers performing cleanups in error handling is mainly independent of the languages. However, factor (1) is dependent on the languages. To extend HERO, we need to instruct it to identify the errors and error-capturing mechanisms dependent on languages. For example, C++ typically uses the "try-catch" blocks, so HERO needs to further recognize the corresponding patterns in LLVM IR.

Exploitability of detected bugs. To further explore the security impacts of identified DiEH bugs, we need to determine the exploitability of these bugs. However, in this paper, we focus on detecting DiEH bugs instead of exploiting them. We believe that bug exploitation is a separate research topic and is out of our scope. To exploit DiEH bugs, the key is to trigger the corresponding errors, so that the error paths can be executed, which has been demonstrated by the previous works such as fault injection [44] and memory exhaustion [60]. Memory leak and refcount leak bugs can already cause the DoS problem if they can be steadily triggered through these techniques. For other DiEH bugs, after being triggered, adversaries can reuse existing attack techniques such as memory collision attacks [56] to generate the exploits.

Suggestions for avoiding DiEH. Based on our interactions with the kernel maintainers, we suggest several ways to avoid DiEH bugs. First, program developers should try to separate the cleanup operations from normal executions and handle the errors uniformly with a standardized error-handling structure. As shown in Figure 1, all the cleanup functions are called after the jump target `unreg_video_dev`.In contrast, in some cases, only parts of follower functions are used with a standardized error-handling approach, like this example, but other follower functions are called directly after the errors. This inconsistent error-handling often makes the code hard to maintain and can further lead to DiEH bugs. Second, API developers should follow the programming convention and provide clear instructions. For example, [6] shows the source code of function `kobject_init_and_add()`. In the latest version of the kernel, the comments clearly emphasize that "If this function returns an error, `kobject_put()` must be called to properly clean up the memory associated with the object," which, however, is missed before v5.2 and further incur lots of API misuse errors. This information can guide API users to correctly use this API. Third, API users should read instructions to understand how to use the API, instead of assuming its usage. At last, API users can cross-check the usages of API by looking into how other caller functions use the API. Fourth, checking the related patches of this API (e.g., through `git log`) is also helpful to know the common mistakes.

More applications of pairing analysis. Pairing analysis can be used in other areas, such as helping API users check function usage and bug detectors identify other types of bugs. For example, by identifying the lock/unlock function pairs, we can infer the functions that can execute concurrently and further detecting potential race conditions. These function pairs can be used to detect temporal bugs based on different temporal rules.

9 Related Work

Function pairs detection. As we compared in §7.2, several previous works also try to identify function pairs in large programs. In particular, Mao et al. [28] focused on identifying refcount-related bugs by comparing the inconsistent paths. To this end, they collected 800 pairs of refcount-related APIs by simply string-matching function names, e.g., `*_inc` and `*_dec`. WYSIWIB [22] analyzes the data dependencies of pointers to collect 304 pairs of allocation and deallocation functions. Compared to these works, HERO is not limited to a specific type of pair, and its delta-based pairing is more precise. PF-Miner [23] and PairMiner [24] have been introduced in §7.2, which employ data mining and string matching. To the best of our knowledge, PairMiner represents the state-of-the-art in automatically detecting various types of function pairs. Compared to HERO, since PF-Miner and PairMiner employ simple mining to collect pairs, we believe that the tools cannot support custom functions and are likely to suffer from precision issues, although they do not evaluate precision. Different from these static analysis tools, Bai et al. [3] employed dynamic tracing to collect 81 function pairs in four device drivers in Linux, which is not representative of the whole kernel.

Error-handling analysis. Many previous works also analyze error-handling code to detect bugs in software like OpenSSL and OS kernels. Rubio-González et al. [45] and EIO [18] detect error-propagation bugs in file systems. APEx [20], ErrDoc [53], and EPEx [19] reason about the error-code propagation in open-source SSL implementations, either automatically or via user definitions. Saha et al. [47] proposed an automatic approach, which can transform the coding style and structure of the error-handling code to a goto-based standardized error-handling strategy. Tang [50] proposed a tool to detect error code misuses in system programs. EESI [13] is a static analysis tool, which can infer C program function-error specifications through return-code idiom. EESI can identify inadequate and inverted error-checks, and also incomplete error handling bugs. An inherent difference is that these works focus on reasoning about the "handling" itself—if an error code is returned, passed, or handled in callers—instead of the cleanup operations before the handling.

Unlike previous works that aim to make error handling sufficient, EeCatch [42] instead detects exaggerated (or excessive) error handling which often causes crashes. EeCatch employs spatial and temporal cross-checking to identify irregular and *over-severe* error handling as potential exaggerated error-handling bugs. HERO differs from EeCatch in both research goals and approaches. First, HERO aims to detect the ordering issues in the error-handling code, instead of the incorrect severity level of error handling. DiEH causes not only crashes but also memory corruption. Second, HERO's key technique is the precise function pairing while EeCatch features the spatial and temporal cross-checking. To explore the structure of error-handling code, Thummalapenta et al. [52] proposed a mining algorithm, which mining sequence association rules and rule violations of function calls in a large number of the normal and error paths. Different from this work, HERO can precisely identify function pairs based on delta analysis, which can handle the custom functions.

Bug detection in error paths. There is also a line of research that focuses on finding bugs in cleanup operations in error paths. In particular, Saha et al. [48] proposed Hector, which identifies missing resource-release functions in the systems software. Hector assumes the pointer-returning functions are allocation functions, and the last pointer-usage function is a deallocation function. They identify the missing-release bugs by comparing the inconsistencies in different error paths. Mao et al. [28] implemented RID, which can identify refcount related bugs by analyzing the inconsistent paths in the function; oftentimes, the bugs are in error paths. Lawall et al. [21] proposed a tool to detect error-handling bugs in the Linux kernel and OpenSSL, which are related to API usage protocols. GUEB [16] and CRED [58] are static-analysis tools that can identify use-after-free bugs. All these works focus on a specific type of error-handling bugs, such as missing release. To the best of our knowledge, none of the tools could detect incorrect-order and redundant DiEH bugs, which requires precise and comprehensive identification of function pairs.

Bug detection with rules inference. Some previous works also identified bugs though rules inference based on code semantics. APISan [59] detects API misuses by analyzing rich symbolic contexts. Acharya et al. [1] proposed a mining technique to check the partial-order rules of API usages and detect related rules violation bugs. Gruska et al. [17] presented a tool to mine API usage rules across different projects. Similarly, some previous works [5, 12, 26, 54, 57] detect different types of bugs in a program through a mining approach to generate rules and detect violations. Different from these works, HERO does not rely on unknown-rule mining to detect bugs, thus it can support custom functions; instead, HERO takes advantage of the unique structures of the error-handling code.

10 Conclusion

Large programs such as OS kernels usually have complicated error-handling and code-cleanup mechanisms, which are buggy because they are less tested and hard to implement. Prior research attempted to detect the bugs, but mainly on the "handling" part instead of the cleanup mechanisms. This paper proposed DiEH bugs, a class of error-handling bugs that are caused by improper cleanup operations—incorrect-order, redundant, and inadequate cleanups. Through a study, we show that DiEH is hard to avoid and thus is prevalent; it also causes critical security problems such as memory corruption and privilege escalation. This paper then presented a new detection system, HERO. At its core is a precise function pairing technique that leverages the unique error-handling structures in low-level languages. We evaluate HERO on two OS kernels and the OpenSSL library. The results show that HERO can precisely identify a large number of function pairs including custom ones, and can detect 239 critical DiEH bugs, most of which were confirmed by maintainers. HERO is generic, and its precise pairing analysis can be applied to benefit other research such as race detection and temporal-rule inferences.

11 Acknowledgment

We thank our shepherd, Mathias Payer, and the anonymous reviewers for their helpful suggestions and comments. We are grateful to Linux maintainers for providing prompt feedback on patching the Linux kernel. The authors also thank Peng Le for helping implement function pairing. This research was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, 2007.

[2] B. Andersson. Linux kernel patch log, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6e5da6f7d82474e94c2d4a38cf9ca4edbb3e03a0.

[3] J.-J. Bai, H.-Q. Liu, Y.-P. Wang, and S.-M. Hu. Runtime checking for paired functions in device drivers. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 407–414. IEEE, 2014.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[5] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, 45(10):984–1001, 2018.

[6] Bootlin-Community. Linux kernel: kobject_init_and_add(), 2020. https://elixir.bootlin.com/linux/v5.7-rc7/source/lib/kobject.c#L464.

[7] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[8] M. Corporation. Common weakness enumeration (cwe), 2020. https://cwe.mitre.org/.

[9] M. Corporation. Owasp top ten 2004 category a9 - denial of service, 2020. https://cwe.mitre.org/data/definitions/730.html.

[10] M. Corporation. Cwe-200: Exposure of sensitive information to an unauthorized actor, 2020. https://cwe.mitre.org/data/definitions/200.html.

[11] M. Corporation. Cwe-416: Use after free, 2020. https://cwe.mitre.org/data/definitions/416.html.

[12] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*, 2018.

[13] D. DeFreez, H. M. Baldwin, C. Rubio-González, and A. V. Thakur. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 466–476, 2019.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[15] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82. IEEE, 2019.

[16] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[17] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130, 2010.

[18] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[19] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

[20] Y. Kang, B. Ray, and S. Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 472–482. ACM, 2016.

[21] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *2010 European Dependable Computing Conference*, pages 191–196. IEEE, 2010.

[22] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 43–52. IEEE, 2009.

[23] H. Liu, Y. Wang, L. Jiang, and S. Hu. Pf-miner: A new paired functions mining method for android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 33–42. IEEE, 2014.

[24] H.-Q. Liu, J.-J. Bai, Y.-P. Wang, Z. Bian, and S.-M. Hu. Pairminer: mining for paired functions in kernel extensions. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 93–101. IEEE, 2015.

[25] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[26] K. Lu, A. Pakki, and Q. Wu. Automatically identifying security checks for detecting kernel semantic bugs. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 3–25, Cham, 2019. Springer International Publishing.

[27] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786. USENIX Association, 2019.

[28] J. Mao, Y. Chen, Q. Xiao, and Y. Shi. Rid: finding refcount bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 531–544, 2016.

[29] C. Minyard and T. Hellstrom. Cve-2019-0685: A refcount leak vulnerability., 2004. https://sigpwn.io/blog/2020/5/7/cve-2019-0685-win32k-reference-count-leak.

[30] MITRE-CVE. A refcount leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2016-0728/.

[31] MITRE-CVE. A use-after-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15292/.

[32] MITRE-CVE. A double-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15504/.

[33] MITRE-CVE. A deadlock vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15538/.

[34] MITRE-CVE. A null dereference vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15923/.

[35] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-16994/.

[36] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5023.

[37] MITRE-CVE. A refcount leak vulnerability in the freebsd, 2019. https://www.cvedetails.com/cve/CVE-2019-5607/.

[38] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8980.

[39] MITRE-CVE. Cve-2019-0685, 2020. ttps://www.cvedetails.com/cve/CVE-2019-0685/.

[40] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15393.

[41] MITRE-CVE. Cvedetils, 2020. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[42] A. Pakki and K. Lu. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *27th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2020.

[43] L. project community. Llvm alias analysis infrastructure, 2020. https://llvm.org/docs/AliasAnalysis.html.

[44] H. A. Rosenberg and K. G. Shin. Software fault injection and its application in distributed systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 208–217. IEEE, 1993.

[45] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM Sigplan Notices*, volume 44, pages 270–280. ACM, 2009.

[46] R. Russell. What if I don't actually like my users?, Apr. 2008. https://ozlabs.org/~rusty/index.cgi/tech/2008-04-01.html.

[47] S. Saha, J. Lawall, and G. Muller. An approach to improving the structure of error-handling code in the linux kernel. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 41–50, 2011.

[48] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. 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)*, pages 1–12. IEEE, 2013.

[49] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[50] W. Tang. Identifying error code misuses in complex system. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 428–432, 2019.

[51] Thgarnie. Syzkaller, 2019. https://github.com/google/syzkaller.

[52] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*, pages 496–506. IEEE, 2009.

[53] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762. ACM, 2017.

[54] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[55] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[56] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[57] H. Yan, Y. Sui, S. Chen, and J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54, 2017.

[58] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[59] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. Apisan: Sanitizing {API} usages through semantic cross-checking. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 363–378, 2016.

[60] H. Zhang, D. She, and Z. Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1663–1674, 2016.

[61] S. Zhang, J. Zhu, A. Liu, W. Wang, C. Guo, and J. Xu. A novel memory leak classification for evaluating the applicability of static analysis tools. In *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 351–356. IEEE, 2018.

A Appendix

Program	File	Line#	Impact	Category
OpenSSL	crypto/x509/v3_crld.c	85	ML	D3
	crypto/cms/cms_sd.c	326	ML	D3
	crypto/store/loader_file.c	406	DF	D2
FreeBSD	lib/libkiconv/kiconv_sysctl.c	50	ML	D3
	lib/libkiconv/kiconv_sysctl.c	75	ML	D3

Table 6: DiEH bugs found in OpenSSL and FreeBSD. D1, D2, D3 denote incorrect-order, redundant, and inadequate DiEH bugs, respectively. Column "Line#" is the line number, and Column 4 indicates impact of bug. ML = memory leak, DF = double-free.

Buggy func name	Imp	Cat.	S	R
add_mdev_supported_type	RL	D1	A	
dmi_sysfs_register_handle	RL	D3	S	SIQ
kfd_topology_update_sysfs	RL	D3	S	IQ
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
fimc_md_register_sensor_entities	RL	D3	S	SIQ
NILFS_DEV_INT_GROUP_FNS	RL	D3	C	
power_supply_add_hwmon_sysfs	ML	D3	A	SIQ
intel_gtt_setup_scratch_page	ML	D3	A	IQ
nilfs_sysfs_create_snapshot_group	RL	D3	A	
acpi_cppc_processor_probe	RL	D3	A	SIQ
edac_device_register_sysfs_main_kobj	RL	D3	A	SIQ
netdev_queue_add_kobject	RL	D3	C	SIQ
nilfs_sysfs_create_snapshot_group	RL	D3	C	
bq24190_charger_get_property	RL	D3	S	SIQ
bq24190_charger_set_property	RL	D3	S	SIQ
bq24190_battery_get_property	RL	D3	S	SIQ
bq24190_battery_set_property	RL	D3	S	SIQ
stm32_mdma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dma_alloc_chan_resources	RL	D3	S	SIQ
tegra_adma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dmamux_route_allocate	RL	D3	S	IQ

Table 7: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).

Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R
aspeed_video_probe	ML	D3	C	SIQ	stm32f7_i2c_xfer	RL	D3	S	SIQ	rcar_pcie_probe	RL	D1	C	SIQ
nfp_abm_vnic_set_mac	ML	D3	A		stm32f7_i2c_reg_slave	RL	D3	S	SIQ	xcan_probe	RL	D1	S	SIQ
mlx4_opreq_action	-	D3	A		nv50_mstc_detect	RL	D3	C	SIQ	xcan_open	RL	D1	S	SIQ
rxkad_verify_response	ML	D3	A		nouveau_fbcon_open	RL	D3	S	SIQ	fec_enet_mdio_read	RL	D1	C	SIQ
siw_create_qp	ML	D3	C		nouveau_drm_ioctl	RL	D3	S	SIQ	macb_mdio_read	RL	D1	C	SIQ
cas_init_one	ML	D3	A	SIQ	radeon_drm_ioctl	RL	D3	C	SIQ	macb_mdio_write	RL	D1	C	SIQ
mlx4_opreq_action	ML	D3	A		radeon_crtc_set_config	RL	D3	C		omap4_keypad_probe	RL	D1	S	SIQ
add_port	ML	D3	A	SIQ	cdns_dsi_transfer	RL	D3	C	SIQ	mic_pre_enable	RL	D1	C	SIQ
img_i2s_in_probe	RL	D3	A	SIQ	v3d_get_param_ioctl	RL	D3	S	SI	img_spdif_in_probe	RL	D1	C	SIQ
iommu_group_alloc	RL	D3	A	SIQ	v3d_v3d_debugfs_ident	RL	D3	S	S	nouveau_drm_open	RL	D1	S	IQ
pblk_sysfs_init	RL	D3	C		v3d_measure_clock	RL	D3	S	S	radeon_dp_detect	RL	D1	S	
configfs_rmdir	RL	D3	C	SIQ	v3d_job_init	RL	D3	S		radeon_vga_detect	RL	D1	S	
f2fs_init_sysfs	RL	D3	C	SIQ	dss_runtime_get	RL	D3	S	SIQ	radeon_tv_detect	RL	D1	S	
f2fs_register_sysfs	RL	D3	C	S	dsi_runtime_get	RL	D3	S	SIQ	radeon_lvds_detect	RL	D1	S	
pci_create_slot	RL	D3	A	SIQ	venc_runtime_get	RL	D3	S	SIQ	bdisp_probe	RL	D1	S	SIQ
bond_sysfs_slave_add	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	bdisp_start_streaming	RL	D1	S	SIQ
iscsi_boot_create_kobj	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	hva_hw_probe	RL	D1	C	SIQ
rx_queue_add_kobject	RL	D3	C	SIQ	dispc_runtime_get	RL	D3	S	SIQ	hva_hw_get_ip_version	RL	D1	S	SIQ
img_spdif_out_probe	RL	D3	C	SIQ	clk_pm_runtime_get	RL	D3	C	SIQ	coda_open	RL	D1	C	SIQ
rvt_create_qp	ML	D3	A		musb_irq_work	RL	D3	C	SIQ	fimc_is_probe	RL	D1	C	SIQ
gfs2_create_inode	RL	D3	C	SIQ	usb_port_resume	RL	D3	S	SIQ	fimc_lite_open	RL	D1	S	SIQ
ath10k_sta_state	RL	D3	C	SIQ	ina3221_write_enable	RL	D3	S	S	dcmi_start_streaming	RL	D1	S	SIQ
ccp_run_sha_cmd	ML	D3	C		gpmi_nfc_exec_op	RL	D3	C	SIQ	s3c_camif_probe	RL	D1	C	SIQ
rockchip_pdm_resume	RL	D3	A	SIQ	bch_set_geometry	RL	D3	C	SIQ	img_i2s_in_probe	RL	D1	C	SIQ
tegra30_ahub_resume	RL	D3	A	SIQ	delta_get_sync	RL	D3	S	SIQ	venc_open	RL	D1	C	SIQ
tegra30_i2s_resume	RL	D3	A	SIQ	hva_hw_dump_regs	RL	D3	S	S	vfe_get	RL	D1	S	SIQ
img_i2s_out_set_fmt	RL	D3	C	I	stm32f7_i2c_reg_slave	RL	D3	S	SIQ	exynos_trng_probe	RL	D1	C	SIQ
img_i2c_xfer	RL	D3	C	SIQ	isp_video_open	RL	D3	C	SIQ	rvin_open	RL	D1	C	SIQ
configfs_rmdir	RL	D3	C	SIQ	s5pcsis_s_stream	RL	D3	S	SIQ	rvt_create_qp	ML	D1	A	
img_prl_out_set_fmt	RL	D3	A	I	fimc_capture_open	RL	D3	C	SIQ	rawsock_connect	RL	D1	S	SIQ
ethoc_probe	ML	D3	S	SIQ	vpe_runtime_get	RL	D3	S	SIQ	lpi2c_imx_master_enable	RL	D3	S	SIQ
img_i2s_out_probe	RL	D3	C	SIQ	xiic_xfer	RL	D3	S	SIQ	panfrost_job_hw_submit	RL	D3	S	SIQ
img_i2c_init	RL	D3	C	SIQ	s3c_camif_open	RL	D3	C	SIQ	vc4_dsi_encoder_enable	RL	D3	S	SIQ
img_i2c_xfer	RL	D3	C	SIQ	s5p_mfc_power_on	RL	D3	S	SIQ	vc4_vec_encoder_enable	RL	D3	S	SIQ
display_init_sysfs	RL	D3	A	SIQ	img_i2s_in_set_fmt	RL	D3	C	I	cpuidle_add_state_sysfs	RL	D3	A	IQ
bq24190_sysfs_show	RL	D3	S	SIQ	csid_set_power	RL	D3	C	SIQ	efivar_create_sysfs_entry	RL	D3	C	SIQ
bq24190_sysfs_store	RL	D3	S	S	ispif_set_power	RL	D3	C	SIQ	esre_create_sysfs_entry	RL	D3	A	SIQ
img_pwm_remove	RL	D3	S	SIQ	csiphy_set_power	RL	D3	S	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
img_pwm_config	RL	D3	S	SIQ	vsp1_probe	RL	D3	C	SIQ	dwc3_pci_resume_work	RL	D3	C	SIQ
ti_qspi_setup	RL	D3	C	SIQ	rcar_fcp_enable	RL	D3	S	SIQ	cdns_dsi_bridge_enable	RL	D3	C	SIQ
tegra_sflash_resume	RL	D3	C	SIQ	__vxlan_dev_create	ML	D3	C	Q	nfc_genl_llc_set_params	UAF	D2	C	IQ
tegra_spi_setup	RL	D3	S	SIQ	cpuidle_add_sysfs	RL	D3	A	SIQ	wlcore_regdomain_config	RL	D3	C	IQ
tegra_spi_resume	RL	D3	S	SIQ	fw_cfg_register_file	RL	D3	S	SIQ	radeon_driver_open_kms	RL	D3	C	
sprd_spi_remove	RL	D3	C	SIQ	edd_device_register	RL	D3	S	SIQ	nouveau_crtc_set_config	RL	D3	-	SIQ
tegra_slink_setup	RL	D3	C	SIQ	dmi_system_event_log	RL	D3	S	IQ	rga_buf_start_streaming	RL	D3	C	
tegra_slink_resume	RL	D3	C	SIQ	mc13xxx_rtc_probe	DU	D2	A	SIQ	s5p_jpeg_start_streaming	RL	D3	S	SIQ
img_spfi_resume	RL	D3	S	SIQ	m66592_probe	DF	D2	A	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
edma_probe	RL	D3	C	SIQ	cros_ec_ishtp_probe	DU	D2	S	SIQ	mtk_jpeg_start_streaming	RL	D3	S	SIQ
rcar_dmac_probe	RL	D3	S	SIQ	punch_hole	DU	D2	-	SIQ	stm32f7_i2c_unreg_slave	RL	D3	S	SIQ
sprd_dma_remove	RL	D3	S	SIQ	nfc_genl_llc_sdreq	UAF	D2	C	IQ	fimc_isp_subdev_s_power	RL	D3	S	SIQ
zpa2326_resume	RL	D3	C	SIQ	qcom_pcie_probe	-	D2	S	SIQ	nouveau_gem_object_del	RL	D3	S	
arizona_clk32k_enable	RL	D3	A	SIQ	s3c_camif_probe	-	D1	A	SIQ	panfrost_perfcnt_enable_locked	RL	D3	S	
gpio_rcar_request	RL	D3	C	SIQ	tegra_adma_probe	-	D1	C	SIQ	etnaviv_gpu_recover_hang	RL	D3	S	SIQ
arizona_gpio_get	RL	D3	A	SIQ	i915_gem_init	ML	D1	C	IQ	arizona_gpio_direction_out	RL	D3	A	SIQ
sata_rcar_resume	RL	D3	A	SIQ	pvrdma_pci_probe	-	D1	A	SIQ	vc4_hdmi_encoder_enable	RL	D3	S	SIQ
sata_rcar_restore	RL	D3	A	SIQ	qib_create_port_files	RL	D1	A		amdgpu_display_crtc_set_config	RL	D3	S	
cdns_pcie_host_probe	RL	D3	S	SIQ	add_port	RL	D1	A	SIQ	nouveau_connector_detect	RL	D3	S	SIQ
cdns_pcie_ep_probe	RL	D3	-	SIQ	i915_gem_init	ML	D1	-	IQ	nv50_disp_atomic_commit	RL	D3	C	SIQ
xcan_get_berr_counter	RL	D3	S	SIQ	test_hints_case	RL	D1	A	SIQ	edac_pci_main_kobj_setup	RL	D3	A	IQ
fec_enet_open	RL	D3	C	SIQ	gfs2_create_inode	RL	D1	C	SIQ	nouveau_gem_object_open	RL	D3	C	
fec_enet_mdio_write	RL	D3	C	SIQ	rocker_dma_rings_init	ML	D1	A	SIQ	nouveau_debugfs_pstate_set	RL	D3	C	SIQ
bma150_open	RL	D3	S	SIQ	tegra_spi_probe	RL	D1	S	SIQ	nouveau_debugfs_strap_peek	RL	D3	S	S
stmfts_input_open	RL	D3	S	IQ	tegra_slink_probe	RL	D1	C	SIQ	amdgpu_connector_dp_detect	RL	D1	S	
stm32f7_i2c_xfer	RL	D3	S	SIQ	tegra_adma_probe	RL	D1	C	SIQ	amdgpu_connector_vga_detect	RL	D1	S	
arizona_extcon_probe	RL	D3	C	SIQ	usb_dmac_probe	RL	D1	S	SIQ	amdgpu_connector_lvds_detect	RL	D1	S	
etnaviv_gpu_init	RL	D3	S	SIQ	sprd_dma_probe	RL	D1	S	SIQ	amdgpu_driver_open_kms	RL	D1	S	
etnaviv_gpu_debugfs	RL	D3	S		sata_rcar_probe	RL	D1	A	SIQ	tegra_vde_ioctl_decode_h264	RL	D1	C	SI
etnaviv_gpu_bind	RL	D3	S	SIQ	tegra_pcie_probe	RL	D1	S	SIQ	qlcnic_83xx_interrupt_test	ML	D1	A	I
vc4_v3d_pm_get	RL	D3	S	SI	qcom_pcie_probe	RL	D1	S	SIQ	acpi_sysfs_add_hotplug_profile	RL	D1	A	IQ
amdgpu_drm_ioctl	RL	D3	S	SIQ	dra7xx_pcie_probe	RL	D1	S	SIQ	nilfs_sysfs_create_device_group	RL	D1	C	S

Table 8: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).