Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection

Aditya Pakki, and Kangjie Lu
University of Minnesota

ABSTRACT

Operating system (OS) kernels frequently encounter various errors due to invalid internal states or external inputs. To ensure the security and reliability of OS kernels, developers propose a diverse set of mechanisms to conservatively capture and handle potential errors. Existing research has thus primarily focused on the *completeness* and *adequacy* of error handling to not miss the attention. However, we find that handling an error with an over-severe level (e.g., unnecessarily terminating the execution) instead hurts the security and reliability. In this case, the error-handling consequences are even worse than the error it attempts to resolve. We call such a case *Exaggerated Error Handling (EEH).* The security impacts of EEH bugs vary, including denial-of-service, data losses, broken control-flow integrity, memory leaks, etc. Despite its significance, detecting EEH remains an unexplored topic.

In this paper, we first conduct an in-depth study on EEH. Based on the findings of the study and rules generated via manual investigation, we then propose an approach, EeCatch, to detect EEH bugs in a context-aware manner. EeCatch accurately identifies errors and extracts their contexts (both spatial and temporal), and automatically infers the appropriate severity level for error handling. Using the inferred severity level, EeCatch finally detects EEH bugs in which the used error handling exceeds the inferred severity level. To demonstrate the effectiveness and scalability of EeCatch, we develop a prototype that uses inter-procedural, field- and context-sensitive static analysis. By analyzing the whole Linux kernel, EeCatch reports hundreds of potential EEH bugs that may cause security issues such as crashing the system. After evaluating 104 cases reported by EeCatch, we manually confirmed 64 EEH bugs and submitted patches for all of them. Using our patches, Linux maintainers have fixed 48 reported EEH bugs, confirming the effectiveness of EeCatch. To the best of our knowledge, we are the first to systematically study and detect EEH bugs. We hope the findings could raise the awareness of the critical consequences of EEH bugs to help developers avoid them.

CCS CONCEPTS

• **Security and privacy → Operating systems security**;

KEYWORDS

OS Kernel Bug, Exaggerated Error Handling, Static Analysis; Bug Detection

ACM Reference format:
Aditya Pakki, and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of CCS '20, Orlando, FL, USA, November 09-13, 2020,* 17 pages.
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

OS kernels form the bedrock for software applications and hardware, and they are expected to work correctly in both malicious and benign scenarios. When a system encounters an error—an invalid internal state or input, the error-handling (EH) mechanisms are often the first line of defense for ensuring the security and reliability of the system. For example, before a size variable obtained from the userspace is used for memory writes, the kernel typically uses a bound-check to capture an over-size error and handles it properly instead of continuing (a buffer overflow) or terminating (a denial-of-service) the execution. According to our study, the Linux kernel alone has more than 446K EH sites, showing the prevalence.

One can classify error handling into multiple levels based on severity. While less-severe errors require only warning or logging, more-severe errors may warrant the termination of whole-system execution. For example, in case of a failure in the `ioctl` system call for a USB driver, the appropriate EH is typically logging the error to the ring buffer, and the kernel continues its execution instead of terminating. By contrast, if the file system fails to mount during system boot time, the kernel would issue a `panic` call that hangs the system. This extreme EH is in fact necessary to avoid potential disk corruption or data loss. When an error is handled at a level lower than the desired severity level, the error is considered inadequately handled. On the contrary, if the impact of the EH is more severe than the actual consequences of the error, it unexpectedly introduces security and reliability issues such as unnecessarily crashing the kernel. We name such cases as *Exaggerated Error Handling* (EEH), a new class of semantic bugs. Figure 1 shows an example of an EEH bug that unnecessarily crashes the process via `BUG_ON` (an EH function) when the field `cx231xx_send_usb_command` is `NULL`. In this scenario, returning an error `-EINVAL` is sufficient as all the callers of `cx231xx_i2c_register` can handle this error safely.

According to our study, a majority of EEH bugs occur due to the following reasons: regression issues introduced by code fixes, subjective error severity estimation, assertion usage in production code, and inconsistent EH mechanisms in the call chain. In our study, we identified the top two reasons for the occurrence of EEH bugs (78%) as (1) incorrect reasoning about the severity of the error and (2) improper use of assertions in production code. Further, developers do not want to introduce unintended regressions caused

```
1  /* drivers/media/usb/cx231xx/cx231xx-i2c.c */
2  int cx231xx_i2c_register(struct cx231xx_i2c *bus) {
3      struct cx231xx *dev = bus->dev;
4  -   BUG_ON(!dev->cx231xx_send_usb_command);
5  +   if (!dev->cx231xx_send_usb_command)
6  +       return -EINVAL;
7      ...
8  }
9  /* the error is handled properly in callers in
10  * cx231xx-cards.c, without crashing the process */
```

Figure 1: A new EEH bug detected by EeCatch. It avoids a process crash when `cx231xx_send_usb_command` is `NULL` by replacing line 4 with lines 5 and 6.

by their fixes. Due to the challenges of testing the whole kernel, errors are often handled locally, rather than evaluating the global impact of their fixes.

EEH bugs may cause critical issues hurting security and reliability, such as denial-of-service, data losses, memory leaks, information leaks, inconsistent resource release, among others. More importantly, these bugs can be introduced anywhere within the system, and are often located in the less frequently executed code [62]. Linus Torvalds frequently advises the Linux contributors about the need to use appropriate EH mechanisms, rather than terminating a process [51, 52]. Unfortunately, EEH bugs are still common. In particular, we studied 168 crashes of the Linux kernel reported by Syzkaller [55]. We identified that at least 34 (20%) crashes are caused by EEH bugs.

Although EEH may cause critical security issues, to the best of our knowledge, EEH remains an unexplored topic. Due to the critical consequences of missing the handling of an error, prior research has primarily focused on the completeness [16, 23, 50] and adequacy [22, 45] of EH. Security researchers propose techniques to avoid failures by focusing heavily on the cause of errors and their patterns, often underestimating the security impact of EH code that is "beyond" fixing the errors.

Despite the significance of EEH bugs, detecting them suffers from the following challenges. First, even for the same error, EH can vary according to the error context. For example, a failure in heap memory allocation in the `arch` subsystem deserves a kernel crash, whereas in the `sound` subsystem, it is merely logged, without impacting the rest of the kernel. Therefore, effective detection of EEH must be context-aware. However, it is unclear what contexts are related to EH, and how they can be modeled and extracted. Second, determining the severity of the error is not straightforward. An EEH bug's EH is at a higher severity level compared to its expected level. There are no guidelines in the kernel on properly assigning the severity to either errors or its EH. To detect bugs, it is important to determine the severity of not only various errors with particular contexts but also EH. Third, EH is diverse and there is no single comprehensive list of EH techniques. However, EEH bug detection requires precise identification of EH along the call chain. Fourth, performing manual analysis to determine the severity of 446K EH is impractical. Such an analysis also requires determining the appropriate severity level of the fix. Thus, it is important to develop a reliable strategy that can identify EH. The technical challenges encountered in detecting EEH bugs are summarized as follows.

- **Identifying errors and EH** Deducing the appropriate error severity requires identifying errors and the corresponding EH. Unlike `exceptions` in object-oriented languages, there are no clear patterns to identify errors, their handlers, terminating functions, and wrappers of such functions in the kernel.
- **Understanding error contexts** The appropriate severity for EH highly depends on the contexts of errors. It is not clear yet what factors constitute the contexts for EH. It is also unclear how the contexts should be modeled and incorporated to determine the appropriate severity for an error.

In this paper, we first study the EEH problem in the Linux kernel. We collect 239 instances of EEH bugs via `Git`. We manually identify the errors, the EH mechanisms, and assign the severity level based on the consequences. We observed that 208 out of 239 (87%) instances of EEH are fixed by either eliminating existing EH or replacing them with a lower-severity EH. The remaining cases required extensive logic changes to fix the EH. By encoding the manually identified rules of our study, we develop EeCatch, a tool capable of detecting EEH bugs in the Linux kernel. EeCatch first automatically identifies errors and EH mechanisms. In the second step, EeCatch models a two-dimensional context of errors that contains appropriate spatial and temporal context information. Based on the identified errors and their contexts as well as associated EH mechanisms, EeCatch automatically infers the appropriate severity level for errors with particular contexts. By identifying the severity level of the deployed EH and comparing it with the inferred one, EeCatch detects EEH bugs. To ensure the detection precision, EeCatch incorporates inter-procedural, flow-, context-, and field-sensitive static analysis for the context-aware detection. EeCatch employs a probabilistic approach to bug detection [11]. The technique is vulnerable to false positives (§6.3), false negatives (§6.4), and requires manual effort to confirm bugs.

We implemented EeCatch using the LLVM infrastructure [26] and evaluated its effectiveness and scalability by applying it to the latest stable version of the Linux kernel. It finished the analysis for the whole kernel within 56 minutes and we found 104 potential EEH warnings. We identified 64 new EEH bugs and sent patches to the maintainers for all confirmed bugs. At the time of paper submission, Linux maintainers have fixed 48 bugs using our patches. These new bugs have critical security impacts. The majority of the detected bugs, 66% of 64 bugs (see Table 10), can crash the process, which is particularly critical for long-running servers. Two bugs can crash the kernel. EeCatch has a low false-negative rate in detecting previous EEH bugs; evaluation results show that it can detect 92.1% of the 239 previous EEH bugs (see §6). These results show that EeCatch is scalable and effective in detecting general EEH bugs in the Linux kernel.

We make the following research contributions in this paper.

- **First study of EEH.** We introduce, to the best of our knowledge, the concept of Exaggerated Error Handling, a new class of semantic bugs. We conduct the first in-depth study against EEH in the Linux kernel. The study covers the causes and impact of EEH bugs and categorizes EH based on the severity levels. The study provides an interesting and important step in detecting EEH bugs in OS kernels.

- **Context-aware detection of EEH.** We propose a novel approach that effectively detects EEH bugs in a context-aware manner. By modeling and extracting both spatial and temporal contexts, our detection can automatically infer the severity level for errors with particular contexts and uses that for detecting EEH bugs. To realize the context-aware detection, we also develop techniques to automatically detect errors in the kernel by tracking the error variables and to identify EH functions based on their patterns.
- **Open-source prototype and new bugs.** We develop a prototype for EeCatch, which will be open-sourced With EeCatch, we found 64 new EEH bugs in the latest version of the Linux kernel, which may cause critical security and performance issues. Most of the bugs were fixed with our patches. We hope that the findings and the prototype could help future developers avoid EEH bugs.

2 A STUDY OF EEH

In this section, we first provide background information about errors and EH in the Linux kernel. Then, we study a set of previously fixed EEH bugs we manually collected via the git history of the Linux kernel. In particular, we will describe the importance of contexts in determining the severity level of EEH bugs.

2.1 Errors and Error Handling in the Kernel

Errors, faults, and failures. For consistency, we use the definitions presented in the seminal paper by J.C. Laprie [25], and define errors as deviations from expected behaviors. Faults are the root cause behind an error, such as hardware failures or software bugs. Failures are the consequences of incorrect handling of the errors. In this paper, we will use errors interchangeably with error codes to indicate deviations. Error handlers are code pieces performing the EH. OS kernels and other critical systems are designed for availability and to minimize halting on failures.

Unlike other modern programming languages such as C++, Java, Python, etc.; C language does not provide any structured EH primitives such as `exceptions`. Due to such limitations, developers instead use the runtime indicators of errors, typically error codes, to capture and handle errors. As a programming convention, Linux developers negate the standard error codes and assign them to the `errno` variable to indicate the captured errors. Further, the error is typically propagated via the return value of a function to its callers. Unlike numeric error codes, a `NULL` pointer also captures an error for pointer variables. Correctly processing these errors is important to ensure that the system is not prone to crashes or exposing itself to new exploits.

Error-handling primitives. Previous researchers analyzing Linux kernel [16, 22, 23, 31, 50] modeled error returning (i.e., an error code propagates to its caller) as valid EH, Such cases account for 37% of all EH [22]. We believe that researchers adopted this strategy to avoid tracking complicated data flow. However, error returning is not an actual EH because the caller function that takes the returned value will have to eventually handle the error. In our model, we classify valid EH into the following three categories.

- **Terminating Execution.** Fatal errors such as the ones causing data loss or damage to the system are of the highest severity level.

```
1  /* drivers/tty/tty_ldisc.c - Apr 16, 2017*/
2  static void tty_ldisc_restore(struct
3          tty_struct *tty, struct tty_ldisc *old)
4  {
5  -   old = tty_ldisc_get(tty, old->ops->num);
6  -   WARN_ON(IS_ERR(old));
7  -   tty->ldisc = old;
8      /* Null pointer dereference */
9  -   tty_set_termios_ldisc(tty, old->ops->num);
10
11 +   if (tty_ldisc_failto(tty, old->ops->num) < 0) {
12 +       const char *name = tty_name(tty);
13 +       pr_warn("Falling back ldisc for %s.\n", name);
14      ...
15 }
```

Figure 2: Commit# 598c2d41f fixing a potential EEH bug. It causes a process crash (in line 9) or even machine crash [18] (in line 6) when the `kernel.panic_on_warn` value is either zero or non zero, respectively.

  The corresponding EH would either terminate the machine or crash the process execution by calling either `BUG` (see Figure 1) or `panic` in the kernel. Depending on the specific configuration, `assertion` failures typically crash the process as well. We model the severity difference between a process crash and a kernel crash but use terminating functions to describe both scenarios.
- **Logging errors.** Logging and monitoring for important errors is a common EH strategy. Depending on the severity of the error, various logging levels (`kern_levels.h`) are used to ensure appropriate response and escalation. The Linux kernel uses seven levels of error severity (see Table 2) to log errors. Unfortunately, various subsystems customize the logging functions to suit their demands, and introduce new logging functions, within a limited scope.
- **Ignoring errors.** Contrary to other application software, the kernel code frequently handles potential errors, without compromising the correctness, by ignoring them. There are three reasons often stated for such behavior. First, the kernel previously validated a scenario to ensure subsequent failure-free execution. For example, checking for NULL from a `kmalloc()` call using `__GFP_NOFAIL` is unnecessary, as the memory allocation does not fail. Second, capturing and handling every possible error can add performance overhead by generating branches (unnecessary speculative execution). Third, an erroneous scenario may not be critical at all to impact the execution. For example, developers ignore potential errors while unregistering a device driver [16]. Within the scope of this paper, we treat functions ignoring errors as a valid EH strategy, as long as it does not introduce issues.

Factors determining severity level. The severity levels indicate the relative seriousness of the error's occurrence and the potential consequences of not handling at the said level. For example, EEH bugs can occur when an error is handled at a higher severity level than its intended level. Based on our EEH bug dataset, we identified that the appropriate severity level of the EH depends on the error and its context. For each of the 239 bugs, we collect the existing EH, updated EH, and the errors. We determined the updated EH of 218 bugs is consistent with the other EH within the function or the module. The remaining 21 bugs required major code changes (12 out of 21) impacting EH across modules or incorrect regression fixes (9 out of 21). These observations suggest that severity depends not only on the error but also the error-context.

Level	Error-handlers	EH Category	Examples
0	Crashing kernel	Terminate Execution	panic
1	Terminate process	Terminate Execution	BUG
2	Severe errors	Logging errors	WARN, pr_[emerg,alert,crit]
3	Important errors	Logging errors	pr_err
4	Warnings	Logging errors	pr_warn
5	Non critical errors	Logging errors	pr_info, pr_dbg, pr_notice
6	No EH	Ignoring errors	No visible indicators

Table 1: The assigned severity levels to errors in EeCatch based on their impacts, and corresponding EH mechanisms at each severity level.

Assigning severity levels for errors and EH. Based on the EH primitives discussed earlier, we attempt to assign the severity level based on our empirical observations of identified EEH bugs, and previous work by other researchers [27]. Table 1 shows the modeled severity levels and the corresponding EH mechanisms. We borrow the idea of classification used in the logging levels within the kernel (`kern_level.h`) to determine relative severity between various errors. The seven error logging levels used in the kernel are emerg, crit, alert, err, warn, notice, info, and, debug.

In our study, we observe that developers use subjective preferences to assign severity of errors. Among the severity levels in Table 2, we combine critical, alert, and emergency logging levels to indicate severe errors. Similarly, debugging, information, and notice levels represent non-severe errors. Finally, we model a kernel crash as the most severe error and ignoring error as the least severe category. `WARN` and `BUG_ON` can crash the Linux kernel by setting `panic_on_warn` and `panic_on_oops` kernel sysctl, respectively. Besides Android OS, most Linux configurations do not enable either parameter. Thus we model `WARN`, `BUG_ON`, and `panic` at different severities, based on their impact.

2.2 Exaggerated Error Handling in the Kernel

We study existing EEH bugs to find insights into the effective detection of EEH bugs. In this part, we briefly describe our collection of previous EEH bugs, the composition of the dataset, and our insights from the study.

Feature	Details
Errors	NULL, standard error codes such as -ENOMEM.
Severity levels	7 Error-logging levels found in `kern_level.h`.
Log functions	printk, pr_X, dev_X; X is one of the severity levels.
Terminate execution	panic(), BUG(), WARN*

Table 2: Initial set of known errors, EH, and severity levels in the Linux kernel. * In Android OS, as `panic_on_warn` and `panic_on_oops` are enabled, WARN and BUG_ON cause a panic.

Dataset collection for previous EEH bugs. Due to the paucity of previous research identifying EEH bugs, we rely on a pattern-based search to identify Linux kernel commits, involving the said bugs. We searched the git log of the kernel, using *grep* for commits that contained one of "reduce", "remove", or "replace", along with the terminating or logging functions listed in Table 2 which is empirically prepared for OS kernels. By studying the commit messages and the code changes, we identify the error, the original EH mechanism, the impact of new EH mechanism, and their severities. We collected patches submitted between Jan 2016 and Nov 2019 and identified 86 commits involving EEH bugs, containing 343 valid exaggerated error-handling changes (a single commit may fix several EEH bugs). After eliminating bugs that can be categorized into other common semantic bug categories, such as buffer overflow or memory corruption, we observed that 29 commits were identified by Syzkaller. We infer that the remaining commits are fixed via manual analysis.

Insights of study. We present interesting insights about EEH bugs in this section. ❶ EEH bugs are present across all subsystems within the kernel. However, due to the code size and quality, they are over-represented in the `drivers` subsystem, followed by `fs`. ❷ `BUG()` is the most common erroneous EH in exaggerated error-handling bugs. Maintainers are always on the lookout for new additions of `BUG()`. However, numerous wrappers of `BUG` exist and are added to the active code. ❸ Syzkaller, a fuzzing tool for Linux kernel, enables `panic_on_warn` and causes a panic when it encounters a `WARN()`. In the commits identified manually, there are no instances of EEH bugs involving `WARN_ON`. ❹ Incorrect error logging occurs infrequently. However, 24% of EEH bug fixes use logging as an EH technique. ❺ The top three causes for EEH bugs include, using assertions in production code (38.2%), liberal use of `BUG` and `WARN` wrappers (29.7%), and failure to track the impact across other callers (8.1%). Further reasoning shows that the occurrence of EEH bugs is mainly because (1) developers are unaware of the EEH issues and (2) it is hard for developers to induce the proper EH due to the complexity of kernel code.

Impacts of EEH bugs. Based on our study of the dataset, we determined the security impact of the identified EEH bugs Table 3.

- Crashing a machine can cause denial-of-service (DoS) exploits. We observed that a majority of the commits (62.1%) lead to potential DoS. As Syzkaller reports a crash caused by `panic`, `BUG_ON`, and `WARN_ON` as fatal, we manually identified the impact of 32 instances of bugs detected by it. 12.5% of the bugs detected by Syzkaller can cause DoS without enabling either `panic_on_warn` or `panic_on_oops`. 59% of 32 instances cause memory corruption, and the remaining do not have a security impact.
- A process crash within the file system avoids data corruption. However, 22.7% of the patches can cause data corruption, based on the resources they acquired before a crash.
- 15.2% of the commits are also vulnerable to memory leaks caused by crashes. A memory leak can occur upon encountering `BUG_ON`. Since the acquired resources are not freed automatically, we treat such cases as memory leaks. We exclude cases involving `panic`, as a machine crash is severe compared to memory leaks.
- According to the 2017 OWASP vulnerability study [39], inaccurate logging of the errors is one of the top 10 vulnerabilities. Most exploits start with vulnerability probing, and incorrect logging levels can miss the detection of these vulnerabilities. Our dataset contains 8.9% of the commits categorized as inaccurate logging.
- Other consequences of inaccurate monitoring can cause information leak of kernel pointers, and potentially introducing an exploitable vulnerability.

DoS	Incorrect logging	Data corrupt.	Memory leak	Information Leak
62.1%	8.9%	22.7%	15.2%	15.8%

Table 3: Common security impacts of EEH bugs.

```
1  /* File: net/netlink/af_netlink.c */
2  static void __init netlink_add_usersock_entry(void) {
3      listeners = kzalloc(sizeof(*listeners) ..., GFP_KERNEL);
4      if (!listeners)
5          panic("netlink_add_usersock_entry: Cannot allocate listeners\n");
6      ...
7  }
8  struct sock * __netlink_kernel_create(...) {
9      listeners = kzalloc(sizeof(*listeners) ..., GFP_KERNEL);
10     if (!listeners)
11      return NULL;
12     ...
13 }
14 void netlink_kernel_release(struct sock *sk) {
15     if (sk == NULL || sk->sk_socket == NULL)
16         return;
17 }
```

Figure 3: A simplified example showing diverse EH explained using temporal context.

The Linux community's view of EEH. A bug capable of crashing the kernel is critical (CVSS greater than 8.0 out of 10), as the vulnerabilities are susceptible to DoS exploits. The Linux creator, Linus Torvalds, repeatedly emphasizes the need for appropriate EH mechanisms, rather than crashing the kernel [51, 52]. From a developers viewpoint, kernel maintainers reject patches that use `BUG` as an EH strategy. Despite the widespread community measures to avoid crashes, caused by incorrect EH, there are over 13K instances of `BUG` and its wrappers, at the time of writing the paper. Therefore, it is important to not only study EEH bugs, but also develop techniques capable of precisely identifying these bugs in the kernel.

2.3 Contexts of Errors

After analyzing the EEH bugs from the dataset in §2.2, we observed that, in addition to the error itself, its context frequently determines the appropriate severity level of EH.

Modeling the life-cycle. We found that the severity of an error depends on the life-cycle phase (i.e., temporal context) the error is in. We use Figure 3 to show how the temporal context determines the severity. Two functions within the file `af_netlink.c` attempt to allocate the same size memory using `kzalloc`. In case of failure, `netlink_add_usersock_entry` crashes the kernel (line 5), whereas `__netlink_kernel_create` returns NULL (line 12) to its wrapper `netlink_kernel_create`. By contrast, during the finalization life-cycle phase, when the `kernel_kernel_release` attempts to release memory and fails, it ignores the error (line 16). Therefore, the EH for the same error is different in terms of severity level for different life-cycle phases. We generate a rule based on these observations that, depending on the life-cycle phase of the function with an error, the corresponding EH can differ in terms of severity, within the same module. By applying this rule to the dataset in §2.2, we narrow the possible EH techniques in both initialization and finalization phases. We mark the functions not belonging to either category as regular functions.

Modeling the subsystem. In addition to the life-cycle phases, we further found that the subsystem an error is located in also influences the severity of the error, which we refer to as spatial context. The kernel groups the major subsystems according to their functionality. Besides the standard error codes, each subsystem uses its own EH techniques. To generalize the EH and severities across subsystems, we extract the spatial context from the EEH bug dataset. Despite their functional similarities, `drivers` and `sound` subsystems, use different EH. Excluding the bugs detected by Syzkaller, EEH bugs in `sound` do not crash the kernel, whereas 24 out of 28 patches in `drivers` terminate the process. In fact, we found 3 instances in the entire `sound` subsystem capable of a process crash but 15K instances in `drivers`. The corresponding error density is one instance of terminating function per 304K and 840 lines, respectively. We conclude that not all code in the kernel is critical. It is dependent on the subsystem containing the error. We present a more detailed evaluation of the spatial context in Table 4.

To effectively capture the diverse error handlers, we model the error-context to contain both local factors involving the subsystem, and the global factors determined by the life-cycle. We evaluate the importance of each factor within the context and empirically determine the granularity based on our results.

3 EECATCH OVERVIEW

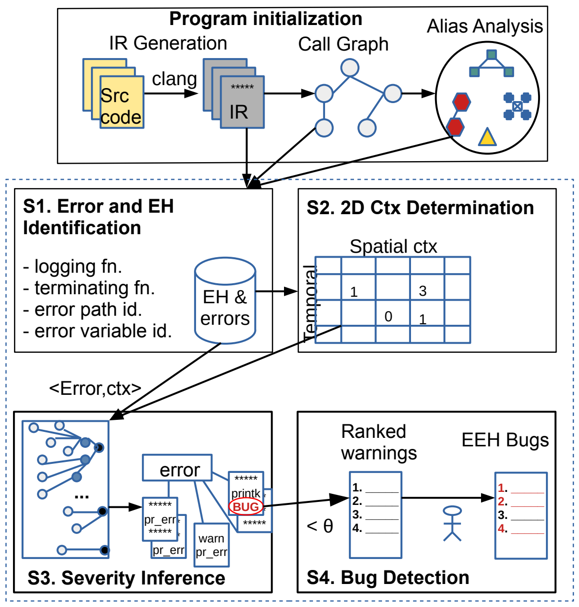

Figure 4: An overview of EeCatch.

While the first part of the paper presents the study of EEH, in the second part, we will present EeCatch, a tool that uses the findings of the study to effectively find EEH bugs in a context-aware manner. As shown in Figure 4, EeCatch identifies EEH bugs with the following steps.

Automatic identification of errors and error handlers. Defining comprehensive specifications for errors and error handlers in the Linux kernel is a challenge. EeCatch attempts to detect the errors and models different error handlers using static analysis and pattern-based detection. EeCatch identifies errors through their representations—error variables, which employs backward data-flow analysis. To identify error handlers; using an initial set of error handlers (see Table 2), EeCatch automatically expands the list via association mining. It also collects wrappers of standard terminating functions (`BUG` and `panic`) by tracking `unreachable` instructions and functions with `__noreturn` attribute. In all, EeCatch automatically identifies 62 wrappers of terminating functions, and 643 functions that log errors.

Extracting two-dimensional error contexts. From our study (§2.3) of the EEH bugs, we observed that same errors are often handled differently given different spatial and temporal contexts. As such, EeCatch generates a context-aware model for determining the severity level of an error. Our model uses both spatial and temporal factors of the function encompassing the error. The spatial context consists of subsystem (e.g., subsystems `sound` and `filesystem`) information, whereas a temporal context uses the life-cycle phase (e.g., phases `initialization` and `finalization`) information. In the absence of a probability distribution model, EeCatch determines the granularity and importance of these contexts, empirically, as explained in §2.3.

Inferring the severity level for <error, contexts>. The goal of this step is to automatically infer the appropriate severity level for errors with particular contexts (i.e., <error, contexts>). Using static analysis to determine the proper EH for each error in the kernel, is error-prone and infeasible. Instead, EeCatch uses statistical analysis to infer the severity of <error, contexts>. The idea is to first collect peer cases that share the same error and contexts as the one in question. Then, by statistically analyzing in which severity level such an error is commonly handled, EeCatch infers the appropriate severity level.

Detecting EEH bugs in a context-aware manner. Given a user-defined hyperparameter θ, an error, its peer set of EH, and the inferred severity level of the error, EeCatch reports deviations as warnings. If the severity level of the actual EH is higher than the inferred one, EeCatch reports it as an EEH case. EeCatch prunes the generated list of warnings and ranks them according to their likelihood of being an EEH bug. At last, we confirm the reported EEH warnings and fix them manually by reducing the severity level of the EH to the suggested level, without introducing security issues. EeCatch requires manual effort to confirm EEH bugs and to guarantee the soundness of the fix. We describe the sensitivity analysis of θ in §6 and accuracy of EeCatch in §6.3 and §6.4.

4 METHODOLOGY

In this section, we describe the major components of EeCatch, including accurate extraction of errors, identifying EH mechanisms, modeling the spatial and temporal contexts, inferring the severity of errors in specific contexts, and detecting EEH cases.

4.1 Identifying Errors

The first step in detecting EEH bugs is the identification of errors.

Representation of errors. Errors are not only numerous but also take diverse forms in OS kernels. As such, it is challenging to generally and automatically identify the actual errors. Fortunately, we observe that for a handled error (since EeCatch aims to detect EEH, we focus on handled errors), a corresponding *error variable* is used to indicate and capture the error. A common pattern is the use of standard error codes, e.g., `EINVAL`, to represent different errors.

Identifying error variables. Since handled errors are always represented by error variables, the key step of identifying errors is to precisely identify error variables, which is still a challenge for a number of reasons. First, OS kernel code is huge and the number of variables is often in the order of millions. Static analysis of these variables is not scalable and error-prone. Second, error variables often propagate across modules, involving complicated data flow. Third, identifying all possible errors is impossible, as there are no predefined error sets. Developers often add custom error codes. To minimize the impact of these challenges, we leverage the programming convention of C on what constitutes an error. The kernel defines a set of standard error codes in linux/include/errno.h to represent a majority of erroneous scenarios. Further, `NULL` pointers as return values are also treated as errors in C.

```
 1  /* File: net/dccp/ccid.c */
 2  struct ccid* ccid_new(const u8 id, struct sock *sk, bool rx) {
 3      if (ccid_by_number(id) == NULL)
 4          return NULL;
 5      ...
 6  }
 7  /* Caller: net/dccp/feat.c */
 8  int dccp_hdlr_ccid(struct sock *sk, u64 ccid, bool rx) {
 9      struct ccid *new_ccid = ccid_new(ccid, sk, rx);
10      if (new_ccid == NULL)
11          return -ENOMEM;
12      ...
13      return 0;
14  }
```

Figure 5: A simplified example to show how EeCatch tracks errors. In this example, `ccid_new` returns NULL to the caller `dccp_hdlr_ccid`. The latter propagates a different error code `-ENOMEM` to its callers.

Given an error variable V_e containing one of the possible values for an error, the problem is now transformed to identifying the seed of V_e. Identifying the seed is necessary to avoid analyzing the same error code, tainting multiple variables. The key idea to identify seeds relies on the observation - in the presence of an error, the control-flow uses a conditional statement to distinguish the normal path and the error (or EH) path. Starting from the error variable V_e used in the conditional statement, we track the seed of V_e within a function, by performing an intra-procedural backward data-flow analysis. We terminate tracking a variable when the analysis reaches either a function callsite or a local variable. On reaching a function callsite, EeCatch queries if a function can return one or more errors to its callers. If a function is not yet evaluated, EeCatch recursively performs the same data-flow analysis, starting from the return instruction of the function. EeCatch memoizes all errors returned by a function to its callers, at a function level.

To illustrate the above technique, we use the example shown in Figure 5. The function `dccp_hdlr_ccid` in line `9` captures the error (`NULL`) from the return value of `ccid_new`. The error path in line `11` returns another error (`-ENOMEM`) to its callers, while the regular path returns `0` in line `13`. EeCatch performs a dataflow analysis, identifies line 10, and assigns the seed of error, `NULL`, to the callsite of `ccid_new`.

4.2 Identifying and Classifying Error Handling

It is important to identify EH as it marks the end of an error's lifetime. Detecting EH is necessary to accurately reason about an error and its impact. Within the kernel, EH is not only unstructured but also numerous. Classifying the EH can avoid reasoning about each function individually, and help study the properties of each category as a whole. Unlike error codes which can be constrained to a smaller subset, EeCatch requires as complete EH set as possible, to minimize false negatives. An error within the subset can be handled by a wide range of error handlers taking diverse forms. Thus, a major component of EeCatch identifies EH and assigns each handler a severity. The key insight in identifying EH code in the kernel requires to reliably track the error path of a conditional statement capturing errors. By performing an inter-procedural, forward data-flow analysis along the error path, EeCatch collects various EH for an error. One thing to note is that, unlike previous research tools, the error path does not terminate if the error code is returned to the caller. As such, we track each caller recursively until the error variable is either overwritten or encounters an EH. As mentioned in the study, EH techniques are of three types, terminating functions, error logging functions, and ignoring the error. Starting from the initial set (Table 2), and using an inter-procedural data-flow analysis, we describe how we collect other EH.

Collecting terminating functions. On encountering a fatal error, the expected EH mechanism involves terminating the execution. Functions `panic` and `BUG` are well known to developers to crash the machine and kernel process, respectively. However, we do not have an exhaustive list of terminating functions. Therefore, we first collect basic terminating functions and then collect wrappers of them to augment the list. Specifically, `BUG` and `panic` are the two basic terminating functions. The functions use `unreachable()` to notify the compiler about potential abnormal execution. To find wrappers of such functions, we scan for unreachable instructions, `unreachable()` in the kernel, and use backward slicing to detect the functions causing a crash. Moreover, we check for the functions marked with `noreturn` attributes. This attribute indicates a function has no further scope and does not unwind the call stack. We mark functions in either of these categories as terminating execution. With all the analyses, we collect 62 EH functions that can terminate the execution.

Association mining for logging functions. Besides terminating execution, the most common EH strategy is error logging. In our study (see Table 4), we identified 248K conditional statements that use error logging as its EH. Yet, not all errors logged are of the same importance. The majority of these error logging functions handle errors that do not cause security impact. However, other logging functions are important in identifying potential vulnerability probing [39] and to triage fatal crashes [62]. It is important to comprehensively detect error handlers that can handle both severe and non-severe errors. Besides using patterns to identify new logging functions as described by researchers in [31], we employ association rule mining [7] to extract a comprehensive set of error handlers limited to logging the errors. In the previous works [22, 31], the authors use a pattern-based approach to detect logging functions. Such a technique cannot detect macros or the numerous logging functions without a fixed pattern.

Association rule learning is capable of generating strongly correlated rules among frequently occurring events, in large codebases. We attempt to learn new EH performing error logging, not previously identified by other tools. We rely on the insight that an error path of a conditional statement contains error handlers at a higher frequency, compared to its regular path. Given a conditional statement validating an error, learning new EH via association rule mining works as follows.

First, we determine the error path based on the predicate of the conditional statement. An error path is often the path where the predicate evaluates to NULL or less than zero. We mark the other path as the normal execution path. Second, for both error and regular paths, we perform an intra-procedural analysis identifying the logging statements. Third, a logging function maps to the source code via debugging information. Further, we extract the function name via regular expression search. Fourth, we count the number of occurrences of each error-handler in the error path and normal path, over the entire kernel. To minimize false positives in detecting EEH bugs, we use `0.9` as the *confidence* and *support* as 4 or more instances.

4.2.1 *Classifying Severity Levels for EH.*

It is important to classify EH according to their severity, as terminating the program on encountering a fatal error may be the best solution, to avoid data loss or machine damage. In this paper, we assign an integer number to a severity level from 0 (highest severe) to 6 (least severe) as seen in Table 1. We made the design for two major reasons. First, Linux kernel and most production quality software use numeric severity levels to distinguish among log levels. Our design extends this model to accommodate terminating functions (`BUG`, `panic`) and no EH. Second, using numerical severity levels can establish a total ordering over EH strategies. For example, an EH at level 0 strictly dominates an EH at level 4, or various EH at level 3 are all of the same severity.

Unfortunately, manually modeling each function is not only error-prone but also infeasible. Instead, we model error handlers by categorizing them based on impact and assigning a non-negative value. In this paper, we treat error handlers with lower severity values to produce more severe consequences. We model terminating functions as level 0 and the no EH at the highest value 6. More comprehensive reasoning for other severity levels is described in §2.1. EeCatch automatically extracts the severity of logging functions captured via its argument or its name. For example `printk(KERN_CRIT)` and `pr_crit()` log errors with a severity level `crit`.

Modeling the severity of error handlers must account for potential outliers. For example, logging functions are susceptible to subjective usage; developers might use `pr_crit` instead of `pr_emerg` for their logging. Relying on a name-based strategy might introduce inaccuracies within the model. To mitigate the impact we classify EH according to their intent and impact, and then assign a severity level. We estimate programmer intent based on the type of EH (e.g., missing error handlers indicate a low priority error). The impact is determined by the security consequences for the said error. Another notable EH function - WARN can crash the Android kernel but is not considered critical in other Linux configurations. Accordingly, we model WARN as a non-fatal error.

4.3 Analyzing and Modeling Error Contexts

After augmenting the initial knowledge set with new error handlers, EeCatch attempts to generate a context for the error and its handlers. A context-sensitive approach to EH improves the accuracy of EeCatch. This approach is effective in determining the appropriate EH for each error, in the kernel. On one hand, it is impossible to enumerate all possible factors affecting the error-context, and determining the error handlers. On the other hand, a context-aware and flow-sensitive, interprocedural dataflow analysis, generates a long call-chain from the seed of the error to EH. Such a strategy can reduce the *sensitivity* of the tool. Instead, we extract the context determined as the function containing the conditional statement, to minimize the impact of these two challenges.

4.3.1 Spatial Context. We observe that in which severity level an error should be handled highly depends on where the error occurs. For example, over 15K errors in `drivers` are handled via crashing the process or system. Yet, the `sound` subsystem has 3 instances that can terminate the process. Such a diversity in the EH indicates the importance of modeling the subsystem as context. We refer to the name of the subsystem, where the conditional statement identifies an error, as the error's *spatial context.* A key question in extracting spatial context is determining the appropriate granularity. In the absence of an underlying data distribution model, we rely on our observations to determine the granularity. We perform a study to identify the errors handled exclusively via logging (see Table 4). For the severe errors, we observed that the larger subsystems - `drivers`, `fs`, `arch`, and `net` have a higher number of errors per severity level. Thus, refining the spatial context by one more level, for these four subsystems, will not impact the precision.

On the other hand, a division treating every module (entire file name) as its own context would generate a smaller set of error handlers with fine-grained spatial context and rules with low *confidence*. EeCatch's design uses statistical analysis to detect exaggerated error-handling and requires a large number of error handlers to generate meaningful rules. Using a function or a module as a subsystem would generate a smaller set of error handlers, thereby a high false-positive rate.

4.3.2 Temporal Context. In our study, we further observed that, besides the spatial context, errors are often handled differently based on the life-cycle phase of the corresponding code, as described in §2.3. For example, memory-allocation failures in a function in the `initialization` phase are an indication of serious errors in the `mm` subsystem and require a kernel crash. However, other EH of memory allocation failure in other phases can be either ignored (example using `__GFP_ATOMIC`) or retried multiple times (see `fs`). Therefore, phases also influence the severity of errors. We refer to the life-cycle phase, the error is located in, as *temporal context.* Based on our observations, we categorize temporal context into three categories; initialization, finalization, and regular functions. We chose the above categories to minimize the false positives, and are based on our empirical observation from the dataset. The temporal context can be further augmented based on function similarity [2, 10] or name-based function matching [41]. These tools also use static analysis techniques but are prone to false positives. Thus we did not use them in our current design and instead rely on information captured by various life-cycle phases.

Initialization Functions. During boot time via `initcall`, the kernel starts allocating resources for successful operation in its later stages. Resources allocated before initialization include filesystem, core kernel, various subsystems, architecture-specific functions, privileged(root) functions, timer in `drivers`, etc. Once the kernel is initialized, these functions are removed to conserve kernel memory and to avoid inadvertent write operations. Faults in this context are dangerous to the correct working of the system. The typical error handlers in this context crash the machine, via `panic`, to enable faster recovery. To determine the function set in this context, we perform a forward control-flow traversal from the `initcall`, along the callsites in these functions. We also observed that all the functions marked with the `__init` section "attribute", belong to the initialization phase.

Finalization Functions. On the other hand, the errors occurring in the exit functions are considered finalization functions. Previous research works studying the error code propagation have observed that errors in the finalization context of driver code [16] can be ignored as the driver and kernel functionality is not negatively impacted. Similar to the initialization function, we instead look for the `__exit` section for each function to determine these finalization functions. Further, we track the callees from these functions and treat them as finalization functions.

One potential improvement of modeling temporal context involves grouping `probe` and `register` functions within the *initialization* phase set. Similarly, `remove` and `unregister` functions belong to the *finalization* set.

4.3.3 Chain construction from errors to error handlers. We have identified the set of errors and various EH strategies. Further, EeCatch also stores a set of errors, a function can return to its callsite. The goal of this step is to generate a set of peer error handlers per error, within a specific context. We can represent the factors as a tuple T containing four elements - the error, its context, a peer set of error handlers, and inferred severities of error handlers. EeCatch performs a backward data-flow analysis from each error until the analysis encounters one of the following in its path - a terminating function, a logging function, a caller with void type, or if the error is either lost or overwritten in the caller. The last two cases are an indication of error being handled without emitting any severity indicators. On encountering an error-handler, the severity is inferred via the severity level, and it is accounted for in the peer set. To avoid state explosion by traversing the same path multiple times, we memoize the errors and error handlers per function.

4.4 Statistical Analysis for EEH Bug Detection

The final step of EeCatch is performing statistical analysis via cross-checking [54] to identify potential EEH bugs. To avoid analyzing the semantics of the code and minimize false positives in static analysis, cross-checking can determine deviations from the majority. We define Component Severity Threshold (CST) of an error, as the ratio of error handlers at a severity level, (deviating from the majority) to the total number of error handlers within the peer set. For effective statistical analysis, it is important to generate reasonably large peer error handlers set, per tuple.

For an error occurring in an error-context, array V of size n contains the occurrences of EH at each severity level. V_i indicates the frequency of EH at a severity level i. θ is a user-defined hyper-parameter, threshold, such that $0 < \theta \leq 1$. We detect an EEH bug if EH at level j satisfies the following three conditions - Equation 1, Equation 2, and Equation 3.

- There is a majority EH within V. The maximum EH severity level m in the peer set occurs when

$$V_m > \sum_{i=0}^{n, i \neq m} V_i \tag{1}$$

- The majority EH CST_m is greater than or equal to $\overline{\theta}$.

$$\frac{V_m}{\sum_{i=0}^{n} V_i} \geq 1 - \theta \tag{2}$$

- The exaggerated severity level j must be higher than m. (A numerically smaller value for j indicates a more severe error.)

$$0 \leq j < m \tag{3}$$

Post evaluating all errors, the output of EeCatch is a list of warnings, i.e., potential EEH bugs. EeCatch employs a probabilistic approach to bug detection and requires manual effort to differentiate bugs from false positives. We employ ranking and pruning strategies to generate a smaller subset for human analysts. For each error, the tool infers exaggerated error-handling by identifying elevated error handlers, among the peers of error handlers.

5 IMPLEMENTATION

We implemented EeCatch as a set of LLVM (of version 10.0.0) passes, including passes for errors and error handlers identification via association mining, context analysis and modeling, and exaggerated error-handling bug detection and report generation. As the accuracy of a static analysis tool relies on a precise control flow graph, we use the call-graph (including indirect calls) generated by Crix [32] and alias analysis results generated by LLVM. We present other interesting implementation details within the system.

Compiling kernel source code. To generate the IR bitcode files from the Linux source code, we use `-O2` optimization, enable debugging (`-g`), and disable inlining. LLVM's Alias Analysis infrastructure introduces false positives in the `MayAlias` category with O0, and the O2 optimization can improve the points-to results. To ensure precision while not missing a significant number of aliases, EeCatch adopts the `MustAlias` results provided by LLVM as the baseline. It further extends the alias set through context- and field-sensitive data-flow analysis of `MayAlias` result set. EEH requires the identification of callers handling the error. Inlining replaces the call-sites with function code and can introduce false negatives in determining the error-context, so we disable inlining.

5.1 Collecting Error-Handling Functions

Terminating functions. The accuracy of EeCatch relies on having an exhaustive list of error handlers and their severities. To supplement the widely known terminating functions, `BUG` and `panic`; we rely on LLVM's `UnreachableInst` that models `unreachable()`, to detect wrappers of these functions. We search each function for this instruction and then determine potential successors by tracking its control-flow. If the control-flow terminates, we extract the function and mark them as terminating functions. To avoid false positives, we count the occurrences across the kernel and eliminate the functions with a single instance. Using this strategy, we collected 62 new terminating functions, a majority of which are wrappers of `BUG` and `panic`.

Logging functions. Unlike terminating functions, functions that only log errors have multiple patterns in the source code. Most of the semantic information is lost in the IR, and functions are translated to a `printk` call. To collect macros that do not have clear patterns such as error severities at the end of the function name [31], we use association mining [7] similar to PR-Miner [28]. The intuition here is, an EH strategy is more likely to exist in the error path rather than in the normal path. To minimize false positives, we set the *confidence* to `0.9`, and a *support* value greater than 4. Using this technique, we identified 112 new error handlers macros that log errors.

5.2 Fine-tuning Contexts and Severity Levels

Extracting Temporal Context. Unlike determining the initialization function set (see §4.3.2), collecting the finalization function set is not straightforward. Although the functions marked with the `__exit` attribute provide an initial set F_f, a comprehensive set of finalization functions are unavailable. To overcome this challenge, we perform a forward control-flow analysis for each call-site found in the functions within F_f. We recursively track and store the life-cycle phase of each call-site function. We mark a function as a finalization function if the function's *confidence* is greater than `0.95` and has a *support* value greater than one. We collect `1,481` additional functions to the finalization set. We evaluate the accuracy of the collected functions in §6.2.

Granularity of the spatial context. We determine the granularity of spatial context empirically from two studies, to minimize underfitting and overfitting to a specific model. In the first study, we collect the EH macros (see §4.1), and the frequency count of each subsystem containing the errors. The second study (Table 4) detects errors whose EH strategy is limited to logging functions. From these two studies, we determined that (1) larger subsystems localize EH within a subsystem. (2) Treating each directory within the subsystem as an independent context generates rules with low confidence and a high false-positive rate. Based on these observations, we refine the spatial context for `drivers`, `arch`, `net`, and `fs` subsystems, to contain the subsystem and the directory. For example: `drivers/media` and `drivers/dma` are different spatial contexts, whereas the spatial context of `sound/soc` and `sound/x86` is `sound`.

Grouping severity levels. As there are no predefined severity levels for error handlers, EeCatch must infer their severities, based on their impact. Previous research [27] suggests that developers

Subsystem / Level	arch	block	certs	crypto	drivers	fs	init	ipc	kernel	lib	mm	net	samples	scripts	security	sound	virt
Emergency	58	0	0	0	126	27	4	0	19	1	17	5	0	0	0	0	0
Alert	23	0	0	0	303	120	0	0	65	6	16	4	0	0	0	10	0
Critical	17	2	0	6	498	140	1	0	15	3	2	66	0	0	1	56	0
Error	658	54	8	379	89500	4877	14	34	612	329	259	2561	92	75	413	7862	75
Warning	349	45	0	37	16015	1204	13	3	440	89	112	438	10	7	51	715	3

Table 4: Criticality of standard error codes determined via known logging functions. We can refine the subsystems fs, drivers, arch, and net across the spatial dimension, without introducing noise.

struggle to determine the severity levels that are next to each other, compared to levels further apart (for example, an EH at level 4 is easy to distinguish from EH at level 1, compared to level 3). In our study of EEH bugs (see §2), we did not find error handlers that use logging, to fix their EH within adjacent levels (2 to 3 or 3 to 4). To avoid subjective determination of the error severity, we group `alert`, `crit`, and `emerg` to a single level - `crit`. Similarly, we group `info`, `notice`, and `debug` to the `info` level. A categorization of the error severity levels is available in Table 1.

5.3 Bug Reporting and Ranking

EeCatch outputs a report containing potential EEH bugs for human analysts to verify. To ensure that the users can quickly validate the report, the tool generates the error, its source code, the EH, its severity, the error-context, the peer EH severities. It also suggests the *expected* severity level, determined as the majority rule. EeCatch employs ranking and pruning strategies to produce a subset, containing a higher proportion of true positives. First, the ranking strategy prefers reports that are further away from the expected severity, followed by smaller values of relative frequency, among the error handlers. Second, EeCatch prioritizes reports where the majority error handlers can cause security issues, such as crash, due to process termination. Users can prioritize bug fixing depending on the severity, while pruning and ranking ensures the top reports are likely to be true positives.

5.4 Scalable Data-Flow Analysis through Memoization

EeCatch performs a data-flow analysis for each error to identify the error handlers along the call chain. As error return is not considered a valid EH strategy, the analysis must explore the call chain repeatedly. Each function within the call chain generates its own set of callers. In the kernel, the chain grows exponentially per error. To avoid repeatedly performing the same data-flow analysis for multiple callers within the same path, EeCatch employs a memoization strategy, to store the results per function. For each error within a function, EeCatch stores the tuple containing the error, possible error handlers, their severities and the error-context. EeCatch uses these results to ensure that the analysis is scalable and updates the entries once per error, within the function. This strategy reduced the running time of exaggerated error-handling bug detection pass from 26 hours to 32 minutes. Further, EeCatch uses a dynamic callgraph that minimizes the overall space overhead.

6 EVALUATION

Experimental setting. In this section, we evaluate EeCatch and its bug detection capabilities using the Linux kernel of version 5.3.0-rc2. The top git commit number of the kernel is `609488bc979f`, the latest stable version as of August 1, 2019. To evaluate our model's capabilities, we chose the Linux kernel as it is complex, critical, and open sourced. We performed the experiments on Ubuntu 18.04 LTS with LLVM version 10.0 installed. The machine has 64GB RAM and an Intel CPU (Xeon R CPU E5-1660 v4, 3.20GHz) with 8 cores. Using the `allyessconfig`, we generated 18,071 LLVM IR bitcode files to cover as many modules as possible. The generated bitcode files are for `x86` architecture.

Hyper-parameter selection. Bug detection in EeCatch relies on one user-defined hyper-parameter, threshold (θ). We chose the value of θ to optimize for both false negatives and false positives. Increasing θ would decrease the likelihood of a warning being a bug, which increases the manual effort. On the other hand, decreasing θ improves the false-positive rate at the cost of completeness (i.e., having more false negatives). To determine θ, we evaluate EeCatch against a set of known bugs as identified in §2.2. The entire dataset contains 239 EEH bugs. We reverted their patches to test if EeCatch can detect them. Table 5 shows the results of this experiment. Based on the evaluation, we choose $\theta = 0.2$ in the current implementation to make a trade-off between false positives and false negatives. In this setting, we can detect 208 EEH bugs out of the possible 239. The false-negative cases involve incorrect error logging severity (7/31), complex code modifications (16/31), and limitations of static analysis (8/31).

Threshold θ	Generated Warnings	Pruned Warnings	Known bugs
0.01	5	0	0
0.1	56	0	9
0.15	237	53	125
0.2	640	104	208
0.25	1157	260	208
0.4	5375	3022	212

Table 5: Evaluating the hyper-parameter value, θ, among possible choices.

Analysis-time performance. EeCatch completed the analyses of the kernel for exaggerated error-handling cases within 56 minutes. Specifically, the analysis to identify and generate exaggerated error-handling warnings required 32 minutes. The other passes such as callgraph generation and alias analysis required 3 and 19 minutes, respectively.

Manual effort. Detecting EEH bugs via EeCatch involves manual effort. For detecting EEH in the Linux kernel, we approximately spent 120 man-hours to identify various rules based on code inspection, 45 man-hours to evaluate various parameters, and 90 man-hours to evaluate warnings and report true EEH bugs.

6.1 Bug Findings

Based on the initial knowledge set (see Table 2) of this experiment, we identified 2.28 million conditional statements, 62 new EH functions that terminate execution, and 112 new error handlers macros that log errors. In all, we collected 705 EH functions logging the error or terminating the execution. We also collected 13,683 and 12,891 functions in the `init` and `exit` functions sets, for the temporal error-context. By setting the θ to `0.2`, EeCatch generated 640 warnings for potential EEH bugs, across varying severities. We first perform pruning to prioritize identifying security-critical bugs (i.e., the ones that may crash the kernel). This step eliminates cases with severity level less than level 4 (see Table 1). The typical configuration for error logging in the Linux machines is `WARNING`, thus levels 4 or higher would indicate severe errors of interest to the maintainers. We rank the remaining results based on the likelihood of them being an EEH bug; that is, the smaller the CST (Component Severity Threshold), the higher the confidence a deviation is a true positive. This selection returns us 104 warnings. We then manually confirm these warnings for real EEH bugs.

We analyzed the 104 reports manually and finally confirmed 64 new EEH bugs. We then submitted patches for all of the confirmed bugs. EeCatch generates a detailed report for each bug suggesting the severity level. Based on this information, it took researchers a total of 90 man-hours to confirm the bugs and submit patches. We spent a significant amount of time analyzing the impact of the fixes compared to identifying true positives. However, we believe the effort is manageable considering our lack of kernel development expertise, and the critical nature of the bug fixes. At the time of writing the paper, maintainers applied patches to fix 48 EEH bugs and confirmed 4 other patches. The four confirmed cases indicate the presence of EEH bugs, but the fixes either require substantial code changes or thorough auditing of all callers of the function. We are still discussing with the maintainers to fix the remaining bugs. Table 10 contains a detailed list of EEH bugs and a summary version of it is in Table 6. Based on our analysis of the warnings, we present some of the interesting findings in the remaining part of this section.

Bug statistic	DoS	Memory corruption	Inconsistent state	Excessive logging	Local DoS
Submitted	4	18	15	2	26
Confirmed	4	11	14	2	21

Table 6: Summary of EEH bugs detected by EeCatch.

Distribution and latent period. Of the confirmed bugs, the majority of the bugs are in `drivers` subsystem, followed by `fs`, `net`; and a couple in `kernel` subsystem. We expected to find bugs in the driver code given the code quality. However, our spatial model for error-context refines the `drivers` subsystem by another level. These results indicate that using a fine-grained subsystem, for example using drivers/media, instead of drivers, is effective in detecting EEH bugs. Second, most of the bugs (66%) can crash the existing process (`BUG`) and have an average latent period of 7 years, with 30 bugs having a latent period greater than 10 years. The other cases can cause inconsistent state in the kernel, generate excessive notifications, memory corruption, and information leaks due to stack dump.

Bug classification. The majority of the confirmed bugs `90%` use `BUG` as the error handlers. To confirm the validity of our model, in identifying other error handlers, we studied all `104` warnings reported by EeCatch. Among the 104 warnings, `28` cases involve `WARN`, `8` cases involve `panic`, and the remaining involve critical logging functions. The two valid cases of `panic`, bugs 41 and 42 in Table 10 detected by EeCatch, are due to incorrect regression patches. We further evaluated all the error handlers involving `panic` in the kernel and observed that 78% of the panic calls are in `init` temporal error-context. On the other hand, 18% of the panic cases are in `drivers/scsi` and `fs/btrfs` subsystems. These observations further validate our error-context model and indicate likely reasons for the skewed distribution in our result set.

Security impact of found bugs. The impact of the detected EEH bugs involves Denial of Service (i.e., crashing the process or the whole kernel), memory corruption, inconsistent states, and information leak. Temporal violations of memory corruption are possible when a process exits from its current context, without releasing its resources. First, two of the bugs calling `panic`, bugs #41 and #42 in Table 10, can crash the kernel caused by triggering a memory allocation failure. As C lacks automatic garbage collection, heap-allocated memory during a process crash is not recovered until a reboot. Second, bug #29, occurring in the btrfs file system, acquires a lock on the file system. It allocates at least 1KB on heap memory and crashes the process via `BUG_ON`. Besides memory leak, it locks the filesystem leading to a kernel crash [38]. Third, EeCatch can detect bugs that cause information leak either by excessive logging (see bug#3) or by leaking kernel addresses (Figure 6). `DCCP_BUG_ON()` causes an information leak but does not crash the process. Finally, bug #6 fixes an incorrect control-flow bug, where the execution of the code halts in drm_dev_init(). The error is detected earlier and logged in the caller devm_drm_dev_init().

```c
/* net/dccp/feat.c */
int dccp_feat_default_value(u8 feat_num)  {
    int idx = dccp_feat_index(feat_num);
    DCCP_BUG_ON(idx < 0);
    // wrapper calls stack_dump() without a crash
    return idx < 0 ? 0 : dccp_feat_table[idx].default_value;
}
```

Figure 6: A potential EEH bug detected by EeCatch, causing an information leak.

Security impact due to fixing strategies. As the majority of detected bugs involve `BUG`, we use the suggested severity of the error handlers to fix the issues. Most of our patches required less than 5 lines of code change and replaced the error handlers by returning the error upstream to the caller, introducing lower severity error handlers and appropriate checks, or using `WARN` instead of `BUG`. The soundness of our patches depends on the suggested EH's CST(Component Severity Threshold), determined from the peer error handlers set. Besides, EeCatch also requires manual effort to verify the security impact of the patch caused by demoting the severity level of EH.

6.2 Importance of Error-Context

In this section, we evaluate the importance of using error-context in identifying EEH bugs. We conduct the same experiment, described in §6, by modifying the definition of the context to contain - only spatial context, only temporal context, and with no context. We further evaluate the emitted warnings and search for the previously confirmed EEH bugs. Table 7 shows the results of this experiment.

Statistic	EeCatch	Temporal context only	Spatial context only	No context
Warnings	640	573	1528	221
Pruned Warnings	104	308	389	43
EEH Bugs	64	13	48	7

Table 7: Impact of error-context in detecting EEH bugs; threshold = 0.2. The columns are the various context configurations within EeCatch.

Evaluation of spatial context. By running EeCatch only with the spatial context, we observed that EeCatch generates 1528 warnings. Similar to our main experiment, we rank and evaluate the warnings less than level 4. In this experiment, we identified 48 of the 64 EEH bugs among 389 warnings. We believe the lack of temporal context introduces both a large number of false positives and many false negatives.

Evaluation of temporal context. Running EeCatch without the spatial context resulted in generating 573 warnings. After ranking and looking at the 308 warnings below level 4, we detected 13 EEH bugs. Moreover, we relaxed the threshold to 0.5, and found 21 bugs, fewer than 48 bugs, detected by modeling the spatial context. Therefore, lacking the spatial context introduces a large number of false negatives.

Evaluation with no context. In the third experiment to study the error-context, we consider error and the error handlers with previously established severity levels. By eliminating the error-context feature, we generated 221 warnings and detected 7 EEH bugs from a pruned list containing 43 warnings. The significant decrease in warnings compared to other error-context is due to a higher proportion of errors being handled at level 6 (Table 1) and are eliminated by Equation 2. These experiments suggest modeling the error-context can dramatically improve the detection of EEH bugs within the Linux kernel. Additionally, the results suggest that the spatial context explores new branches, and the temporal context bounds the search space.

6.3 False Positives

We analyze 104 warnings post ranking and pruning the 640 warnings, and identify 64 bugs. The false-positive rate of evaluated bugs of EeCatch is 64 out of 104, for a false-positive rate of 40%. While we are confident that analyzing more warnings would identify more EEH bugs, it will further increase the false-positive rate. We analyze the causes of false positives as follows.

Multiple error severities. If an error is logged and handled differently across the call-chain, EeCatch currently chooses the more severe level as the valid EH severity. Such a strategy minimizes EEH false negatives but introduces false positives when the actual EH strategy ignores the error. We estimate 50% of the false positives are due to an incorrect assignment of severity level. A possible solution is to use the average severity level in the call-chain.

Switch-statement default. We observed that the default case of switch statements in the kernel (`SwitchInst`) often uses exaggerated error-handling techniques and contributes to 25% of false positives. This is commonly seen in `drivers` and `net` subsystems which perform validation of user input. While the IR was able to generate correct instructions for the other cases of `switch` statement, EeCatch marks the default case as an EEH bug. Currently, our model cannot eliminate such bugs, as the warning is a valid EH according to our definition.

Statistical analysis. As described in §4.4, EeCatch employs a probabilistic approach to distinguish an EEH bug from false positives. This approach eliminates reasoning about the semantics of the error, but it can introduce false positives. While we did not identify such a scenario in our evaluation, increasing θ can cause false positives in this category. Besides, we tuned other hyper-parameters to minimize the false positive rate. Table 9 contains a detailed evaluation of hyper-parameters.

Others. 10% of the false positives are due to other causes, including imprecise call-graph, limitations of data-flow analysis and points-to analysis, difficult to analyze warnings, among others. We rely on manual analysis to eliminate the majority of false positives in these categories. One can mitigate a few of these false positives by employing more sophisticated programming analyses like symbolic execution [42] or dynamic taint analysis, among others.

6.4 False Negatives

EeCatch uses a simple model to detect exaggerated error-handling bugs and attempts to minimize the false positives at the cost of completeness. We list a few of the design choices leading to potential false negatives. First, EeCatch defines errors as the standard error codes set in `errno.h`. However, there are numerous other non-standard errors, within each subsystem, not captured by EeCatch. Second, we use both statistical analysis and heuristic-based design to detect the error-context and infer error handlers' severity. These design choices are potential sources for false negatives in detecting EEH bugs. Third, we use a hyper-parameter (θ) to avoid reasoning about the code semantics for identifying the EH severity. Using statistical analysis can mitigate false positives yet it is vulnerable to false negatives. Fourth, data-flow analysis is not precise. Potential false negatives caused by data-flow analysis include overwritten return value and lost errors within the same function [16].

6.5 Portability

Besides the recursive backward data-flow analysis, the three main components within EeCatch are the identification of EH, the modeling of two-dimensional context, and the modeling of severity levels of EH. Classifying and assigning the severity levels to EH is based on error log levels available in most production software. We empirically analyze how EeCatch might implement the other two techniques in other codebases, including POSIX-based OS, Chromium browser, OpenSSL, and microkernel-based OS such as

Software	Written in	Lines of code	Error & EH identification	Error-context identification	Severity Levels	Relative effort
POSIX kernels	C	15M	Similar to Linux	Similar to Linux	✓	Similar
Chromium	C++	8M	Error codes, exception handling	Spatial and temporal contexts	✓	Greater
OpenSSL	C	500K	Error codes, logging, and terminating functions	N/A due to code size & impact	✓	Lesser
Fuchsia OS	C/C++	2.5M	Error codes, exception handling, logging & terminating functions	Similar spatial & temporal contexts for Zircon microkernel	✓	Similar

Table 8: Portability of EeCatch's techniques. The 'Relative effort' column compares the effort of software against the Linux kernel.

Fuchsia, as summarized in Table 8. We describe the impact of hyper-parameters used in EeCatch in Table 9.

Linux and other POSIX-based kernels (e.g., FreeBSD, Android, and Darwin-XNU) are written in C and are compatible based on the POSIX standard. They store standard error codes in a single file such as `errno.h`, use terminating functions, logging functions, and share similar contexts. In fact, besides identifying the corresponding EH, EeCatch requires minimal modification to detect EEH bugs in the other kernels. OpenSSL, written in C, is also similar to Linux, in terms of identifying errors and EH. Due to its smaller code size and its criticality as application software, a fine-grained error-context is unnecessary. Therefore, EeCatch is portable to OpenSSL and similar sized applications by running without a context (see §6.2).

C++-based software such as Chromium browser and Zircon microkernel in Fuchsia OS, use exceptions and error codes to handle errors. Extracting various user-defined exception error handlers and generating patterns for terminating and logging functions do require certain coding expertise. On the other hand, these software programs are similar to the kernel in modeling contexts. For example, spatial contexts can include discrete modules and third-party software. Temporal contexts include resource allocation, functional operations, and resource release functions. As such, porting EeCatch to C++ based software is straight-forward, and we foresee manageable manual effort in generating various specifications.

7 DISCUSSION

In this section, we discuss limitations of EeCatch that can be potentially improved and explored as future work directions.

Capturing more errors. There are a wide variety of custom error codes in use within the kernel. EeCatch uses association mining to identify EH macros. However, we ignore the error macros that use positive integers, or variables with return values less than zero, to minimize the false positives. An improved model can include such macros and the associated error handlers, to make the EeCatch more complete. To detect these errors, one might perform association mining starting with the known EH and identifying the errors leading to these EH. After eliminating the standard errors, we identify the error-variables that have a high likelihood of containing an EH as error codes.

Model accuracy. One can improve the detection rate of EEH bugs by modeling more factors within the error-context. Our model uses equal weights for spatial and temporal contexts - to indicate equal importance. However, a more precise model might weight the factors of error-context proportionate to their importance. One such model might prioritize local EH, which is using the same EH as the majority EH within each function, higher than the temporal context (see §6.2).

On the other hand, we can also improve the accuracy of EeCatch by refining the temporal context. There are research works [2, 10, 41] that attempt to determine functions with similar context, using various techniques. By modeling a list of similar functions based on their names, previous works [19, 30, 41, 44] can enhance EeCatch's temporal context.

Exploitability of EEH bugs. EEH bugs may cause DoS attacks via a system crash. Fuzzers, such as Syzkaller, identified a number of crashes that can be classified as EEH bugs. These bugs not only cause DoS, but are also capable of memory corruption, information leak, and inconsistent state. The goal of exploit generation is to trigger the corresponding error that leads to the EEH bug. Therefore, a successful exploit has two parts: (1) reaching the code of the error (2) triggering the error. Automated exploit generation is a challenging and open problem; it is out of the scope of this work. However, we discuss the following techniques that could facilitate the exploit generation. To reach the error code, an analyst can use either directed fuzzing (e.g., [4, 13, 14]) or symbolic execution (e.g., UC-KLEE [42]). To trigger heap-based errors (the majority of detected bugs), a line of research [24, 43] has tried to use different vulnerabilities to manipulate heap allocation failures [17].

Minimizing manual effort via automation. While designing EeCatch, we spent a large amount of manual effort in identifying various error-context. We argue that the effort is manageable given the complexity of the target system and the impact of the identified bugs. The automation effort of EeCatch took us over 480 man-hours, and its techniques scale to other large systems, written in C. To reduce the manual effort, EeCatch employs a flexible specification list containing new error handlers and is extensible to include new error-context within its model.

Hyperparameter optimization. We chose the value of θ empirically to minimize both the human effort in verifying the warnings and the false-negative rate. A robust model accounts for over-fitting by evaluating on a validation dataset (e.g., k-fold cross-validation). However, we have not performed validation before evaluating the test data (whole kernel). Given the size of our study set, we believe $k = 10$ to be an optimal size for cross-validation, without repetition. Further, our study is indirectly impacted by variables including, granularity of spatial context, number of severity levels, support and confidence values for terminating functions, logging functions, and temporal context identification phases (see Table 9). We chose the confidence values to be permissive (two standard deviations from the mean), and picked support values based on our observations from the training dataset. These parameter values present an external threat to validity when applying EeCatch on other software systems.

8 RELATED WORK

8.1 Detection of EH bugs

Rule-based EH specification. Previously, researchers relied on observations to generate specifications in large software such as the kernel. Prior works such as LRSan [58], Talos [21], and AutoISES [49] use range-based numeric values to represent errors. In the kernel, these values are in the file `errno.h`. EeCatch also uses range-based values and `NULL` to represent errors. Other works using implicit programming rules to identify EH include Hector [46], EIO [16], Rubio-González et al. [45], and Pex [64]. CheQ [31] uses keyword- and wrapper-based approach to identify EH that includes error return, error logging, and stopping execution. Unlike EeCatch, all these works treat error return as valid EH. EeCatch uses rule-based specification to detect EH. It further identifies occurrences of `UnreachableInst` and `__noreturn` in the IR to collect wrappers of terminating functions.

Automatic inference of EH specification. To minimize the manual effort, many researchers proposed techniques to automatically infer the specifications of EH based on code properties. Acharya et.al [1] inferred APIs by mining static traces of a system's run-time behaviors. Tools using APIs to infer specifications include APISan [63], ErrDoc [50], PEH [22], and Apex [23]. On the other hand, there are many other tools [9, 12, 29, 37, 48, 61] that infer the specification not limited to APIs. Similar to these tools, EeCatch also uses statistical analysis to detect EH that logs the error. It expands the number of logging functions by identifying macros across modules.

Detection of incorrect EH. Research works focusing on the adequacy of EH in software include identifying missing-check bugs [16, 22, 50]. Aspirator [62] performs a postmortem failure diagnosis of EH causing catastrophic failures in distributed systems. Their detector works on Java byte code and is simple - it issues warnings when the EH is empty. There are other works that can detect incorrect EH by either using fault injectors [5, 35, 47] or an unintended crashes while fuzzing [13, 56]. However, both techniques cannot explore deeper paths. By contrast, EeCatch targets exaggerated error-handling, an opposite type of bugs against inadequate-EH bugs.

8.2 Analysis of OS Kernels

Static analysis. EeCatch performs a static analysis of the Linux kernel. There are other research tools that also analyze the kernel, such as Dr. Checker [34] and K-Miner [15]. The tools perform data-flow analysis on device drivers and perform multiple passes from the system calls. Researchers also use Coccinelle [40] to perform static analysis using source code pattern matching to find bugs, such as [57]. Similarly, Smatch [6], which is based on Sparse [53], is a code-checking framework targeted to find simple bugs in the kernel using source-code analysis. Other complementary approaches to static analysis use machine learning [61] and symbolic execution [8, 42, 63]. EeCatch uses inter-procedural, field-, and context-sensitive, static analysis techniques to efficiently identify EEH bugs over the whole kernel. It tracks various error handlers across each call-chain, using scalable data-flow analysis techniques.

Dynamic analysis. Besides static analysis, dynamic analysis techniques such as fuzzing are capable of detecting crashes. Syzkaller [55] detected many crashes that are due to EEH bugs. As previously described, dynamic analysis suffers from limitations in exploring deeper paths and requires significant fuzzing time to detect the bugs. In comparison, EeCatch can generate `640` EEH warnings in less than an hour.

Statistical analysis. Engler et.al [11] were among the first to explore the idea of statistical analysis, cross-check a property, and detect bugs. Despite the unsound approach, the idea was widely adopted by other researchers such as Yamaguchi et.al [60], APISan [63], Crix [32], and Juxta [36], to detect bugs in the OS kernels. EeCatch also uses statistical analysis to infer the severity of errors to minimize false positives.

Semantic-bug detection. Research works targeting detection of semantic bugs include - Deadline [59] and Wang et.al [57] to detect double fetch bugs, UniSan [33] to detect uninitialized uses of variables, LRSan [58] to detect lacking recheck bugs. Moreover, Pallas [20] detects inconsistent bugs in fast paths, DCNS [3] detects incorrect sleeping functions in atomic context bugs, and Hector [46] detects bugs involving a failure to release resources. These tools either use the multi-pass framework provided in LLVM IR or perform code based pattern mining using Coccinelle. We built EeCatch as a sequence of LLVM passes, to benefit from the information contained in the IR. In comparison, EeCatch detects a new class of semantic bugs - EEH bugs across the kernel using two-dimensional context information.

9 CONCLUSION

In this paper, to the best of our knowledge, we present the first study of Exaggerated Error Handling (EEH) bugs, a new class of critical semantic bugs that has been largely overlooked. EEH bugs may unnecessarily cause DoS such as crashing the whole system or the process. Based on the findings from our study, we develop EeCatch, an effective and scalable tool for detecting EEH bugs. EeCatch models and extracts both spatial and temporal contexts of errors to determine the appropriate error-handling mechanisms and thus to detect EEH. EeCatch's analysis is context-, field-sensitive, inter-procedural, and built on top of LLVM. Evaluating EeCatch on the entire Linux kernel, we found 64 new EEH bugs that may cause critical security issues like crashing the kernel. Linux maintainers have fixed most of the bugs using our patches. We hope that this paper raises the awareness of the critical impact of EEH so that developers could minimize EEH in the future.

10 ACKNOWLEDGMENT

We would like to thank our shepherd, Herbert Bos, and the anonymous reviewers for their helpful suggestions and comments. We are also grateful to Linux maintainers for providing prompt feedback on patching bugs. 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] Mithun Acharya and Tao Xie. 2009. Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 370–384.

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[3] Jia-Ju Bai, Julia Lawall, Wende Tan, and Shi-Min Hu. 2019. DCNS: Automated Detection Of Conservative Non-Sleep Defects in the Linux Kernel. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 287–299. https://doi.org/10.1145/3297858.3304065

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344.

[5] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. 2002. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*. Citeseer.

[6] Dan Carpenter. 2009. Smatch - the source matcher. http://smatch.sourceforge.

[7] Aaron Ceglar and John F Roddick. 2006. Association mining. *ACM Computing Surveys (CSUR)* 38, 2 (2006), 5.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *SIGPLAN Not.* 47, 4 (March 2011), 265–278. https://doi.org/10.1145/2248487.1950396

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 73–88.

[10] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-Based Function Embedding and Its Application to Error-Handling Specification Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 423âĂŞ433. https://doi.org/10.1145/3236024.3236059

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 57–72. https://doi.org/10.1145/502059.502041

[12] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[14] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 474–484.

[15] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[16] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct.. In *FAST*, Vol. 8. 1–16.

[17] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic heap layout manipulation for exploitation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 763–779.

[18] Tetuso Honda. 2017. tty: Avoid possible error pointer dereference at tty_ldisc_-restore(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=598c2d41ff44889dd8eced4f117403e472158d85.

[19] Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *European Conference on Object-Oriented Programming*. Springer, 294–317.

[20] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-Aware Checking for Finding Deep Bugs in Fast Path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 709–722.

[21] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *2016 IEEE Symposium on Security and Privacy (SP)*. 618–635. https://doi.org/10.1109/SP.2016.43

[22] Z. Jia, S. Li, T. Yu, X. Liao, J. Wang, X. Liu, and Y. Liu. 2019. Detecting Error-Handling Bugs without Error Specification Input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 213–225. https://doi.org/10.1109/ASE.2019.00029

[23] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 472–482.

[24] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 957–972.

[25] Jean-Claude Laprie. 1995. Dependable computing: Concepts, limits, challenges. In *Special issue of the 25th international symposium on fault-tolerant computing*. 42–54.

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[27] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22, 4 (01 Aug 2017), 1684–1716. https://doi.org/10.1007/s10664-016-9456-2

[28] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 306–315. https://doi.org/10.1145/1095430.1081755

[29] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 306–315.

[30] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*. 1063–1073.

[31] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Computer Security – ESORICS 2019*, Kazue Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer International Publishing, Cham, 3–25.

[32] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1769–1786. https://www.usenix.org/conference/usenixsecurity19/presentation/lu

[33] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[34] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1007–1024.

[35] Paul D. Marinescu, Radu Banabic, and George Candea. 2010. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, USA, 23.

[36] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[37] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Acm Sigplan Notices*, Vol. 47. ACM, 997–1016.

[38] Ingo Molnar. 2015. Deprecate BUG_ON() use in new code, introduce CRASH_-ON(). https://lore.kernel.org/patchwork/patch/568291/.

[39] OWASP. 2017. OWASP Top 10 - The Ten Most Critical Web Application Security Risks. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.

[40] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*.

[41] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 147.

[42] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

[43] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 25–35.

[44] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–22.

[45] Cindy Rubio-González, Haryadi S Gunawi, Ben Liblit, Remzi H Arpaci-Dusseau, and Andrea C Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *ACM Sigplan Notices*, Vol. 44. ACM, 270–280.

[46] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.

[47] Martin Susskraut and Christof Fetzer. 2006. Automatically finding and patching bad error handling. In *2006 Sixth European Dependable Computing Conference*. IEEE, 13–22.

[48] Mana Taghdiri and Daniel Jackson. 2007. Inferring specifications to detect errors in code. *Automated Software Engineering* 14, 1 (2007), 87–121.

[49] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.

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

[51] Linus Torvalds. 2007. BUG_ON in workingset_node_shadows_dec triggers. https://lkml.org/lkml/2016/10/4/1.

[52] Linus Torvalds. 2007. Do not use BUG. https://yarchive.net/comp/linux/BUG.html.

[53] Linus Torvalds. 2019. Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.

[54] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 762–774.

[55] Dmitry Vyukov. 2015. Syzkaller.

[56] Dmitry Vyukov. 2019. Syzbot and the Tale of Thousand Kernel Bugs. https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Syzbot-and-the-Tale-of-Thousand-Kernel-Bugs-Dmitry-Vyukov-Google.pdf.

[57] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1–16. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei

[58] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 1899âĂŞ1913. https://doi.org/10.1145/3243734.3243844

[59] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[60] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. 797–812. https://doi.org/10.1109/SP.2015.54

[61] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 499–510.

[62] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[63] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[64] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1205–1220.

A APPENDIX

A.1 Detected Bugs

Component	Hyper-parameter	Rules used in Linux kernel	Rule generation technique	Sensitivity	Software
EH identification	Logging functions	Confidence = 0.9, support = 4 instances	Determined via association mining	90%	F, P, O, C
EH identification	Terminating functions	Track UnreachableInst and __noreturn attribute. Number of instances > 1	Control-flow analysis	95%	P
Spatial context	Granularity	Subsystem and folder	Empirically determined (Table 4)	95%	F, P, C
Temporal context	Initialization phase	Track __init attribute for function, and for their callees support > 1 instance, confidence = 0.95	Control-flow analysis. Assumes normal distribution	99%	F, P
Temporal context	Finalization phase	Track __exit attribute for function, and for their callees support > 1 instance, confidence = 0.95	Control-flow analysis. Assumes normal distribution	80%	F, P
Bug detection	Threshold	0.2	Likelihood computation (Equation 2) Empirically determined (Table 5)	60%	F, P, O, C

Table 9: Hyper-parameters used in modeling EeCatch. Column Sensitivity indicates the estimated true positive rate of applying the rules from column 3 to the Linux kernel. Column Software refers to the applicability of rules to other software. F, P, O, and C indicate Fuchsia OS, POSIX-based kernels, OpenSSL, and Chromium browser respectively.

ID	File	Function	Handler	Impact	Sev	Years	Status
1	drivers/spi/spi-dw.c	dw_spi_add_host	BUG_ON	Inconsistent state	2	10	A
2	drivers/net/ppp/pppoe.c	pppoe_pernet	BUG_ON	DoS	2	10	A
3	net/atm/clip.c	unlink_clip_vcc	pr_crit	excessive logging	3	15	A
4	drivers/media/.../vpfe_capture.c(15)	vpfe_register_ccdc_device	BUG_ON	local DoS	1	10	A
5	drivers/net/.../orinoco/orinoco_usb.c	ezusb_priv	BUG_ON	Inconsistent state	2	6	A
6	drivers/gpu/drm/drm_drv.c	drm_dev_init	BUG_ON	Inconsistent state	1	1	A
7	drivers/uwb/lc-dev.c	uwb_dev_add	BUG_ON	local DoS	2	4	X
8	drivers/uwb/lc-dev.c	uwb_dev_add	BUG_ON	local DoS	2	4	X
9	drivers/uwb/lc-dev.c	uwb_dev_add	BUG_ON	local DoS	2	4	X
10	drivers/xen/grant-table.c	grow_gnttab_list	BUG_ON	Inconsistent state	2	2	A
11	drivers/xen/grant-table.c	nr_status_frames	BUG_ON	Inconsistent state	2	2	A
12	drivers/xen/grant-table.c	gnttab_expand	BUG_ON	Inconsistent state	2	2	A
13	drivers/xen/grant-table.c	gnttab_init	BUG_ON	Inconsistent state	2	2	A
14	drivers/nfc/s3fwrn5/firmware.c	s3fwrn5_fw_recv_frame	BUG_ON	excessive checking	2	4	A
15	drivers/media/.../cx231xx-i2c.c	cx231xx_i2c_register	BUG_ON	memory corruption	2	10	A
16	drivers/media/.../saa7146_video.c	video_begin	BUG_ON	memory corruption	2	15	A
17	drivers/media/.../saa7146_video.c	video_end	BUG_ON	memory corruption	1	15	A
18	drivers/base/regmap/regcache.c	regcache_read	BUG_ON	memory corruption	1	8	S
19	drivers/base/regmap/regcache.c	regcache_write	BUG_ON	memory corruption	1	8	S
20	drivers/base/regmap/regcache.c	regcache_sync	BUG_ON	local DoS	1	6	S
21	drivers/base/regmap/regcache.c	regcache_sync_region	BUG_ON	memory corruption	1	5	S
22	drivers/char/tpm/tpm_ppi.c	tpm_eval_dsm	BUG_ON	memory corruption	1	5	A
23	drivers/staging/.../fileops.c	kpc_dma_transfer	BUG_ON	memory corruption	2	<1	A
24	drivers/staging/.../fileops.c	kpc_dma_transfer	BUG_ON	local DoS	2	1	A
25	drivers/net/hamradio/hdlcdrv.c	hdlcdrv_register	BUG_ON	local DoS	2	14	A
26	drivers/net/caif/caif_serial.c	caif_xmit	BUG_ON	local DoS	2	9	A
27	fs/btrfs/check-integrity.c	btrfsic_process_superblock	BUG_ON	memory corruption	1	8	C
28	fs/ecryptfs/crypto.c	crypt_scatterlist	BUG_ON	memory corruption	1	12	A
29	fs/btrfs/extent_io.c	__clear_extent_bit	BUG_ON	DoS	1	8	C
30	fs/gfs2/trans.c	gfs2_trans_begin	BUG_ON	local DoS	1	12	S
31	net/mac80211/util.c	wiphy_to_ieee80211_hw	BUG_ON	memory corruption	1	11	A
32	net/rfkill/core.c	rfkill_register	BUG_ON	memory corruption	1	10	A
33	kernel/bpf/core.c	bpf_prog_realloc	BUG_ON	inconsistent state	2	5	A
34	drivers/atm/fore200e.c	fore200e_send	ASSERT	inconsistent state	2	14	A
35	drivers/atm/fore200e.c	fore200e_send	ASSERT	local DoS	2	14	A
36	drivers/atm/fore200e.c	fore200e_close	ASSERT	inconsistent state	2	14	A
37	drivers/block/rbd.c	__rbd_object_map_load	rbd_assert	inconsistent state	2	<1	S
38	drivers/infiniband/hw/cxgb3/iwch_qp.c	iwch_modify_qp	BUG_ON	leak, inconsistent state	2	12	C
39	drivers/gpu/.../amdgpu_dm.c	dm_update_crtc_state	BUG_ON	memory corruption	2	1	S
40	drivers/infiniband/ulp/srpt/ib_srpt.c	srpt_queue_response	BUG_ON	memory corruption	2	8	A
41	arch/x86/platform/olpc/olpc_dt.c	prom_early_alloc	panic	DoS	1	<1	C
42	drivers/clk/samsung/clk.c	samsung_cmu_register_one	panic	DoS	1	4	A
43	drivers/scsi/aic94xx/aic94xx_task.c	asd_unbuild_smp_ascb	BUG_ON	memory corruption	2	13	S
44	drivers/target/tcm_fc/tfc_io.c	ft_recv_write_data	BUG_ON	memory corruption	1	8	A
45	fs/nfsd/nfs4xdr.c	nfsd4_encode_replay	BUG_ON	local DoS	2	15	A
46	drivers/gpu/.../vmwgfx_resource.c	vmw_resource_alloc_id	BUG_ON	memory corruption	2	8	S
47	fs/ocfs2/dlm/dlmmaster.c	dlm_migrate_lockres	BUG_ON	inconsistent state	1	8	A
48	drivers/media/.../saa7146_fops.c	saa7146_buffer_finish	BUG_ON	inconsistent state	1	14	A
49	fs/nfsd/nfs4layouts.c	nfsd4_layout_setlease	BUG_ON	memory corruption	2	5	S
50	drivers/net/ppp/ppp_generic.c	ppp_pernet	BUG_ON	inconsistent state	2	10	A

Table 10: List of new bugs detected by EeCatch. Column 6, Sev is the peer severity referenced from Table 1, suggested by EeCatch. Years in column 7 is the latent period the detected bugs. The fields A, C, S, X in Status column 8 indicate the status of the patch - applied, confirmed, submitted, and module not present in mainline respectively. Bug #4 has 15 instances of EEH, individually detected by EeCatch but fixed with a single patch.