Identifying Error Code Misuses in Complex System*

Wensheng Tang
The Hong Kong University of Science and Technology
Hong Kong, China

ABSTRACT

Many complex software systems use error codes to differentiate error states. Therefore, it is crucial to ensure those error codes are used correctly. Misuses of error codes can lead to hardly sensible but fatal system failures. These errors are especially difficult to debug, since the failure points are usually far away from the root causes. Existing static analysis approaches to detecting error handling bugs mainly focus on how an error code is propagated or used in a program. However, they do not consider whether an error code is correctly chosen for propagation or usage within different program contexts, and thus miss to detect many error code misuse bugs. In this work, we conduct an empirical study on error code misuses in a mature commercial system. We collect error code issues from the commit history and conclude three main causes of them. To further resolve this problem, we propose a static approach that can automatically detect error code misuses. Our approach takes error code definition and error domain assignment as the input, and uses a novel static analysis method to detect the occurrence of the three categories of error code misuses in the source code.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; *Software maintenance tools*; *Software verification and validation.*

KEYWORDS

error code handling, software development, static analysis

ACM Reference Format:
Wensheng Tang. 2019. Identifying Error Code Misuses in Complex System. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338986

1 INTRODUCTION

The sophisticated software infrastructure has brought a large number of error scenarios that need to be correctly handled. Software practitioners design the error codes (or status codes) as an explicit reminder of choices on error handling. For simplicity, error codes are typically constant numbers that stand for some error scenarios. For example, error code `404` refers to the "not found" error in the HTTP/1.1 protocol. Error codes are commonly used in both open source projects and large commercial software systems. Some complex systems with millions of lines of code, such as Windows, IBM DB2, and Chromium browser, even use thousands of error codes [5, 7, 11]. However, programmers often forgot to handle the error code correctly, and this leads to severe problem. For example, error code misuse is one of the common root causes of many CVE security vulnerabilities [1, 2]. It is also a major security issue listed in in the OWASP Top 10 [12].

```
1  int ExitCode = api_call();
2  if (ExitCode == OK) {
3      return ExitCode;
4  }
5
6  return OK;   // All other error codes fall to OK
```

Figure 1: An example of simplified code snippet of error code misuses that we detected in a *commercial* system

Figure 1 gives an example of error code misuse problem. The error code returned from an API function call `api_call()`. However, due to inadequate check of the error code at line 2, all error codes other than `OK` are mapped to `OK`. These mappings cause the further caller of current function loses error signal. As a result, unresolved errors may lead to potential instability issues. Some of the misuses can cause visible effect, such as memory corruption errors, which can be vulnerable to denial-of-service attack. Another common result is non-visible effect, such as logic errors, but it never triggers any crashes or other abnormal program behaviors. These underlying problem can lead to continuous damages to a system. Therefore, such mistakes should be avoided to ensure the robustness of a software system.

Although many studies are proposed to detect error handling bugs [6, 8, 14, 16], they are still ineffective. This is because, these studies mostly focus on analyzing how an error code is propagated or used, but ignore a fundamental problem – whether an error code is correctly chosen for propagation or usage under a given program context. Due to the lack of understanding on error code, it is challenging to make a decision for the aforementioned problem. Therefore, in this study, we propose to conduct the first systematical empirical study on error code handling in a mature *commercial* system[1]. This system frequently generates error codes to differentiate error scenarios. It also makes error code conversion frequently from each program part. We make a classification on error code misuses based on real bug reports and manual inspection of code

*This work was partially funded by Hong Kong GRF16214515, GRF16230716, GRF16206517, NSFC61628205 and ITS/215/16FP grants.

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338986

[1]Due to confidentiality agreement, we can neither mention the company's name nor the code size of this commercial system.

snippets related to error codes. Our findings indicate that error code misuse problem has three representations in general: Initialization of Incorrect Error Code (IC), Missing or Incorrect Log of Error Code (ML/IL), Incorrect Error Code Mapping (IM). We also notice that both large open-source projects and commercial systems divide error codes into error domains that cover different program parts. Besides, error codes often map from an error domain to another, which conforms to Guanwi et al. 's observations [6]. This observation enables us to automatically extract semantics to more precise error specifications since each domain has already indicated its governed feature.

Inspired by these observations, we design a comprehensive approach to conveniently specify the error code use rules and identify the three categories of error code misuse problems. We first perform value flow analysis to collect precise error code uses concerning its domains. However, since different projects have their choices on error code uses, we create a universal rule grammar to describe the error code use rule and use it as our generic inputs. To further determine the correctness of each error code use, we summarize the actual use of error codes into data entries and then compare the rules with the error code rule grammar to verify the correctness of each error code use site.

2 EMPIRICAL STUDY

To understand the error code misuse problems, we first conduct a manual study on a commercial system written in C++. We use a natural language processing (NLP) technique to find error code related fixes. We remove the tense and perform regular expression-based keyword searches in the pre-fetched commit messages. For example, "`fix`" and "`resolve`" indicate the commit is an error code related fix.

To evaluate the effectiveness of this approach, we randomly walk through 150 commits over hundreds of the search results and find that this regular expression can achieve 100% precision on discovering error code related fix. Because we use verbs as indicators to discover fix related commits, many other fix related commits can be missed. As such, we start a second round of searches without these verbs. By filtering previously discovered fix commits, we further manually inspected another 150 random commits by this method. The result shows that only 23 of 150 belong to the error code problem fix. We notice that the reason for imprecision is mainly caused by some feature changes that also bring changes to error code definitions. For example, the commit message "added error codes" do have words "error code". However, it shows that some feature updates bring new error codes into the code but it is not a fix related commit.

Table 1: Distribution of Each Problem in Study Projects

Category	Bugs	%
Initialization of Incorrect Error Code (IC)	26	15.0%
Missing or Incorrect Log of Error Code (ML/IL)	135	78.0%
Incorrect Error Code Mapping (IM)	12	6.9%

Characterization of the Error Code Misuse Problem. We manually investigate 300 commits from the study and find 173 true

```
1  int Credit::api() {
2      // ...
3      /* Correct Init */
4      return ERR_CREDIT_X;
5  }

6  int Cash::api() {
7      // ...
8      /* Incorrect Init */
9      return ERR_CREDIT_X;
10 };

11 int business_logic() {
12     int ret = Cash::api();
13     if (ret == ERR_CASH_X) {
14         // Log("Err", ERR_SYS_CASH);/* Missing Log */
15         return ERR_SYS_CASH;
16     }
17     // ...
18     ret = Cash::api();
19     if (ret == ERR_CASH_X) {
20         Log("Err", ERR_CASH_X);   /* Incorrect Log */
21         return ERR_SYS_CREDIT;/* Incorrect Mapping */
22     }
23     return OK;
24 }
```

Figure 2: Example of Error Code Misuse Problem

bug fixes related to the error code misuse problem. We inspect the code patches to understand their root causes and classify them into three categories based on the inspection: Initialization of Incorrect Error Code (IC), Missing or Incorrect Log of Error Code (ML/IL), Incorrect Error Code Mapping (IM). We show the numbers of fixes in each category in Table 1 and show examples simplified from real patches of each category in Figure 2.

Initialization of Incorrect Error Code (IC). To correctly represent the status of a program error, the appropriate error code shall return to the caller. However, we find that developers often forget to set the correct error code. By comparing with the original code, we find that this problem is often caused by copy-and-paste code from another code snippet that has a similar code structure. Figure 1 has shown the example the `Credit::api()` function has returned the correct error code `ERR_CREDIT_X` at Line 5, since the `class Credit` requires the user functions use error code in the same error domain prefix `ERR_CREDIT`. Similar requirements are for `class Cash`. However, the function `Cash::api()` returns an error code from another domain. It will further cause the mishandling of such an erroneous case. In our study, this type of error code misuse problem counts 15.0% of the bugs in total in Table 1.

Missing or Incorrect Log of Error Code (ML/IL). Error codes not only record the status of program termination but also represent the status of intermediate result at runtime. Hence, many systems require logging of a specific error code and its description for debugging and monitoring purpose. However, missing report of current status makes the monitoring service fail to alarm the error status. Similarly, misreported error codes in the log messages can bury real problems at runtime and cost more for the developer to diagnose the bug. For example, at Line 14, a log function shall present to record the error state if the project requires the of intermediated states. At Line 20, this mis-logged error code `ERR_CREDIT_X` can be misleading to project maintainer because only the last returned error code is to log. If the error code is mis-logged, it is troublesome to diagnose where produces the error from the logs. As per Table 1, this type contributes the most in the studied bugs (78.0%).

Incorrect Error Code Mapping (IM). Error codes usually map between different program parts to refine the error states for better

code management. Moreover, many open-source projects and pro systems convert error codes from third-party libraries to their error code format. The translation between error codes must be correct; otherwise, it can cause mishandling of the errors and even unexpected program behaviors. In Figure 2, the code ERR_CASH_X maps to ERR_SYS_CASH at Line 13-16. However, another code ERR_CASH_X also maps to the code ERR_SYS_CREDIT. These divergent mapping of error codes may cause inconsistent program behavior when the same program error occurs due to different error handlers of callee function. This category contributes only 6.9% of the bugs (in Table 1).

3 METHODOLOGY

Our approach utilizes four kinds of inputs from either document and hand-written rules. We first use the knowledge from error domain to collect the uses of error codes together with their error domains, by perform a value flow analysis. Then, we use rules to detect the violations in error code uses. The workflow of our approach is in Figure 3.

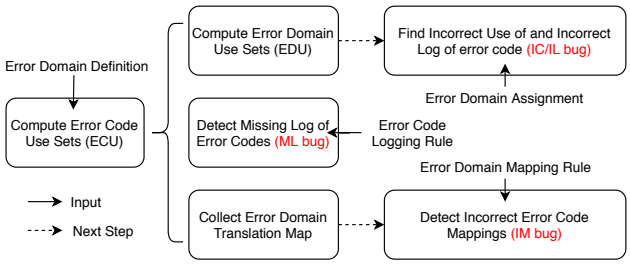

Figure 3: Workflow of Error Code Misuse Detection

3.1 Error Code Use Rule

The use of error codes follow certain specifications as such we can utilize the rules as input to detect misuse bugs. We design a universal error code rule grammar. the rules as prerequisites for our approach in error code misuse bug detection. The rule grammar are the combination of four sub-rules as shown in Figure 4.

$$\textit{Error Domain Definition } \mathcal{D} : E \xrightarrow{<prefix>} D \quad (1)$$

$$\textit{Error Domain Assignment } \mathcal{A} : C \longrightarrow D \quad (2)$$

$$\textit{Error Code Logging Rule } \mathcal{L} : \ell_e \xhookrightarrow{call\ \texttt{LOG(e)}} \texttt{ret e} \quad (3)$$

$$\textit{Error Domain Mapping Rule } \mathcal{T} : D_i \xrightarrow[[f_{API}]]{} D_j, \quad i \neq j \quad (4)$$

Figure 4: Error Code Use Rule

3.1.1 Error Domain Definition. The term of error domain originally comes from Apple. We further extend the its definition: An error domain d is to govern the specific program part to use sets of error code e_i from that domain only. As per project settings, the error domain are defined from naming rules of error codes, including enumerated type name, object name or the same macro prefixes (e.g. ERR_). In this work, the assumption is to use the enumerated type name as error domain indicators because our studied system uses it. We use a map $\mathcal{D}(e) = d$ to record an error code e belongs to an error domain d. For example, the $\mathcal{D}(\texttt{ERR\_CASH\_X}) = d_{credit}$ means the error code ERR_CASH_X belongs to error domain d_{credit} in Figure 2 because it has the prefix ERR_CASH. Note that, the length of prefixes is specified per each project's setting.

3.1.2 Error Domain Assignment. The purpose of the error domain design is used to confine the error code used by specific program parts. Projects director defines whether the error domains assign to either classes, functions, directories or other types of program parts. In our evaluated system maps error domain to classes, our setting is the assignment between classes and error domain, denoted as $\mathcal{A}(c) = d$ for each class c. For example, $\mathcal{A}(\texttt{class Credit}) = d_{credit}$ in Figure 2.

3.1.3 Error Code Logging Rule. Depending projects require the developers to record the error code before a function returns. This observation also indicates the second category (ML/IL) of misuse problems exists only if the error code is required to be logged. From a VFG node of error code e's initialization ℓ_e to the return statement of an error code ℓ': `ret e`, the value flow path is $\ell_e \rightarrow \ell'$. There must be another path from ℓ_e to a logging statement ℓ'': `call LOG(e)`, which records the correct error code e. Otherwise, either the error code e is not logged or it is incorrectly logged. The rule is represented as $\mathcal{L}$ in Figure 4(2).

3.1.4 Error Domain Mapping Rule. From the observations of previous work [6] and ours, the error codes map from one to another. We further notice that error code mappings does not occur randomly. Typically, one error code does not map to multiple error codes in another error domain. The rule can be encoded as mapping function: for any pair of error codes $\langle e_1, e_2 \rangle \in D_1 \times D_2$, there may or may not exists a mapping $e_1 \mapsto e_2$. For example, at Line 13-16 the error code ERR_CASH_X from domain d_{credit} is mapped to ERR_SYS_CASH from domain d_{sys}. We also find that this mapping usually happens in particular functions that interact with API functions. For example, only a small group of functions will call the third-party library function and convert the library's error codes. As such, we make our mapping rule function $\mathcal{T}$ (in Figure 4(4)) to be a conditional function that only permits inside certain API function f_{API}. If there is no such restriction, the f_{API} is any function $f \in F$, where F is the universe of all functions in a program.

3.2 Analysis Algorithm

3.2.1 Value Flow Graph. The value flow graph (VFG) is a well-studied data structure that encodes the value propagation [3, 10, 15]. In this stage, we use the value flow graph to encode error code propagation. The procedure construction of the value flow graph is similar to FastCheck[3]. In addition, inspired by SMOKE[4], two kinds of indicator nodes are added to VFG: if the value of error code has been checked, i.e. the error code flows to a `cmp` statement, we create an edge from the definition of an error code to a pseudo node *checked*; if the value of error code is reaching the end of the function, we also create another edge from current definition to an *unchecked* node.

3.2.2 Error Code Use Set and Error Domain Use Set. The prerequisite of determine the correctness of an error code use is the knowledge of possible returned error codes by a function f. For each error code $e \in E$, we run global value flow analysis to get possible returned or logged error codes for each function f, named the error code use set $ECU(f)$. By the naming rules of error domain, we have the mapping function $\mathcal{D}$ from an error code e to an error domain d. We can further discover the error domain(s) used by a function f, named error domain use set $EDU(f) = \bigvee_{e \in ECU(f)} \mathcal{D}(e)$.

3.2.3 Error Domain Translation Map. While building value flow graphs, we also identifies the error handler structure that translates the error code from one to another. For example, line 13-16 is an error handler code snippet that has mapped error code `ERR_CASH_X` to `ERR_SYS_CASH`. We save these mapping $e_1 \xrightarrow[[f]]{} e_2$ to error domain translation map EM. Besides, we also stores $e \rightarrow null$ to EM if there is a path $\ell_e \rightarrow unchecked$ on VFG for each error code e.

3.2.4 Detection of Error Code Misuses. We discuss the detection approach specific to each category of the error code misuse problem on the value flow graph by the rules in Section 3.1.

Initialization of Incorrect Error Code (IC). From Section 3.2.2, we can know the possible returned error codes for function f as $ECU(f)$. To further investigate if error code correctly represents the semantics of program parts, we further compute domains that a function f has used, denoted as $EDU(f)$. Recall that the input rules $\mathcal{A}$ suggest that a function f can belongs to at most one error domain in Figure 4(2). If $EDU(f)$ does not only include $\mathcal{A}(c_f)$, then the semantics of the error code usage in that program part is incorrect and we report it as an IC bug. For example, we have $EDU(\texttt{Cash::api()}) = d_{credit}$ and $\mathcal{A}(\texttt{Cash}) = d_{cash}$ in Figure 1 and $d_{credit} \neq d_{cash}$, we can report the error code initialization at Line 9 is an IC bug.

Missing or Incorrect Log of Error Code (ML/IL). For detecting mis-logged error code (ML) bugs, our approach is similar to the detection of IC bugs since $ECU(f)$ also stores the logged error code. If the error code logging rule $\mathcal{L}$ exists, the detection of unlogged bugs, similar to Smoke[4]'s memory leak detection process, we only collected the constraints from paths from an error code e to the end node *checked* or *unchecked* that has never passed through log statement ℓ : `LOG`(x) and verify with a constraint solver. If the constraint evaluates to *true*, we report it as a UL bug candidate.

Incorrect Error Code Mapping (IM). For detecting incorrect error code mapping (IM) bugs, we compare the error domain translation map EM and the error domain mapping rule $\mathcal{T}$. For each error code e, there is a mapping $e \xrightarrow[[f]]{} e_i$ in the translation map EM and another mapping from $e \xrightarrow[[f_{API}]]{} e_j$ from the mapping rule $\mathcal{T}$. If two mappings happen in different functions ($f \neq f_{API}$), we report it as an IM bug because it violates the error code mapping rule $\mathcal{T}$. For example, line 23-26 in Figure 2, `ERR_CASH_X`→`ERR_SYS_CREDIT` is inconsistent of `ERR_CASH_X`→`ERR_SYS_CASH` in $\mathcal{T}$. Besides, for each error code e, if there are two mappings $e \rightarrow e_i, e \rightarrow e_j \in EM, i \neq j$ in the translation map EM , we also report it as an IM bug since it violates the many-to-one mapping property of the function $\mathcal{T}$. Moreover, for each $e \rightarrow unchecked \in EM$ which indicates the error code e is neither handled or mapped to another but silently ignored or overwritten, we also report this as an IM bug.

4 EVALUATION PLAN

To evaluate our approach, we design experiments to address the following research questions:

RQ1 How effective is our approach in detecting error code misuses?
RQ2 How well does our approach scale to large-sized programs?
RQ3 What are the precision and recall of our approach when compared with the state-of-the-art approach?

To answer RQ1, we plan to evaluate our approach on several open source projects to check whether it can detect real error code misuses. To answer RQ2, we plan to evaluate it on large-sized programs such as Linux Kernel, MySQL, and Apache HTTPD to see whether our approach can finish analyzing large projects under reasonable resource constraints. To answer RQ3, we plan to evaluate the precision and recall of our approach by comparing it with the state-of-the-art error handling bug detector, ErrDoc[16] with its error specification miner APEx[9].

5 RELATED WORK

Some research efforts have been made for detecting bugs related to error code misuses statically. EIO [6] investigates the error propagation bugs in Linux file systems, and proposes a static analysis technique, EDP, based on a data-flow analysis, to detect them. Rubio-González et al. [14] 's approach achieves a better precision by employing the WPDS framework [13] that is both flow- and context-sensitive. Recently, some tools like EPEx[8] and ErrDoc[16] use statistic method[9] to learn error code specifications from source code and verifies them against the source code to find error code misuses. However, existing approaches share several limitations that prevent them from detecting more error code misuses. First, their methods do not track the mappings among error codes, which are essential for analyzing error states that across different system components. Our approach uses a light-weight sparse value flow analysis to track the value propagation and evaluates them according to their mapping rules. Second, existing approaches either do not consider the error code specifications[6, 14] or only verify patterns extracted from code[8]. Our approach, on the other hand, takes the error code specifications as input and detects violations of those specifications in the source code. Our approach uses a novel static analysis to extract mappings among different error domains from the source code and then detects errors from the inconsistencies in the extracted mappings and the differences when compared with the input mappings.

REFERENCES

[1] 2016. CVE-2016-6662. Available from MITRE, CVE-ID CVE-2016-6662.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6662
[2] 2018. CVE-2018-8099. Available from MITRE, CVE-ID CVE-2018-8099.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8099
[3] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/1250734.1250789
[4] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*.

[5] Google. 2019. Error Codes - Chromium. https://cs.chromium.org/chromium/src/net/base/net_error_list.h

[6] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*. USENIX Association, Berkeley, CA, USA, Article 14, 16 pages. http://dl.acm.org/citation.cfm?id=1364813.1364827

[7] IBM. 2019. DB2 Version 9.1 for z/OS Codes. http://publib.boulder.ibm.com/epubs/pdf/dsnc1k17.pdf

[8] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 345–362. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[9] 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 (ASE 2016)*. ACM, New York, NY, USA, 472–482. https://doi.org/10.1145/2970276.2970354

[10] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. ACM, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[11] Microsoft. 2019. System Error Codes - Windows applications. https://docs.microsoft.com/en-us/windows/desktop/debug/system-error-codes

[12] OWASP. 2007. Top 10 2007-Information Leakage and Improper Error Handling. https://www.owasp.org/index.php/Top_10_2007-Information_Leakage_and_Improper_Error_Handling

[13] Thomas Reps, Stefan Schwoon, and Somesh Jha. 2003. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *Proceedings of the 10th International Conference on Static Analysis (SAS 2003)*. Springer-Verlag, Berlin, Heidelberg, 189–213. http://dl.acm.org/citation.cfm?id=1760267.1760283

[14] 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 (PLDI 2009)*. ACM, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[15] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[16] 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 (ESEC/FSE 2017)*. ACM, New York, NY, USA, 752–762. https://doi.org/10.1145/3106237.3106300