Cut to the Chase: An Error-Oriented Approach to Detect Error-Handling Bugs

HAORAN LIU, National University of Defense Technology, China
ZHOUYANG JIA, National University of Defense Technology, China
SHANSHAN LI[*], National University of Defense Technology, China
YAN LEI, Chongqing University, China
YUE YU, National University of Defense Technology, China
YU JIANG, Tsinghua university, China
XIAOGUANG MAO, National University of Defense Technology, China
XIANGKE LIAO, National University of Defense Technology, China

Error-handling bugs are prevalent in software systems and can result in severe consequences. Existing works on error-handling bug detection can be categorized into template-based and learning-based approaches. The former requires much human effort and is difficult to accommodate the software evolution. The latter usually focuses on errors of API and assumes that error handling should be right after the handled error. Such an assumption, however, may affect both learning and detecting phases.

The existing learning-based approaches can be regarded as API-oriented, which starts from an API and learns if the API requires error handling. In this paper, we propose EH-Digger, an ERROR-oriented approach, which starts from an error handling. Our approach can learn why the error occurs and when the error has to be handled. We conduct a comprehensive study on 2,322 error-handling code snippets from 22 widely used software systems across 8 software domains to reveal the limitation of existing approaches and guide the design of EH-Digger. We evaluated EH-Digger on the Linux Kernel and 11 open-source applications. It detected 53 new bugs confirmed by the developers and 71 historical bugs fixed in the latest versions. We also compared EH-Digger with three state-of-the-art approaches, 30.1% of bugs detected by EH-Digger cannot be detected by the existing approaches.

CCS Concepts: • **Software and its engineering** → *Error handling and recovery.*

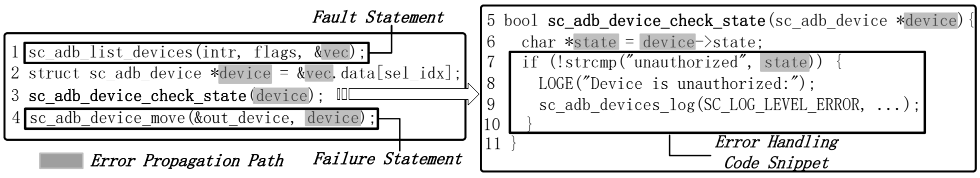
Additional Key Words and Phrases: Bug Detection, Error Handling, Error Oriented

ACM Reference Format:
Haoran Liu, Zhouyang Jia, Shanshan Li, Yan Lei, Yue Yu, Yu Jiang, Xiaoguang Mao, and Xiangke Liao. 2024. Cut to the Chase: An Error-Oriented Approach to Detect Error-Handling Bugs. *Proc. ACM Softw. Eng.* 1, FSE, Article 80 (July 2024), 23 pages. https://doi.org/10.1145/3660787

[*]Corresponding Author

Authors' addresses: Haoran Liu, National University of Defense Technology, Deya road 109, Changsha, China, liuhaoran@nudt.edu.cn; Zhouyang Jia, National University of Defense Technology, Deya road 109, Changsha, China, jiazhouyang@nudt.edu.cn; Shanshan Li, National University of Defense Technology, Deya road 109, Changsha, China, shanshanli@nudt.edu.cn; Yan Lei, Chongqing University, Shazheng street 156, Chongqing, China, yanlei@cqu.edu.cn; Yue Yu, National University of Defense Technology, Deya road 109, Changsha, China, yuyue@nudt.edu.cn; Yu Jiang, Tsinghua university, Shuangqing Road 30, Beijing, China, jy1989@mail.tsinghua.edu.cn; Xiaoguang Mao, National University of Defense Technology, Deya road 109, Changsha, China, xgmao@nudt.edu.cn; Xiangke Liao, National University of Defense Technology, Deya road 109, Changsha, China, xkliao@nudt.edu.cn.

ACM 2994-970X/2024/7-ART80
https://doi.org/10.1145/3660787



```
1 sc_adb_list_devices(intr, flags, &vec);
2 struct sc_adb_device *device = &vec.data[sel_idx];
3 sc_adb_device_check_state(device);
4 sc_adb_device_move(&out_device, device);
```

```
5 bool sc_adb_device_check_state(sc_adb_device *device){
6   char *state = device->state;
7   if (!strcmp("unauthorized", state)) {
8     LOGE("Device is unauthorized:");
9     sc_adb_devices_log(SC_LOG_LEVEL_ERROR, ...);
10  }
11 }
```

Fig. 1. Real-world error-handling example that is hard to be understood by existing approaches.

1 INTRODUCTION

Most software systems frequently encounter errors, and it is essential that reliable software is designed to behave gracefully in the face of such errors [7]. This requires the software to accurately detect failure conditions and handle them appropriately. Incorrect handling of errors can lead to severe problems such as system crashes, data loss, security vulnerabilities [39], and so on. C does not provide exception-handling mechanisms, while C++ contains far fewer try blocks than languages like Java [15]. The error handling in C/C++ needs to be implemented by the developers themselves, so it is more flexible and error-prone [37, 38]. Therefore, in this paper, we focus on error-handling bugs in C/C++ programs.

Existing works on error-handling bug detection can be categorized into two classes: template-based approaches [10, 13, 41, 42] and learning-based approaches [17, 18, 20]. On the one hand, template-based approaches usually depend on manually summarized patterns. For example, ErrHunter [49] summarized error-handling patterns for error code and null pointer in Linux Kernel, ErrDoc [42] and EPEx [16] require error specifications from user input. These approaches require sufficient domain knowledge and are hard to keep up with the changes caused by software evolution [18, 23]. On the other hand, the learning-based approaches usually contain two phases, i.e., learning and detecting. [19, 20]. In the learning phase, they focus on API calls and learn association rules between API calls and their nearby error-handling code snippets (if any). The rules are like "*malloc* requires error handling when returning *NULL*". In the detecting phase, they detect violations of the rules as error-handling bugs. These approaches assume that error handling should be right after the handled error. Such an assumption, however, may affect both phases.

We use an example to demonstrate the limitations of existing approaches in the learning and detection phases. Fig. 1 illustrates a representative error-handling in C from a real-world project, Scrcpy [36]. An unauthorized *device* may cause failure in *sc_adb_device_move* (line 4). Line 7-10 is an error-handling code snippet, and the error being handled is that the *device* generated by *sc_adb_list_devices* (line 1) should not be `unauthorized` (line 7). This error propagates along with variables *vec* and *device* (line 1-3, 6, 4), and is handled (line 7-10) before the *device* is used by *sc_adb_device_move* (line 4). In the learning phase, existing approaches [19, 20] establish associations between *strcmp* and its nearby error-handling code snippet (line 7-10). They may learn wrong rules such as "the return value of *strcmp* requires handling", and miss correct rules like "the error of *sc_adb_list_devices* (line 1) should be handled before *sc_adb_device_move*" (line 4). Even assuming that existing approaches could, somehow, learn the above correct rule, they may still report false positives in the detecting phase — they cannot find error handling near *sc_adb_list_devices* (line 1), which is actually handled through an inter-procedural check (line 3, 7-10).

The limitations of existing approaches are mainly caused by their API-oriented design, which starts from an API and learns if the API requires error handling. In this paper, we propose an ERROR-oriented approach, which starts from an error handling. Our approach can learn why the error occurs and when the error has to be handled. This can be explained by using the **Fault-Error-Failure** model [22]. **Fault** is a flaw in code. **Failure** is an observed behavior. **Error** is abnormal states (value discrepancies) propagating from fault to failure. For example, in Fig. 1, line 1 may generate a *fault*, while line 4 may cause a *failure*. The *error* propagating from line 1 to line 4 is handled in line 3 through an inter-procedural check, and the *check condition* in line 7 determines whether the error-handling code snippet (line 7-10) can be executed. From the error-handling code snippet, we can trace backward to the statement that may generate fault (or *fault statement* for short), and trace forwards to the statement that may cause failure (or *failure statement* for short). The error caused by the fault statement has to be handled before the failure statement. As a proof of conception, we propose EH-Digger, an error-oriented tool to detect error-handling bugs. EH-Digger first collects error-handling code snippets from the code repository, then extracts a code sequence for each error from its fault statement to its failure statement by using inter-procedural analysis. We refer to the code sequence as an **error-handling context**. Taking Fig. 1 as an example, EH-Digger traces the variable *state* in line 7 both backwards and forwards, and extracts line 1, 2, 3, 6, and 4 as the error-handling context. Finally, EH-Digger can learn error-handling rules from frequent contexts.

To better understand the limitations of existing approaches and guide the design of EH-Digger, we conducted studies on 2,322 error-handling code snippets from 22 software systems. We first study the limitations of existing works and find that 43.3% of errors are not handled right after the fault statements. Such cases may affect both learning and detecting phases as discussed above. To address this problem, we propose an error-oriented approach, EH-Digger, which learns frequent contexts of code snippets handling the same error, and then detects error-handling bugs when the contexts occur without proper error handling. There are three main challenges during the design of EH-Digger:

- First, it is non-trivial to determine if two code snippets are handling the same error. To address this, we study the characteristics of errors, and find that 93.1% error-handling code snippets handling the same error can be identified by tracing root causes of the error-prone variables in check conditions. The root causes can be in the form of either error-prone data type (27.4%) or error-prone variable values (72.6%).
- Second, it is hard to represent frequent contexts that may have various forms for the same error. Therefore, we study the characteristics of contexts and find they can be represented with *actions* performing on a series of error-prone variables. The actions include declaration (17.2%), definition (34.5%), and usage (48.3%). One context contains 7.8 actions on average, and 58.4% of contexts can retain the same semantics when changing the action orders.
- Third, tracing error contexts during both learning and detecting phases requires inter-procedural analysis, which may lead to the exponential explosion problem in search space. To avoid this, we propose a summary-based method to perform the inter-procedural analysis. EH-Digger first generates error-handling context for each individual function, then concatenates whole contexts along the call graphs.

We evaluated the performance of EH-Digger in detecting real-world bugs in the Linux Kernel and 11 applications having high stars in GitHub. In the Linux kernel, EH-Digger detected 214 bugs with a precision of 90.3%, and 40 of them are historical bugs that have been fixed in the latest version. We also selected 20 detected bugs that we were capable of fixing and submitted patches to the Linux Kernel. Currently, all of them have been confirmed by the developers, and the rest are awaiting feedback. In the 11 applications, EH-Digger correctly detected 163 bugs with a precision of 89.6% (163/182), of which 33 have been confirmed or fixed by developers, and 31 are historical bugs that have been fixed in the latest versions. The rest are still under discussion. We compared EH-Digger with three state-of-the-art approaches. 30.1% (49/163) of the bugs detected by EH-Digger cannot be detected by the state-of-the-art approaches. The result indicates that EH-Digger can serve as a complementary approach to existing approaches in detecting error-handling bugs.

The key contributions of this paper include:

- We conducted a study on error-handling context using 2,322 code snippets from 22 software systems. The findings help understand the limitations of existing works, and guide the design of our approach.
- We proposed an error-oriented learning method EH-Digger to detect error-handling bugs, which learns error-handling code from the perspective of its context, enabling us to detect error-handling bugs more accurately.
- We applied EH-Digger to the Linux Kernel and 11 open-source software. EH-Digger detected 37 new bugs and 71 historical bugs with a precision of around 91%. 30.1% of bugs detected by EH-Digger cannot be found by comparative approaches.

2 UNDERSTANDING THE ERROR-HANDLING CONTEXT

In this section, we take an in-depth look into the error-handling context through an empirical study. Our study is driven by two research questions:

- What are the limitations of existing approaches?
- How can we overcome them in the new design?

We will first outline the methodology used in this study, then present our findings, including one finding revealing limitations of existing approaches, and two findings guiding the design of our approach.

2.1 Study Methodology

We introduce the criteria for collecting error-handling code snippets, as well as the method to find their error-handling context.

2.1.1 Studied Subjects. As shown in Table 1, we select 22 software systems from 8 domains. We select these projects from GitHub because: a) they span a range of different domains and programming languages; b) they are open-source and well-maintained by the community. These criteria ensure the accuracy and generality of our findings.

2.1.2 Error-Handling Code Snippet Collection. Error-handling code snippets can usually be identified based on return values or special clean functions [14, 19, 20, 25, 31]. For example, a branch statement that returns an error code or calls a cleanup function (e.g., “ENOMEM” in the Linux Kernel, or calls an “exit” function). Existing works consider these special return values and cleanup functions as features of error-handling code snippets. The authors manually specify features to identify and collect such snippets. These features, however, may be program-specific and require extensive human efforts.

In this regard, we propose an automated approach to collect those program-specific features based on error logs (Such as the log function “LOGE(“Device is unauthorized”)” in Fig. 1, not log files). The assumption is that the inclusion of an error log in an error-handling code snippet is relatively random. It means that error-handling code snippets with error logs can serve as random samples of all error-handling code snippets. As such, features of the sampled code snippets should be similar to those of all code snippets. Hence, we merely need to manually specify keywords of error logs and collect a sub-set of error-handling code snippets. Then, our tool learns features of error-handling code snippets from the sub-set, and collects other error-handling code snippets even without error logs. Missing some less common keywords (beyond err, log, etc.) in logging

Table 1. Studied Subjects.

Domain	Name	Line Number	Studied Code Snippets	Different Function
Web server	lighttpd	132,521	81	35 (43.2%)
	Nginx	230,147	158	69 (43.7%)
	Hiawatha	61,941	85	32 (37.6%)
Database	MonetDB	452,349	205	107 (52.2%)
	Sqlmap	104,574	97	32 (33.0%)
	Mysql (twitter fork)	18,396	183	77 (42.1%)
FTP	FileZilla	152,117	106	46 (43.4%)
	Pure-FTPd	32,924	152	74 (48.7%)
	ProFTPD	845,882	186	77 (41.4%)
Image Editor	Darktable	664,703	134	75 (56.0%)
	gThumb	207,107	107	32 (29.9%)
	KolourPaint	73,004	56	14 (25.0%)
Player	Audacious	49,795	52	18 (34.6%)
	MPV	203,794	105	34 (32.4%)
Network Monitor	Netsniff	65,182	97	52 (53.6%)
	Wireshark	5,949,177	210	130 (61.9%)
Distributed Storage	Minio	10,270	34	7 (20.6%)
	Rclone	3,822	34	3 (8.8%)
	Rpm-ostree	49,388	46	19 (41.3%)
	Rook	4,463	37	9 (24.3%)
Development Tool	NetBeans	9,201	64	30 (46.9%)
	gnome-shell	242,440	93	33 (35.5%)

functions does not affect the collection process since we only need a sub-set. This manual effort is limited compared with providing all features.

We consider code snippets that contain special return values or cleanup functions to be error-handling code snippets. We collect error-handling code snippets in three steps. First, we manually search error logs based on keywords such as “err”, “log”, then EH-Digger automatically collects code snippets containing these error logs. Second, for each function call and return value in the above code snippets, EH-Digger calculates the proportion of its occurrences in error-handling code snippets among its occurrences in the entire software. Third, EH-Digger collects other error-handling code snippets without error logs. Specifically, EH-Digger scores branches based on the above proportion of their return values and function calls within, and collects code snippets with scores above a threshold TH_{score} as error-handling code snippets. For example, if the function appears n times in the collected code and m times throughout the software, its score is n/m. The score for a branch is obtained by adding the scores of all functions and return values in the branch. We will evaluate how to set the threshold TH_{score} in Sec 4.3. In this study, the threshold TH_{score} is empirically set to 0.4.

2.1.3 Error-Handling Code Analysis. We conducted a manual analysis of error-handling code snippets. To establish a fundamental understanding, three participants first examined 294 error-handling code snippets in three software systems and described their error-handling contexts. After comparing their descriptions and discussing divergences, the participants analyzed 2,028 error-handling code snippets in the remaining 19 software systems. Each case was discussed by two participants. When they diverged, a third participant was consulted for additional discussions until a consensus was reached. It spent two months analyzing these 2,322 (294+2,028) error-handling code snippets. All three participants had at least three years of programming experience.

2.2 What are the limitations of existing approaches

We first study the limitation of existing approaches. Existing work can be categorized into two classes, one based on manually specified error specification, and one based on learning. The limitation of the first class are clear — they highly depend on the quality of provided error specifications. Existing learning-based approaches usually assume that errors should be handled right after they occur, and learn association rules between API calls and their nearby error-handling code snippets. The capabilities of these approaches are still unknown. Therefore, we study the relative positions between error-handling code snippets and their corresponding fault statements, and find:

Finding 1: 43.3% (1,005/2,322) of error-handling code snippets are located in different functions with regard to their corresponding fault statements.

Taking Fig. 1 as an example, the error occurs in line 1, and its handling (lines 7-10) is in a different function *sc_adb_device_check_state* (line 3 and 5). Percentage of each project are shown in the "Different Function" of Table 1. This finding implies that, in the learning phase, existing approaches can learn just over half of all error-handling code snippets at most, since 43.3% of error-handling code snippets are far from their fault statements (which may or may not be API calls). In the detection phase, existing works usually perform intra-procedural analysis during the bug detection, since an API call and its error handling (if any) are typically in the same function. This finding implies that the intra-procedural approach may result in false positives when dealing with the remaining 43.3% inter-procedural cases.

2.3 How can we overcome them in the new design

We design EH-Digger to address the limitations of existing approaches. EH-Digger learns frequent contexts of code snippets handling the same error, then detects error-handling bugs when the contexts occur without proper error handling. This process is challenging since: a) it is non-trivial to determine if two code snippets are handling the same error; and b) given the same error, it is still hard to mine its frequent contexts which may have various forms. To address these, we conducted two studies to understand the characteristics of errors and contexts accordingly.

2.3.1 Characteristics of Errors. An error-handling code snippet is always guided by a check condition, which determines if the handled error happens. The error is usually stored in a variable (e.g., *state* in line 7 of Fig. 1) of the condition, and we refer to it as *error-prone variable*. This variable may propagate from or to other error-prone variables (e.g., *vec* in line 1, and *device* in line 2, 3, 4). We find that a variable is error-prone either due to its data type or its value. For example, on one side, all variables with data type *FILE* are error-prone, since they may lack certain access permissions or contain null pointers. On the other side, a variable with the basic data type *int* only could be error prone when its value contains certain error semantics (e.g., *chroot* returns `-1` on error, thus the return value of *chroot* is error prone). We study the error-prone variables and find:

Finding 2: 27.4% (636/2,322) and 72.6% (1,686/2,322) of variables are error-prone due to error-prone data types or error-prone values, respectively. Tracing root causes of the error-prone variables in check conditions can determine 93.1% (2,162/2,322) error-handling code snippets handling the same error.

This finding can guide how to determine if two error-handling code snippets are handling the same error. For example, in "$FILE\ file$", the variable $file$ can be replaced using its data type

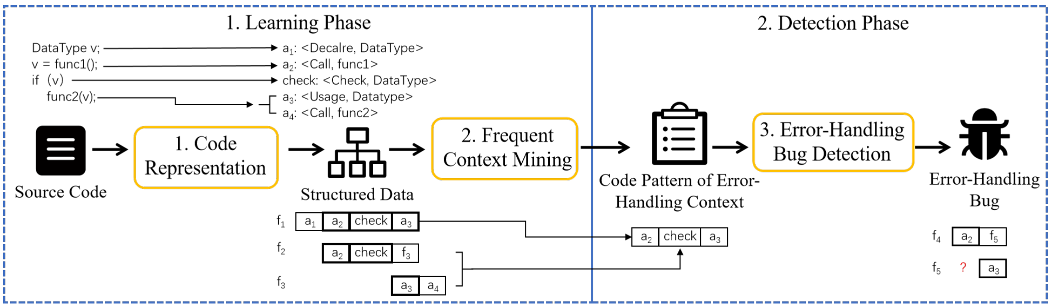


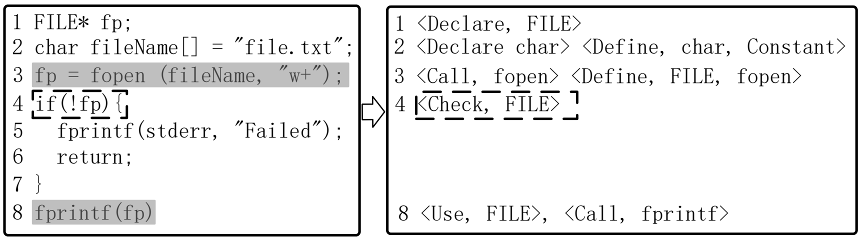
Fig. 2. Overview of EH-Digger.

FILE, while in "*int ret = pthread_create*()", *ret* can be replaced by its value: the return value of *pthread_create*. Such structured replacement removed useless information such as variable names. After such replacement, 93.1% of code snippets handling the same errors will have identical check conditions. Therefore, for the ease of error-handling rule mining, we use the datatype to represent variables. However, 6.9% of errors do not have equivalent checking conditions because they can be checked in multiple ways. For instance, in the Linux Kernel, the function *of_property_read_string* assigns a value to its input parameter, and when encountering an error, it returns an error code. This error can be checked by examining its return value or verifying its input's successful assignment, resulting in non-equivalent check conditions.

2.3.2 Characteristics of Contexts. The context of an error-handling code snippet is a statement sequence from a fault statement generating the error to a failure statement triggered by the error. This sequence can be regarded as a set of *actions* performing on a series of error-prone variables. We find the actions include *declaration*, *definition*, and *usage* of the variables. The declaration action helps to identify the type of a variable from a code snippet that can not be compiled. It enables EH-Digger to determine whether a variable has a basic data type or an error-prone data type from a large-scale code repository that is hard to compile automatically. Besides, the definition and usage actions help to trace the data flow among the serial of error-prone variables based on the Definition-Use Chain. In this regard, we study the actions of error-prone variables and find:

Finding 3: The contexts of error-handling code snippets contain 7.8 actions on average, including 17.2% (3,115/18,111) for declaration, 34.5% (6,248/18,111) for definition, and 48.3% (8,748/18,111) for usage. Besides, 58.4% (1,356/2,322) of the studied contexts can retain the same semantics even when changing the action orders.

This finding implies that EH-Digger could use three actions to represent the error contexts, and should eliminate the impact of order differences when mining frequent contexts and detecting bugs. Actions can be regarded as structured representations of the source code. For example, consider the following code snippets: c_1 = "*int userid*" and c_2 = "*userid = getid*()". c_1 can be represented as $< Declare, int >$, and c_2 can be represented as $< Call, getid >$ and $< Define, int, getid >$. In comparison to the source code, such structured representation removed useless information such as variable names, semantic structure, and is more conducive to subsequent rule mining. For example, if we rewrite c_1 and c_2 as c_3 = "*int userid = getid*()", although the source code is changed, its action list retain the same. As for the order difference, for example, in Fig 3, if we switch line 1 and line 2, the semantics of this code do not change, but it produces a completely different sequence of actions. This makes it possible for codes with the same semantics to have different representations, which is not conducive to rule mining. We will discuss this in detail in Sec. 3.1.



```
1 FILE* fp;
2 char fileName[] = "file.txt";
3 fp = fopen (fileName, "w+");
4 if(!fp){
5   fprintf(stderr, "Failed");
6   return;
7 }
8 fprintf(fp)
```

```
1 <Declare, FILE>
2 <Declare char> <Define, char, Constant>
3 <Call, fopen> <Define, FILE, fopen>
4 <Check, FILE>




8 <Use, FILE>, <Call, fprintf>
```

Fig. 3. Example of code representation.

3 EH-DIGGER DESIGN

In this section, we describe the design of EH-Digger, an error-oriented error-handling bug detection tool guided by error-handling contexts of existing error-handling code snippets.

As illustrated in Fig. 2, EH-Digger takes the source code of software under test as the input, and reports the detected error-handling bugs. It consists of two main phases: the learning phase and the detection phase. During the learning phase, EH-Digger first transforms the source code into structured representations for the convenience of mining rules. After that, EH-Digger obtains contexts of existing error-handling code snippets based on inter-procedural analysis, and extracts frequent contexts from error-handling code snippets that handle the same error as error-handling rules. During the detection phase, EH-Digger reviews the source code using these learned patterns. It reports code snippets containing sequences present in the patterns but lacking corresponding error handling as error-handling bugs.

EH-Digger can analyze target software without the need for compilation. This feature enables EH-Digger to automatically analyze a larger number of software systems from a software repository.

3.1 Code Representation

EH-Digger learns error-handling rules by extracting frequent code sequences, and detects bugs by matching the learned code sequences. However, code snippets with the same semantics may have different syntactic structures. This may affect the error-handling rule extraction as well as the bug detection. Therefore, we first normalize source code into structured representations based on Finding 2 and 3 to reduce extraneous information such as variable names. Subsequently, we order the representations according to their data/control dependencies so that code snippets with the same semantics will have the same representation sequence.

The normalization process of source code contains two phases. In the first phase, EH-Digger scans the source code and converts each statement into actions, including $Declare$, $Define$, Use, $Call$, and $Check$. The first three actions have been discussed in Sec. 2.3.2, while the $Call$ action is used to record call relations required by the inter-procedural analysis in later steps, and the $Check$ action records where the error handling happens. Each action is associated with action-specific information, which can be represented as tuples. Following are the tuples and examples:

- <Declare, Variable>: "*int a*" is represented as <Declare, a>;
- <Define, Variable, Other>: "*int a = b*" is represented as <Declare, a>, <Define, a, b>;
- <Use, Variable, Other>: "*a = b->c*" is represented as <Define, a, b>, <Use, b, c>, <Use, c, ->;
- <Call, Name>: "*a = foo()*" is represented as <Define, a, foo>, <Call, foo>;
- <Check, Variable1, Variable2, ...>: "*if(a + c > b)*" is represented as <Check, a, b, c>.

The first four tuples are straightforward, while the design of the last one is guided by Finding 2. EH-Digger only traces variables in a check condition, and sorts them in alphabetical order. In the second phase, all variables will be normalized according to their data types. On the one hand, for variables with basic data types (e.g., *int*), they could only be error prone due to their value, so EH-Digger traces these variables back to their last assignment and replace them with their assignee. On the other hand, for variables with complex data types (e.g., *FILE*), EH-Digger replaces these variables with their data types. Fig 3 shows an example of code representations.

According to Finding 3, changing the order of some code snippets in the code sequence may not affect the code semantics. As shown in Fig. 3, swapping line 1 with line 2 does not change its semantics. However, it produces a different code sequence for frequent sequence extraction and bug detection. We find that the reason why lines 1 and 2 can be re-positioned without affecting their semantics is that they have no data/control dependencies. Therefore, we sort the obtained representation sequences according to their data/control dependency. We first construct a directed graph based on the data/control dependency. Since topological sorting algorithm [8] can convert a directed graph into a sequence, and ensure that for any directed edge (line 1->3), the source node (line 1) is sorted in front of the target node (line 3), we use it to convert the dependency graph into a code sequence. The topological sorting algorithm iterates the graph and finds nodes with an in-degree of zero. The algorithm puts these found nodes into the sequence, removes them from the graph, and repeats this process until all nodes in the graph have been put into the sequence. However, it only guarantees the order between nodes with dependencies. Taking line 1-3 in Fig 3 as an example, since there is no dependency between line 1 and 2, the topological sorting may result in two different sequences: <1, 2, 3> and <2, 1, 3>. Therefore, we improve the algorithm by adding an additional sorting. In each iteration, we put the nodes obtained into the sequence in dictionary order. It ensures that nodes without dependencies also have a fixed order. After the above sorting, code sequences with the same semantics are transformed into the same representation sequence.

3.2 Frequent Context Mining

EH-Digger learns patterns from contexts of existing error-handling code snippets, and uses the learned patterns for bug detection. EH-Digger first conducts program analysis on the AST to construct data/control dependencies, and extracts contexts of existing error-handling code snippets accordingly. After that, EH-Digger represents obtained contexts using the method described in Sec. 3.1, and mining frequent contexts handling the same error.

The main challenge is that an error-handling context may spread across multiple functions. Thus, EH-Digger has to perform an inter-procedural analysis and avoid the exponential explosion problem in search space. For example, in Fig. 4, given four functions f_1, f_2, f_3, and f_4, each contains a series of actions a_i and calls of other functions. The *check* in f_2 is a check condition of an error-handling code snippet, whose context may involve its callers f_1, f_2 and the callees f_3, f_4. In this case, the context could be "$< a_1$, f_3, *check*, f_4, $a_2 >$". Among the actions, a_1 and a_2 originate from the caller of f_2, which we refer to as the **caller extension**, while a_3, a_4 are referred to as the **callee extension**. In the caller extension, $< a_1 >$ is in the front of f_2. Thus, we refer to it as *caller prefix*, while $< a_2 >$ is referred to as *caller suffix*.

Notice that we did not extend the callee extension (a_3 and a_4) during inter-procedural analysis, this is because through our experiments, collecting caller extension can retrieve complete contexts for more than 83.2% of error-handling code snippets. In contrast, collecting callee extensions can handle the remaining cases, but the efficiency of EH-Digger may drop significantly. In this regard, collecting callee extensions is optional in EH-Digger and disabled by default.

It is non-trivial to collect caller extensions since a function may have multiple and nested callers. EH-Digger uses Algorithm 1 to collect caller prefixes. The algorithm contains a recursive function, `CallerPrefix`, which returns an empty list if the given function f has no caller (line 2-3). Otherwise, for each *caller* of f, EH-Digger first recursively collects the caller prefixes of *caller* itself (*prefix_of_caller* in line 7), then concatenates the action list inside *caller* before calling f

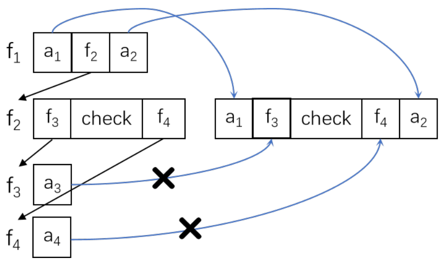


Fig. 4. Example of inter-procedural analysis.

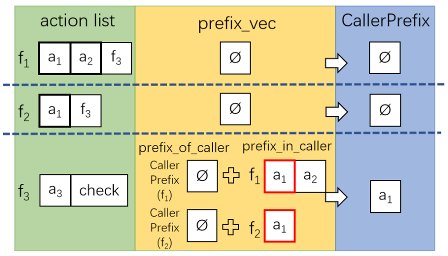


Fig. 5. Example of prefix collection.

(*prefix_in_caller* in line 8-9). Finally, EH-Digger calculates and returns frequent subsequences of the concatenated action lists for all callers (line 11) by using the PrefixSpan algorithm [35].

We give an example in Fig. 5. Three functions are depicted: f_1, f_2, and f_3. The $prefix\_vector$ for functions f_1 and f_2 are empty due to their lack of invocation, resulting in $CallerPrefix(f_1)$ and $CallerPrefix(f_2)$ being empty as well. For function f_3, its interactions with callers f_1 and f_2 are analyzed independently (line 6). Specifically, for f_1, we determine $prefix\_of\_f_1 = CallerPrefix(f_1)$ (line 7), and identify the $prefix\_in\_f_1 =< a_1, a_2 >$ (line 8). Therefore, from f_1, we can obtain $prefix\_vec = CallerPrefix(f_1) + prefix\_in\_f_1 =< a_1, a_2 >$ (line 9). Similarly, for f_2, the analysis yields a $prefix\_vec =< a_1 >$. Upon conducting frequent subsequence mining (line 11), $CallerPrefix(f_3)$ is determined to be $< a_1 >$.

The process of collecting caller suffixes is similar. In this algorithm, caller prefixes and suffixes of all functions will only be calculated once, thus the explosion problem could be avoided.

Algorithm 1 Collect the Action List of Caller Prefix for a Given Function

```
Require: Provide a function $f$ that contains an error-handling code snippet
Ensure: Return the caller-prefix action list of $f$
 1: function CallerPrefix($f$)
 2:     if $f$ has no caller then
 3:         Return an empty list
 4:     else
 5:         Declare an empty vector of action list prefix_vec
 6:         for Each caller in all callers of $f$ do
 7:             Let prefix_of_caller = CallerPrefix(caller)
 8:             Let prefix_in_caller = action list in caller before calling $f$
 9:             prefix_vec.push(prefix_of_caller.extend(prefix_in_caller))
10:         end for
11:         Return frequent_subsequence(prefix_vec)
12:     end if
13: end function
```

Finally, EH-Digger employs the PrefixSpan algorithm [35] again to extract frequent subsequences from error-handling contexts, which are then used as code patterns. Initially, EH-Digger gathers contexts of code snippets that handle the same error based on normalized check conditions. These contexts are then provided as input to PrefixSpan. This algorithm produces all subsequences along with their occurrence frequency. For example, if two action sequences $<a_1,a_2>$ and $<a_1>$ are fed into the algorithm, the algorithm will return: $<a_1>$, 1; $<a_2>$, 0.5; and $<a_1, a_2>$, 0.5. We chose the longest sequence that exceeds an occurrence frequency of TH_f as the code pattern. We will evaluate how to set the threshold TH_f in Sec 4.3. In this study, the threshold TH_f is empirically set to 0.7.

3.3 Error-Handling Bug Detection

Given the previously collected code patterns, EH-Digger can match the patterns in the source code to check if there is proper error handling. For instance, in Fig. 3, our tool will learn the pattern "$<Call, fopen>...<Check, FILE><Use, FILE>''$, the presence of "Call"(line 3) and "Use"(line 8) without"Check"(line 4) indicates a bug. EH-Digger performs inter-procedural checks while ensuring path sensitivity. This step also faces the search space explosion problem. To alleviate this challenge, we adopt a similar method as discussed in Sec. 3.2. Specifically, we perform intra-procedural analysis in each function and store the found consecutive common subsequences of each code pattern. When encountering a function call, we use the stored subsequences to replace the function call instead of performing inter-procedural analysis.

Algorithm 2 Detect Error-Handling Bugs for a Given Code Pattern.

```
Require: PATTERN: the code pattern (e.g., $a_1$, $a_2$, ..., $a_i$, check, $a_{i+1}$, ..., $a_n$)
Ensure: EHB: the set of detected error-handling bugs using PATTERN
1:  Let SEQ = the action sequence in PATTERN (e.g., $a_1$, $a_2$, ..., $a_i$, $a_{i+1}$, ..., $a_n$)
2:  Let CC = the check condition of PATTERN (e.g., check)
3:  GET_INTER_SUBSEQ(root_function)
4:  // Get inter-procedural subsequences of SEQ and detect bugs when matching SEQ
5:  function GET_INTER_SUBSEQ(f) -> (subseq_vec, cc_flag)
6:      Let subseq_vec = f.get_intra_subseq()
7:      Let subseq_vec = f.filter_checked_subseq(subseq_vec)
8:      Let cc_flag = f.has_cc() ? true : false
9:      for Each callee in f.get_callees() do
10:         Let (callee_subseq_vec, callee_cc_flag) = GET_INTER_SUBSEQ(callee)
11:         if callee_cc_flag then
12:             subseq_vec =f.replace_cc(callee).filter_checked_subseq(subseq_vec)
13:             cc_flag |= callee_cc_flag
14:         end if
15:         subseq_vec = subseq_vec.extend_subseq(callee_subseq_vec)
16:     end for
17:     subseq_vec.detect_bug()
18:     Return (subseq_vec, cc_flag)
19: end function
```

We demonstrate the detailed algorithm of our detection method in Algorithm 2. For a given frequent context (e.g., a_1, a_2, ..., a_i, check, a_{i+1}, ..., a_n), the algorithm can find code snippets that contain the action list but lack of the check as error-handling bugs. EH-Digger first obtains the action sequence SEQ (line 1) and the check condition CC (line 2) of the context, then calls the recursive function GET_INTER_SUBSEQ with the root function along the call graph (line 3). GET_INTER_SUBSEQ can: 1) return *subseq_vec* containing a vector of inter-procedural subsequences that continuously match SEQ in the given function; 2) return *cc_flag* indicating if the given function or its callees (an error could be handled in a callee function) contains CC; 3) report an error-handling bug if one element of *subseq_vec* matches the whole SEQ but lacks of CC at the same time.

In the GET_INTER_SUBSEQ function, EH-Digger first collects a vector of intra-procedural subsequences that continuously match SEQ in f (the *get_intra_subseq* function in line 6). When f contains CC, it means f has already performed some error handling. Thus, EH-Digger filters out the subsequences that contain "a_i, check" or "check, a_{i+1}", since these subsequences have already

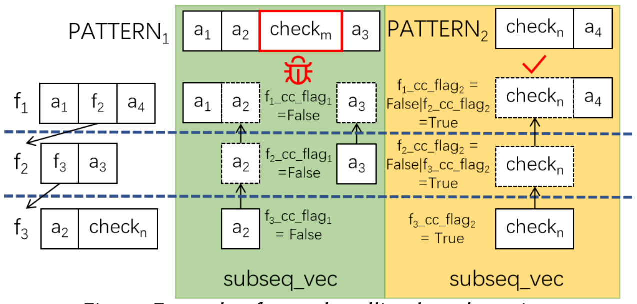


Fig. 6. Example of error-handling bug detection.

been checked (the *filter_checked_subseq* function in line 7). Next, EH-Digger gets *cc_flag*, which will be returned to the caller of f when detecting bugs in the caller (line 8). After that, EH-Digger recursively collects *subseq_vec* and *cc_flag* in all callees of f (line 10). If a callee contains CC (line 11), EH-Digger can replace the callee with CC in f, and perform the filter process similar to line 7 (line 12). This operation is used to handle the case that an error is handled in a callee function. Then, EH-Digger updates *cc_flag* if one of the callees has CC (line 13). In line 15, the *extend_subseq* function extends each subsequence in *subseq_vec* by inserting each subsequence in *callee_subseq_vec* in the place where the callee is called, and the extended subsequences that do not continuously match SEQ will be filtered out. After recursively extending all callees, EH-Digger can detect bugs in *subseq_vec* if a subsequence matches SEQ but lacks CC, and save the results in EHB (line 17). Finally, EH-Digger returns *subseq_vec* and *cc_flag* to the caller of f.

We provide two examples in Fig. 6 for clarity. We analyze three functions: f_1, f_2, and f_3, where f_1 invokes f_2, and f_2 subsequently calls f_3. Our analysis identifies error handling bugs using two patterns: $PATTERN_1 :< a_1, a_2, check_m, a_3 >$ and $PATTERN_2 :< check_n, a_4 >$. For the first pattern, the sequence of actions $SEQ_1 =< a_1, a_2, a_3 >$ (line 1) and the check condition $CC_1 = check_m$ (line 2). Within f_1, the algorithm identifies a subsequence $subseq\_vec\_f_1 =< a_1 >$ (line 6), and proceeds to analyze the function it calls, f_2 (line 9). Similarly, the subsequences $subseq\_vec\_f_2 =< a_3 >$, $subseq\_vec\_f_3 =< a_2 >$ can be obtained from f_2 and f_3, respectively. Following the extension process in line 15, we achieve an updated subsequence for f_1: $subseq\_vec\_f_1 =< a_1, a_2, a_3 >$. Given that $subseq\_vec\_f_1$ encompasses SEQ_a but lacks CC_1, we identify this scenario as an error handling bug (line 17). In the second pattern analysis, f_3 is found to include $check_n$, leading to the $f_3\_cc\_flag_2 = True$ (line 8). Thus, $check_n$ is added to $subseq\_vec\_f_2$ (line 12), and similarly, to $subseq\_vec\_f_1$. Since $subseq\_vec\_f_1$ contains CC_2, we conclude that it does not lead to an error handling bug and remove it via the $filter\_checked\_subseq$ process (line 12).

4 EXPERIMENTS

We conduct experiments to evaluate our approach by answering the following research questions:

- **RQ1:** Can EH-Digger find real-world bugs?
- **RQ2:** Does EH-Digger outperform state-of-the-art approaches?
- **RQ3:** How parameters affect the performance of EH-Digger?

The experiments were conducted on a machine running Linux-18.04 with 64GB of RAM and an Intel i9-10900K CPU.

4.1 Answer to RQ1: Performance of Real-World Bug Detection

In this part, we evaluate the effectiveness of EH-Digger in detecting real-world bugs through experiments conducted on the Linux Kernel and open-source applications.

Table 2. Projects chosen for real-world bug detection.

Domain	Name	Line Number	Sponsor
Operating System	Linux Kernel 6.5	36,780,452	Linux Kernel Organization
	Linux Kernel 4.14	25,041,284	Linux Kernel Organization
Development Tool	Bitkeeper	1,236,529	BitMover
	HandBrake	254,797	Community
	Obs-studio	581,997	NVIDIA, Logitech
Data Transfer Tool	Curl	294,521	Haxx
Monitor	Netdata	698,838	Cloud Native Computing Foundation (CNCF)
Database	Redis	299,660	Redis
Media Player	IJKPlayer	51,495	Bilibili
	Vlc	969,331	VideoLAN
Window Manager	Mutter	520,673	GNOME Foundation
FTP	Bftpd	11,123	Community
Messaging Client	Ayttm	108,894	Free Software Foundation (FSF)

4.1.1 Experiment Setup. As shown in Table 2, we evaluate EH-Digger on the Linux Kernel, and 11 open-source applications from 8 domains. These applications have more than 100 stars on GitHub and are distinct from those in our study. Our evaluation consists of two main parts: detecting new bugs on the latest version and detecting historically fixed bugs in the historical version. Both newly confirmed and historical bugs are equivalent, since the tool is not provided with the bug locations. Our evaluation consists of two main parts. On the one hand, we detect errors in the latest releases and determine whether we have discovered new bugs by confirming with developers. On the other hand, we test on historical releases and compare them with the latest ones to determine if we have identified historical bugs that have been fixed. For the Linux Kernel, we selected its latest release 6.5 and its earliest long-term support release 4.14. For other applications, we chose their latest releases as well as their earliest ones.

4.1.2 Experiment Result. As shown in Table 3, EH-Digger detected 132 bugs in the Linux Kernel 6.5 with a precision of 91.7% (121/132). Out of these, we chose 20 bugs that we were capable of fixing and submitted patches to the Linux Kernel. Currently, all of them have been confirmed by developers. Moreover, in the Linux Kernel 4.14, EH-Digger identified 237 bugs with a precision of 90.3% (214/237), of which 40 bugs have already been fixed in the latest release. We also applied EH-Digger on 11 applications, and the results are summarized in Table 3. EH-Digger detected 182 violations with an average precision of 89.6% (163/182). 31 of the violations were historical bugs. We reported the remaining violations to their respective developers. Up to this point, 33 of these violations have been confirmed, and the others are still under discussion.

The false positives in EH-Digger can mainly be attributed to four factors. Firstly, 18 cases were misjudged by EH-Digger due to the lack of certain contexts. In the learning phase, we ignore the callee extensions. This made a trade-off between context coverage and run-time efficiency to ensure the feasibility of our method. However, such a balance may lead to the omission of certain contexts, resulting in erroneous patterns and subsequent false positives. Secondly, 16 cases misjudged by EH-Digger are caused by incorrect program analysis. Our approach utilizes the AST of the program to analyze its data/control flow. Since the program does not need to be compiled, we are able to analyze a broader range of open-source code automatically. However, compared to compiler-based analysis methods, such as LLVM, the accuracy of our approach during program analysis is lower. Thirdly, some functions are implemented using indirect calls [21, 28], which makes it difficult to determine the corresponding functions by static methods. Since EH-Digger does not consider this

Table 3. Performance in 11 applications.

	Precision	Bug_N	Bug_H		Precision	Bug_N	Bug_H
Linux Kernel 6.5	91.7%	121	0	Linux Kernel 4.14	90.3%	174	40
Bitkeeper	84.0%	14	7	Redis	92.0%	17	4
Curl	80.8%	19	2	Vlc	91.3%	11	8
Obs-studio	94.4%	16	0	Mutter	100.0%	11	3
IJKPlayer	100%	11	2	Bftpd	83.3%	5	5
Netdata	100%	11	0	Ayttm	85.7%	6	0
HandBrake	100%	11	0				

type of call, some contexts are neglected, resulting in 12 false positives. Finally, developers believed 8 bugs did not require handling. We will discuss this in detail in Sec. 5.4.

Result 1: In the Linux Kernel, EH-Digger detected 20 new bugs and 40 historical bugs with a precision rate of around 91%. On 11 applications, EH-Digger identified 33 new bugs and 31 historical bugs, achieving an average precision of 89.6%. These results demonstrate that EH-Digger is effective in detecting real-world bugs.

4.2 Answer to RQ2: Comparison with the State-of-the-art

This section presents a comparison of EH-Digger with state-of-the-art approaches. The evaluation primarily focuses on determining if EH-Digger exhibits higher precision and recall, and if it is able to identify bugs that cannot be detected by existing state-of-the-art approaches. We assessed these three approaches in comparison to EH-Digger on two fronts: precision and recall on customized test sets, and their capacity to identify real-world bugs. Evaluating the precision and recall of error-handling bug detection can be challenging due to the lack of ground truth. Similar to the approach taken in previous studies [49], we manually created the test set by injecting error-handling bugs. It has been proven that general-purpose bug detection techniques cannot be used to detect error-handling bugs [19, 49], so we compare our approach with the latest error-handling bug detection techniques. Furthermore, since large language models have been shown to perform effectively in program repair, we also include them for comparison in our study.

4.2.1 Experiment Setup. For the evaluation on customized test sets, we selected the Linux Kernel to form the test set as it contains a sufficient amount of error-handling code snippets. We randomly removed 30 error-handling code snippets to form a test set. To ensure the accuracy of the results, we constructed 5 different test sets, and took the average precision and recall as the final result. As for testing on real-world projects, we chose the Linux Kernel and 11 applications used in RQ1 4.1.

We compare EH-Digger with two state-of-the-art error-handling bug detection approaches EH-Miner [19], ErrHunter [49], and one large language model GLM [11]. EH-Miner is a state-of-the-art learning-based approach. It identifies functions that are frequently checked by equivalent check conditions, and mines error-handling rules for these functions to detect error-handling bugs. ErrHunter is a state-of-the-art template-based approach designed specifically for the Linux Kernel. It utilizes taint analysis techniques to trace null pointers or general error codes defined within the Linux Kernel, such as "ENOMEM". The primary objective of ErrHunter is to detect error-handling bugs by ensuring the proper handling of these identified features. However, these two approaches have a limitation in that their analysis is restricted to a certain amount of code because they require the code to be compiled. Furthermore, ErrHunter does not make its source code publicly available. Consequently, we evaluated their performance based on their optimal theoretical results. Since

Table 4. Comparison in test sets.

	Precision	Recall	F1-score
EH-Miner	72.3%	70.2%	0.71
ErrHunter	**92.4**%	70.1%	0.80
GLM	62.4%	90.4%	0.74
EH-Digger	91.7%	**72.6**%	**0.81**

EH-Miner is a learning-based method, it needs to be trained on a training set first. To ensure fairness, we deployed EH-Digger on the same training set [19] used by EH-Miner and saved the learned patterns. This training set does not intersect with the test data used in this paper. GLM is one of the latest large language models available for complimentary academic use. We conducted our evaluation using its latest variant, glm-130b. Due to the input limitations of large language models, we cannot feed the entire code of the tested projects to the model. We input the code of a single function at a time, accompanied by the following prompt: "This function is from the Linux Kernel. Are there any error-handling bugs? If so, point out the problem".

4.2.2 Experiment Result. We first compared the precision and recall of the method on test sets, and results are shown in Table 4. The average precision and recall achieved by EH-Digger are 91.7% and 72.6%, respectively. In comparison, EH-Miner reports an average precision of 72.3% and a recall of 70.2%. The recall of EH-Miner is similar to EH-Digger, but its precision is significantly lower. We identified two primary reasons for this outcome. Firstly, during the learning process, EH-Digger traced the error-handling context inter-procedurally. This enabled it to capture the complete context of errors that are not handled where they occur. This reduced the occurrence of learned incorrect patterns and resulted in higher precision and recall. Secondly, during the bug detection process, EH-Miner erroneously identified backward-propagated errors as error-handling bugs. In contrast, EH-Digger traced the error propagation inter-procedurally to find its failure statement, which helped to reduce the incidence of false positives and resulted in a higher precision value. ErrHunter achieved average precision and recall of 92.4% and 70.1%, respectively. In comparison, EH-Digger had a lower precision value than ErrHunter, but a higher recall. This is because ErrHunter employs taint analysis to track whether errors are handled inter-procedurally. Even though it requires the software being analyzed to be compiled, it is more accurate than the AST-based program analysis method used by EH-Digger. Moreover, ErrHunter is designed specifically for the Linux Kernel, where it manually specifies features (null pointers and general error codes defined in the Linux Kernel) are prevalent in the kernel. Thus, although some functions using special error codes cannot be solved by ErrHunter [49], it still achieved a high recall on test sets. However, similar to other template-based approaches, such manually summarized patterns are not easily transferable to other software systems, as we will analyze further in the following experiments. We only provide the function containing bugs to the GLM. The average precision and recall of GLM are 62.4% and 90.4%, respectively. While GLM has a high recall in comparison to EH-Digger, its precision is significantly lower. This difference can be attributed to its predominant intra-procedural analysis. It adds error handling for nearly all functions that might return exceptional values and parameters, resulting in a considerable number of false positives.

Subsequently, we compared the ability of state-of-the-art approaches and EH-Digger in detecting real-world bugs by analyzing the bugs detected on the Linux Kernel and applications.

1) On the Linux Kernel, EH-Digger identified 40 historical bugs and 20 new bugs. Experimental results are presented in Fig. 7. 20 bugs could not be found by EH-Miner. There are two main reasons for this outcome. Firstly, EH-Miner focuses on API calls. Since the Linux Kernel does not use

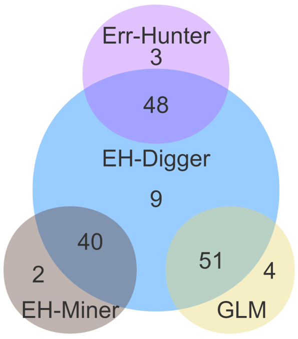


Fig. 7. Comparison in Linux Kernel.

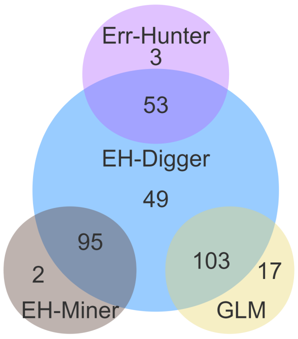
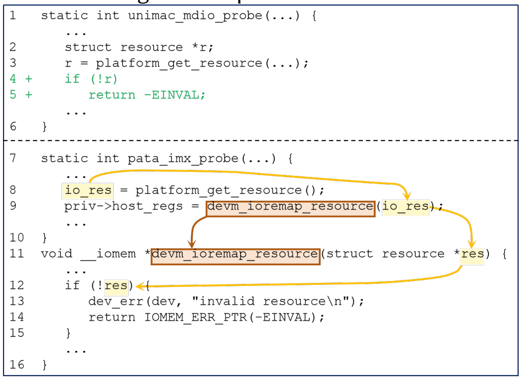
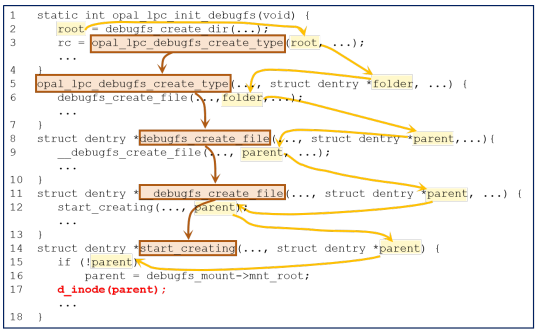


Fig. 8. Comparison in applications.



```
static int unimac_mdio_probe(...) {
   ...
   struct resource *r;
   r = platform_get_resource(...);
+  if (!r)
+     return -EINVAL;
   ...
}
-------------------------------------------------------------
static int pata_imx_probe(...) {
   ...
   io_res = platform_get_resource();
   priv->host_regs = devm_ioremap_resource(io_res);
   ...
}
void __iomem *devm_ioremap_resource(struct resource *res) {
   ...
   if (!res) {
      dev_err(dev, "invalid resource\n");
      return IOMEM_ERR_PTR(-EINVAL);
   }
   ...
}
```

Fig. 9. Examples of false negatives in learning based approaches.



```
static int opal_lpc_init_debugfs(void) {
   root = debugfs_create_dir(...);
   rc = opal_lpc_debugfs_create_type(root, ...);
   ...
}
opal_lpc_debugfs_create_type(..., struct dentry *folder, ...) {
   debugfs_create_file(...,folder,...);
   ...
}
struct dentry *debugfs_create_file(..., struct dentry *parent,...){
   __debugfs_create_file(..., parent, ...);
   ...
}
struct dentry *__debugfs_create_file(..., struct dentry *parent, ...) {
   start_creating(..., parent);
   ...
}
struct dentry *start_creating(..., struct dentry *parent) {
   if (!parent)
      parent = debugfs_mount->mnt_root;
   d_inode(parent);
   ...
}
```

Fig. 10. Examples of false positives in learning based approaches.

any libraries, EH-Miner treats each subsystem (e.g., fs/ext4) as an independent project, considers functions called across multiple projects as API calls, and learns their error handling. This causes functions exclusive to one subsystem to be overlooked and results in 9 false negatives. Secondly, EH-Miner focuses on the handling of a single call and employs an intra-procedural method, which makes it difficult to learn complex error-handling contexts or inter-procedural cases and thus leads to 3 false negatives. Furthermore, we analyzed the bugs detected by EH-Miner, and found that EH-Digger could not identify 2 out of 42 bugs. These 2 false negatives are caused by incorrect program analysis, as EH-Digger traces the data/control flow without compiling the program, making it more susceptible to producing erroneous results than the compile-based approach used by EH-Miner. ErrHunter could identify 48 out of the 60 bugs. The remaining 12 bugs were missed since they were not associated with null pointers or general error codes defined in the Linux Kernel. Since ErrHunter did not open source their code, we can only compare it with the 25 reported bugs mentioned in their paper. 20 of these 25 bugs are resource leak bugs. Such bugs are not the error-handling bugs discussed in this paper, so we only check the remaining 5 bugs. EH-Digger is able to detect 2 of these bugs, as the remaining 3 bugs have never been handled before and therefore cannot be addressed by learning-based approaches. We provided functions with error-handling bugs to GLM. It detected 51 out of the 60 bugs and 4 bugs that EH-Digger failed to detect. These 4 bugs are overlooked by EH-Digger because they are not been handled before. However, because GLM often identifies unchecked parameters or return values as error-handling bugs, it generated 27 false positives while analyzing these 60 functions.

2) On the 11 applications, EH-Digger correctly identified 163 bugs, 30.1% (49/163) of these bugs were not detected by comparative approaches. Results are shown in Fig. 8. EH-Miner only covered 58.3% (95/163) of the bugs detected by EH-Digger, which is caused by the same two reasons we discussed above. Notably, In Bftpd, EH-Miner found 2 more violations than EH-Digger. These 2 violations are missed by EH-Digger due to incorrect program analysis. As for ErrHunter, its performance on these applications significantly declined, covering only 32.5% (53/163) of the bugs detected by EH-Digger. This is because, in most applications, developers often return values such as "False", "-1", instead of specific error codes. Determining which values represent errors is not easy, as the same return value may represent different states. Methods based on manually specified features are difficult to solve such cases, thus the performance of ErrHunter shows a significant degradation. We provided functions with error-handling bugs to GLM. GLM identified 63.2% (103 out of 163) of the bugs detected by EH-Digger and 17 bugs that EH-Digger fail to detect. However, on this provided 163 functions, GLM produced 44 false positives. These false positives could be attributed to its tendency to consider any unchecked return values or parameters as a potential error-handling bug. Such a low accuracy rate also proves that GLM cannot be directly applied to bug detection for the entire software system. If excluding GLM, there are 37.4% (61/163) of bugs found by EH-Digger cannot be found by existing approaches.

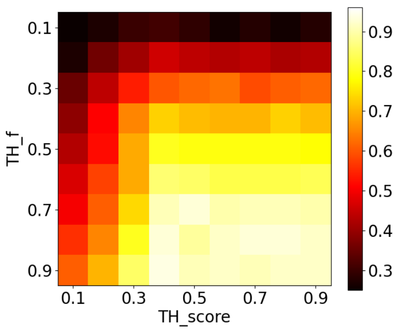
To help understand the contribution of EH-Digger, we conducted case studies on the false positives and false negatives of existing approaches, respectively. On the one hand, Fig. 9 shows a historical bug successfully identified by EH-Digger. This bug is missed by all comparative approaches. The return value of $platform\_get\_resource$ is usually examined and handled in $devm\_ioremap\_resource$. Existing approaches are failed to learn this pattern, and thus miss the bug. On the other hand, the differences between EH-Digger and existing tools are not just inter-procedural and intra-procedural. Existing approaches may learn that there should be an error handling after someplace, but it is hard to know the deadline of the handling (i.e., the handling should be performed before someplace). Fig. 10 illustrates a false positive reported by all comparative approaches. The return value of $debugfs\_create\_dir$ propagates through 4 functions (line 3, 6, 9, 12). It does not need to be handled until d_inode is called. In such a complex situation, existing approaches cannot determine the deadline of error handling, so they usually assume that the error needs to be handled as soon as it occurs, thus often producing false positives.

Result 2: 30.1% (49/163) of the bugs detected by EH-Digger in these applications cannot be detected by any comparative approaches. In the Linux Kernel, EH-Digger outperforms EH-Miner and GLM, and achieves similar performance compared with ErrHunter, which is specifically designed for the Linux Kernel based on template. In other applications, 67.5% (110/163) of the bugs identified by EH-Digger cannot be detected by ErrHunter. EH-Digger can serve as a complementary approach in detecting error-handling bugs.

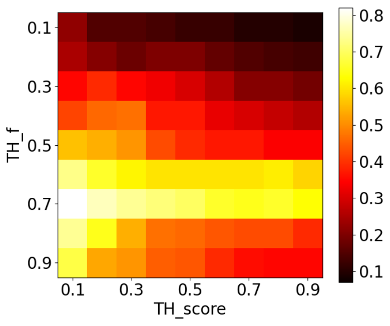
4.3 Answer to RQ3: Impact of Parameters

EH-Miner requires two parameters to be pre-defined, namely TH_{score} (in Sec. 2.1.2) and TH_f (in Sec. 3.2). TH_{score} influences the collection of error-handling code snippets during the learning process, while TH_f affects the extraction of code patterns. Both parameters may impact the performance of EH-Miner. In this section, we evaluate the effect of varying TH_{score} and TH_f on the performance of EH-Digger.

4.3.1 Experiment Setup. We applied EH-Digger to the test set discussed in Sec. 4.1 and evaluated its precision and recall under different parameter values for TH_{score} and TH_f. We iterate the parameter



(a) Precision



(b) Recall

Fig. 11. Precision and recall of EH-Digger under different parameters.

TH_{score} and TH_f from 0.1 to 1 with a step length of 0.1. To facilitate comparison, we prefer an overall metric that considers both precision and recall for ease of comparison. The most common practice is to use the $F1_{score}$, which is the harmonic mean of precision and recall.

4.3.2 Experiment Result. The results are shown in Fig 11. The average precision ranges from 0.07 to 0.82, while the average recall ranges from 0.05 to 0.94. A low value of TH_{score} may result in selecting non-error-handling code segments for learning, leading to a lower precision and recall for the method. Conversely, a high value of TH_{score} may cause some error-handling code snippets to be ignored, resulting in their error-handling contexts not being learned and a decreased recall. TH_f represents the number of cases that support the learned pattern. If TH_f is too low, the learned pattern may be inaccurate, while a value that is too high may cause correct patterns to be filtered out. When TH_{score} is set to 0.4 and TH_f to 0.7, EH-Miner achieves its highest F_{score} 0.81, with the average precision and recall being 91.7% and 72.6%, respectively.

Result 3: With TH_{score} and TH_f set to 0.4 and 0.7, respectively, EH-Digger achieved an average precision of 91.7% and an average recall of 72.6% on test sets. These results demonstrate the effectiveness of EH-Digger in detecting error-handling bugs.

5 DISCUSSION

5.1 Obtaining Error-Handling Code Snippets

EH-Digger considers code snippets that contain error logs as the sampling of all error-handling code. The precision of error log detection may impact the performance of EH-Digger. To this end, EH-Digger supports users to describe the error log using the log function name and the keyword of the log content. Since error logs in software usually follow a fixed format, the use of such a description method can yield accurate results. Furthermore, we sampled 20 functions as the test set, and repeated the test in 3 different applications. By using no more than 6 keywords provided by the user, EH-Digger successfully identified an average of 66.4% of all error-handling code snippets. After learning their distinctive features (Sec. 2.1.2), EH-Digger achieved an average detection rate of 83.9% for all error-handling code snippets. While more advanced NLP techniques could potentially enhance the number of identified error-handling code snippets, we adopted this simpler approach to reduce method complexity and improve runtime efficiency.

5.2 Constructing Data-Flow and Control-Flow Dependencies

To trace error-handling context and detect error-handling bugs, EH-Digger conducts program analysis on the AST to construct data/control-flow dependencies. This design allows EH-Digger to analyze code without the need to compile the target project, and thereby can be applied to more projects automatically. This approach may generate more erroneous results compared to tools requiring compilation, such as LLVM. This is a trade-off between precision and scalability. According to the results of our experiments in Sec. 4.1 and Sec. 4.2, the number of mistakes caused by program analysis is comparatively low (16 false positives and 2 false negatives) since our learning-based method can tolerate a limited number of erroneous results. At present, EH-Digger is implemented exclusively for C/C++.EH-Digger employs tree-sitter [5] to obtain the AST of the code. Since tree-sitter supports more than 20 programming languages, EH-Digger can be easily adapted to other programming languages with minimal code modifications.

5.3 Tracing Error-Handling Context

As inter-procedural methods are prone to space explosion, we only collect caller extensions when tracing the error-handling context. This may lead to the omission of certain contexts. To assess the impact of this omission, we conducted an investigation into error-handling contexts. Our results show that EH-Digger can retrieve complete contexts for more than 83.2% of the error-handling code snippets. In contrast, to obtain the complete context of the remaining 16.8%, the execution efficiency of EH-Digger will drop significantly, sometimes by hundreds of times or even unfeasible. The results in Sec. 4.1 also demonstrate that omitting these contexts has a negligible impact on EH-Digger performance (causing 18 false positives as discussed in Sec. 4.1.2). Therefore, to prioritize execution efficiency and maintain feasibility, we decided to discard these contexts.

5.4 Oracle of Detecting Error-Handling Bug

We label code snippets as buggy if they contain action sequences from learned patterns but lack handling (Sec. 3.3). There are 2 possible error scenarios for such an oracle. First, the learned pattern may be incorrect. This situation is usually due to the trade-off we made during the learning phase (as discussed in Sec. 5.3) or an error in static analysis. In our experiments, this case is considered as a false positive. Specifically, these accounted for 3 out of 16 false positives attributable to static analysis errors (Sec. 4.1.2) and 18 false positives attributable to the trade-off. This low incidence rate is a testament to our method's robustness in pattern mining, which leverages the mining of frequent subsequences to ensure that even when irrelevant actions are included, their influence on the pattern remains minimal. Second, developers may think that some error handling is unnecessary. For example, in situations deemed critical—such as a program running out of memory—developers might opt to restart the system rather than writing error-handling code, believing it to be the most effective solution. Conversely, in cases involving simple code snippets with limited input, such as examples or pre-processing scripts, developers may also view error handing as superfluous. During the process of submitting issues to developers, we identified 8 instances that fell into this category.

6 RELATED WORK

6.1 Error-Handling Bug Detection

Existing approaches of error-handling bug detection can be classified into two classes: approaches based on manually constructed templates/error specifications, and learning-based approaches. For the first class, these approaches usually begin with a study from a particular perspective, summarize common categories, and design patterns accordingly. Tian and Ray. [42] and Jana et al. [16] classified error-handling bugs and design templates accordingly. Pakki et al. [34] studied bugs where the error is handled over-severe, Wu et al. [46] studied disordered error handling, while Li et al. [26] summarized the frequent error handling of SSL library. Some existing approaches [40, 47–49] focused on more specific features, such as null pointers, special error codes, and user inputs. The above approaches require significant domain knowledge, and are hard to accommodate the software evolution. Conversely, learning-based approaches usually use the existing error-handling code snippets to derive templates or error specifications. Some existing approaches [1, 26, 45, 50] examined differences between normal paths and error paths using features such as the length and return expressions. Similarly, Zhong et al. [51], DeFreez et al. [9], Liu et al [27], and Lu et al. [29] also predict the error branch. Jia et al. [19] focused on the equivalence of check conditions, and Shen et al. [39] concentrated on particular functions and pointers. Although these approaches have achieved decent results, they mainly learn API calls near error-handling code snippets. As discussed in Sec. 2.2, errors may not be handled where they occur, making it hard for existing learning-based approaches to accurately establish the correlation between the error and its handling code snippets.

6.2 Exception-Handling Bug Detection

There is a long line of research that focuses on exception-handling mechanisms [3], which are built-in features in many programming languages, such as Java and Python. Existing approaches focus on this problem from two perspectives: whether a correct exception is thrown, and whether the thrown exceptions are properly handled. For the former, Jia et al. [18] studied problems caused by ungraceful exits. ExAssist [32] recommends repairing actions based on machine learning model. Bouzenia et al. [4], Zhong [50], and Chen [6] studied the inconsistency of the error with the thrown exception. Weimer et al. [43, 44] presented a data-flow analysis for finding whether a certain type of exception is properly handled. For the second class, Oliveira et al. [33] conducted an empirical study on the relationship between the usage of Android abstractions and uncaught exceptions. Gu et al. [12] expanded the intrinsic capability of runtime error resilience in software systems. Barbosa et al. [2] presented a tool to recommend repairs with awareness of the global context. Yan et al. [24, 30] improved fault localization effectiveness based on a slice-based approach. These works focus on exception-handling mechanisms like try-catch statements, while EH-Digger detects bugs for error handling of variable constrain violation, which is widely used in languages like C/C++.

7 CONCLUSION

Existing learning-based approaches on error-handling bug detection only learn API calls near error-handling code snippets, which makes them hard to learn the real reasons for the error-handling code in many cases. To address such problems, we propose EH-Digger, an error-oriented approach that learns from the error-handling context of existing error-handling code snippets and detects error-handling bugs accordingly. We applied EH-Digger to the Linux Kernel and 11 mature applications. EH-Digger detects error-handling bugs with a precision of 91.7%. It detected 20 new bugs and 40 historical bugs in the Linux Kernel, 33 new bugs and 31 historical bugs. 30.1% bugs detected by EH-Digger cannot be detected by state-of-the-art approaches. It can serve as a valuable complementary approach in detecting error-handling bugs.

8 DATA AVAILABILITY

The source code and dataset can be found in the repository:
https://github.com/EH-Digger/EH-Digger.

ACKNOWLEDGMENTS

This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No.2023RC1001) and NSFC No.62202474.

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] Eiji Adachi Barbosa and Alessandro Garcia. 2018. Global-aware recommendations for repairing violations in exception handling. In *Proceedings of the 40th International Conference on Software Engineering*. 858–858.

[3] Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai. 2018. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering* 45, 10 (2018), 984–1001.

[4] Islem Bouzenia. 2022. Detecting Inconsistencies in If-Condition-Raise Statements. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–3.

[5] M. Brunsfeld. 2023. Tree-sitter. https://tree-sitter.github.io/tree-sitter/ Accessed 1. October 2021.

[6] Haicheng Chen. 2021. *Combating Fault Tolerance Bugs in Cloud Systems*. The Ohio State University.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. 1–5.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[9] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 466–476.

[10] Daniel DeFreez, Antara Bhowmick, Ignacio Laguna, and Cindy Rubio-González. 2020. Detecting and reproducing error-code propagation bugs in MPI implementations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 187–201.

[11] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. (2022), 320–335.

[12] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic runtime recovery via error handler synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 684–695.

[13] 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.

[14] Foyzul Hassan, Chetan Bansal, Nachiappan Nagappan, Thomas Zimmermann, and Ahmed Hassan Awadallah. 2020. An empirical study of software exceptions in the field using search logs. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[15] Benjamin Jakobus, Eiji Adachi Barbosa, Alessandro Garcia, and Carlos José Pereira De Lucena. 2015. Contrasting exception handling code across languages: An experience report involving 50 open source projects. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 183–193.

[16] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications.. In *USENIX Security Symposium*. 345–362.

[17] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.

[18] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, and Ji Wang. 2019. Automatically detecting missing cleanup for ungraceful exits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 751–762.

[19] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. 2019. Detecting error-handling bugs without error specification input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 213–225.

[20] 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*. 472–482.

[21] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level.. In *NDSS*.

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

[23] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. 2010. Finding error handling bugs in openssl using coccinelle. In *2010 European Dependable Computing Conference*. IEEE, 191–196.

[24] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How test suites impact fault localisation starting from the size. *IET software* 12, 3 (2018), 190–205.

[25] Chi Li, Min Zhou, Zuxing Gu, Ming Gu, and Hongyu Zhang. 2019. Ares: Inferring error specifications through static analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1174–1177.

[26] Chi Li, Min Zhou, Xinrong Han, and Ming Gu. 2021. Sensing Error Handling Bugs in SSL Library Usages. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 686–692.

[27] Huqiu Liu, Yuping Wang, Lingbo Jiang, and Shimin Hu. 2014. PF-Miner: A new paired functions mining method for Android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 33–42.

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

[29] Kangjie Lu Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Conference on Security Symposium*.

[30] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[31] Paul D Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 379–388.

[32] Tam Nguyen, Phong Vu, and Tung Nguyen. 2019. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 390–393.

[33] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1–18.

[34] Aditya Pakki and Kangjie Lu. 2020. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1203–1218.

[35] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering* 16, 11 (2004), 1424–1440.

[36] Genymobile R. Vimont. 2023. Scrcpy utility. https://github.com/Genymobile/scrcpy Accessed 1. March 2023.

[37] Martin P Robillard and Gail C Murphy. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*. 2–10.

[38] 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 *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–280.

[39] Qintao Shen, Hongyu Sun, Guozhu Meng, Kai Chen, and Yuqing Zhang. 2023. Detecting API Missing-Check Bugs Through Complete Cross Checking of Erroneous Returns. In *International Conference on Information Security and Cryptology*. Springer, 391–407.

[40] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.

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

[42] 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*. 752–762.

[43] Westley Weimer. 2004. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. 419–431.

[44] Westley Weimer and George C Necula. 2008. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–51.

[45] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating precise error specifications for c: A zero shot learning approach. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[46] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and detecting disordered error handling with precise function pairing. In *the 30th USENIX Security Symposium (Security'21)*.

[47] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 512–523.

[48] 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*. 499–510.

[49] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. 2022. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2022), 684–698.

[50] Hao Zhong. 2022. Which Exception Shall We Throw?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[51] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

Received 2023-09-28; accepted 2024-04-16