Detecting Error-Handling Bugs without Error Specification Input

Zhouyang Jia*†, Shanshan Li*, Tingting Yu†, Xiangke Liao*, Ji Wang*, Xiaodong Liu*, and Yunhuai Liu‡

*College of Computer Science
National University of Defense Technology
Changsha, China
{jiazhouyang, shanshanli, xkliao, wj, liuxiaodong}@nudt.edu.cn

†Department of Computer Science
University of Kentucky
Lexington, USA
tyu@cs.uky.edu

‡Big Data Research Center
Peking University
Beijing, China
yunhuai.liu@pku.edu.cn

***Abstract*—Most software systems frequently encounter errors when interacting with their environments. When errors occur, error-handling code must execute flawlessly to facilitate system recovery. Implementing correct error handling is repetitive but non-trivial, and developers often inadvertently introduce bugs into error-handling code. Existing tools require correct error specifications to detect error-handling bugs. Manually generating error specifications is error-prone and tedious, while automatically mining error specifications is hard to achieve a satisfying accuracy. In this paper, we propose *EH-Miner*, a novel and practical tool that can automatically detect error-handling bugs without the need for error specifications. Given a function, *EH-Miner* mines its error-handling rules when the function is frequently checked by an equivalent condition, and handled by the same action. We applied *EH-Miner* to 117 applications across 15 software domains. *EH-Miner* mined error-handling rules with the precision of 91.1% and the recall of 46.9%. We reported 142 bugs to developers, and 106 bugs had been confirmed and fixed at the time of writing. We further applied *EH-Miner* to Linux kernel, and reported 68 bugs for kernel-4.17, of which 42 had been confirmed or fixed.**

***Index Terms*—Error handling, Library function, Specification**

I. Introduction

Most software applications interact with their environments through libraries. These libraries provide various Application Programming Interface (API) functions for applications. A reliable application must be designed to behave gracefully even under API function failure conditions such as network packet loss, malformed input, memory allocation failure $etc.$ When any error occurs, the failing API function usually notifies its caller about the error, after which the caller decides how the error should be handled. Incorrect handling of errors can lead to serious problems, $e.g.$, system crashes or data loss, and software developers often make mistakes and inadvertently introduce bugs into error-handling code [1], [2]. Incorrect error handling had been identified as one of the top ten sources of security issues [3].

This work was supported in part by National Key R&D Program of China No. 2017YFB1001802; NSFC No. 61872373 and 61872375; NSF grant CCF-1652149 and CCF-1909085; High-End Generic Chips and Basic Software under grants No. 2017ZX01038104-002; China Scholarship Council.

Many researchers have proposed tools to detect error-handling bugs. Some of them use static analysis to check whether failures are properly handled in error paths [2], [4]–[7]. The process of these tools can be roughly split into two steps. The first step is to manually input or mine error specifications that indicate whether or not an API function may fail, and if so, how the function communicates with its caller. Second, they check whether or not the error is properly handled in the caller; if not, they report a bug. On the other hand, some tools use dynamic fault injection to check whether the program handles API failures correctly [8]–[11]. These works also require the error specifications as input initially. They simulate API failures according to the error specifications and check the response of the program.

The bug-detection abilities of both static and dynamic tools depend heavily on the accuracy of error specifications. Manually generating error specifications is error-prone and tedious, especially for low-level languages such as C/C++. C does not provide exception-handling mechanisms, while C++ contains far fewer try blocks than projects developed in the other languages [12]. To remedy this situation, previous works tried to automate the process of error specification mining [13]–[15]. Among these, Kang $et\ al.$ [13] studied the characteristics of error paths and proposed APEx, which collects error paths and infers error specifications with precision of 77% and recall of 47%. This precision is not accurate enough, since inferring error specifications is only the first step to detect error-handling bugs. Incorrect error specifications will negatively affect the following analysis. Acharya $et\ al.$ [14] designed a tool that mines error specifications automatically. This tool achieves 90% precision, but the recall is very limited — it only mined error specifications of 28 library functions from both POSIX and X11 libraries.

In light of the above, we propose *EH-Miner*, a novel and practical tool to automatically detect error-handling bugs without the need for error specification input. Given an API function, *EH-Miner* mines its error-handling rules when the function is frequently checked by an equivalent condition, and handled by the same action. To achieve this, we propose a Potential Error Handling (PEH) model, which describes

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00029

the potential error-handling behaviors in programs. In the PEH model, an error-handling behavior contains three basic elements: a library-function call, a check condition and a handling-function call. *EH-Miner* extracts PEH instances from a code repository, then normalizes these instances for ease of comparison. Finally, *EH-Miner* mines error-handling rules from the normalized instances.

Taking *chroot* as an example, developers frequently check whether or not the return value of *chroot* is -1; if it is, an error message is printed. The existing tools need the knowledge that "*chroot* returns 0 on success and -1 on error", then check whether the error is well handled in the error path, i.e., the path after the branch condition that the return value is -1. By contrast, *EH-Miner* can mine the error-handling rule that "*chroot* should be logged when returning -1" without inferring the semantics of -1. In *EH-Miner*, the rule is recorded as a PEH instance: $<chroot(...), chroot(...) == -1, log(...)>$. There are three advantages for mining error-handling rules compared with mining error specifications in existing tools:

- **Higher accuracy.** *EH-Miner* does not have to infer the semantics of API return values. This feature enables *EH-Miner* to improve 17% precision based on the state-of-the-art tool.
- **Better scalability.** *EH-Miner* uses light-weighted constraint solving instead of heavy analysis such as symbolic execution. The experiment shows that *EH-Miner* is about 20 times faster than the state-of-the-art tool.
- **Easier to repair.** *EH-Miner* can obtain error-handling actions besides error specifications, which helps to repair the detected bugs. With the assistance of this knowledge, we manually generate and report 210 patches.

To evaluate *EH-Miner*, we select 15 software domains that are widely used in the open source community, and choose 117 C/C++ applications from these domains. We first use the cross-validation technique to tune the parameters in *EH-Miner*, then evaluate the precision and recall under the optimal thresholds. Second, we evaluate the effectiveness of EH-Miner for detecting real-world bugs. *EH-Miner* reports 2266 bugs, and the precision is 88.16% at the 95% confidence level. We report 142 bugs, and 106 of them are confirmed by developers. We further apply *EH-Miner* to Linux kernel, and report 68 kernel bugs, of which 42 are confirmed. Finally, we compare EH-Miner with the state-of-the-art tools, and find EH-Miner outperforms the existing tools in terms of precision, recall and efficiency.

In summary, the contributions of this paper are as follows:

- We design and implement *EH-Miner*, a tool that can automatically detect error-handling bugs without error specification input. We conduct an empirical study on check conditions and error-handling actions, which guide the design of *EH-Miner*.
- In *EH-Miner*, we propose several novel techniques including using the PEH model to describe error-handling behaviors, converting the judgment of check-condition equivalence into a satisfiability problem, automatically recognizing error-handling actions, and a two-step mining method to mine error-handling rules.
- We apply *EH-Miner* to 117 applications across 15 software domains. *EH-Miner* mines error-handling rules with precision of 91.1% and recall of 46.9%. We reported 142 bugs, and 106 of them were confirmed by developers. We further applied *EH-Miner* to Linux kernel, and reported 68 kernel bugs, of which 42 were confirmed.

```
/* Example 1: mysql-5.7.16/sql/mysqld.cc:1586:7 */
if (chroot(path) == -1) {
    sql_print_error("chroot: %s", strerror(errno));
    unireg_abort(MYSQLD_ABORT_EXIT);
}
/* Example 2: opensmtpd-6.0.2p1/smtpd/ca.c:101:6 */
if (chroot(PATH_CHROOT) == -1)
   fatal("ca: chroot");
/* Example 3: httpd-2.4.25/modules/arch/unix/mod_unixd.c:162:13 */
if (chroot(ap_unixd_config.chroot_dir) != 0) {
    rv = errno;
    ap_log_error(... "Can't chroot to %s", ...);
    return rv;
}
/* Example 4: postfix-3.2.2/src/util/chroot_uid.c:68:6 */
if (chroot(root_dir))
    msg_fatal("chroot(%s): %m", root_dir);
/* Example 5: cherokee-1.2.103/cherokee/source_interpreter.c:633:9 */
re = chroot(src->chroot.buf);
if (re < 0) {
    LOG_ERROR (..., src->chroot.buf);
    exit(1);
}
```

Fig. 1. Error handling examples of `chroot`.

II. Motivation and Definition

In this section, we present a real-world example that motivated us to design and implement *EH-Miner*. Figure 1 shows five different applications invoking the API function *chroot*. According to the *man* page documentation for *chroot*, we know that upon successful completion, a value of 0 is returned; otherwise, a value of -1 is returned. Guided by the *man* document, all these five applications handle errors in a similar way — checking whether the return value is -1, and if so, printing error messages.

To find error-handling bugs of *chroot*, the existing solutions first need to either input (e.g., ErrDoc [4], EPEx [5]) or mine (e.g., APEx [13], LFI [11]) the error specification that "0 means success and -1 means error". Only after this is accomplished can they check whether the error is properly handled when -1 is returned. However, manually inputting error specifications is tedious, and sometimes even error-prone for libraries with limited domain knowledge. On the other hand, mining error specifications automatically is hard to get a satisfying accuracy even in the state-of-the-art tool.

In this paper, we propose a novel approach that does not require error specifications as input. Given an API function, if developers frequently use an equivalent check condition and the same error-handling action, we consider this to be an error-handling rule for this API function. This insight is naturally useful for mining error-handling rules, as developers usually perform similar actions in error paths (*e.g.*, printing error messages or exiting the program), whereas in normal

paths, developers perform different actions according to the program-specific functionalities. As illustrated in Figure 1, all the check conditions are equivalent (checking whether the return value is -1) and followed by the same action (printing an error message). Thus, we obtain the rule that *chroot* should be logged when returning -1.

Challenges. There are two main challenges for mining error-handling rules. First, to determine the check conditions that appear frequently (*e.g.*, whether the return value is -1), we need to judge the equivalence of check conditions. This is a difficult task, as two check conditions can be written in various forms but still have the same meaning. For example, "if(chroot() == -1)" and "if(chroot() != 0)" are equivalent — both check whether the return value of *chroot* is -1, since *chroot* can only return -1 or 0 — but semantically different. Second, we need to recognize error-handling actions (*e.g.*, printing an error message) after the check conditions. Achieving this is non-trivial since the implementations of an error-handling action may take various forms across different applications. Taking the *print* action as an example, when printing an error message, most applications implement custom logging functions instead of using the library function *printf* directly. For example, all applications in Figure 1 use custom logging functions.

To overcome these challenges, we study the source code of real-world projects and summarize the characteristics of equivalence between check conditions, as well as the characteristics of error-handling actions. Guided by these characteristics, we employ two normalization methods to judge the equivalence of check conditions and recognize error-handling actions.

Definition. To clearly describe error-handling behaviors like the examples in Figure 1, we summarize the following three elements, and combine these elements into the Potential Error Handling model.

- Library-function Call (*LC*): A call statement that invokes a library function.
- Check Condition (*CC*): A Boolean expression that is the condition of a branch statement (*e.g.*, *if* or *switch*), and the branch statement is flow-dependent [16] on an *LC*.
- Handling-function Call (*HC*): A call statement that is control-dependent [16] on a branch statement, and the branch statement is flow-dependent on an *LC*.
- Potential Error Handling model (*PEH*): A 3-tuple *M*=<*LC*, *CC*, *HC*>.

III. Characteristics Study

As outlined in Section II, there are two main challenges in obtaining error-handling rules. One is the judgment of equivalence between check conditions; the other is the recognition of error-handling actions. We studied the real-world applications and summarized some characteristics that help to overcome these challenges. We select five mature open-source applications from different software domains in order to provide sufficient generality for our investigations. All target applications are active and widely used, written in C/C++, and have long development history. The applications include

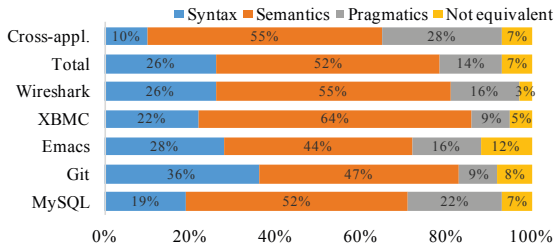

Fig. 2. Distribution of different equivalence levels.

MySQL-5.7.16 (database), Git-2.9.3 (version control), Emacs-25.2 (text editor), XBMC-17.3 (video player) and Wireshark-2.2.7 (packet analyzer).

A. Characteristics of equivalence between check conditions

Developers can use various forms to implement check conditions that perform exactly the same behavior. It is non-trivial to determine whether two check conditions are equivalent. To study the equivalence between check conditions, we manually selected 100 pairs of check conditions in each of the five applications (*i.e.*, 500 pairs in total). For each pair, we first randomly selected a library function used in the studied application, then randomly selected two call-sites of this function. To study the equivalence across different applications, we further selected 100 pairs of cross-application check conditions (*i.e.*, the two call-sites are from different applications).

To judge the equivalence between check conditions, the most intuitive method is to compare conditions as strings. This method requires the conditions to be equivalent at the syntactic level. Most equivalent conditions, however, do not meet this requirement. A practical tool should be capable of judging the equivalence at the semantic level. Moreover, the conditions with different semantics might still have the same behavior in certain contexts. We regard this level of equivalence as pragmatic level. Therefore, we classify the condition equivalence into three levels and investigate their distributions:

Syntactic Level. Two check conditions are exactly the same except for some blanks and comments, *e.g.*, "foo()==-1" and "foo() == -1" are equivalent in syntax.

Semantic Level. Given the same input, two check conditions will choose the same branch, *e.g.*, both "foo()!=0" and "foo()" will choose true branch when *foo* returns non-zero.

Pragmatic Level. Two check conditions have different semantics, *e.g.*, "foo()==-1" and "foo()!=0". But they are still equivalent under the specific context, *e.g.*, the return value of *foo* is limited to {-1, 0}.

For example, in Figure 1, the check conditions of *chroot* in Example 1 and 2 are equivalent at the syntactic level. Example 3 and 4 are equivalent at the semantic level, while Example 2 and 3 are equivalent at the pragmatic level. We studied the distribution of these equivalence levels. The results are illustrated in Figure 2. Taking MySQL as an example, 19%, 52% and 22% of check conditions are equivalent at the syntactic, semantic, and pragmatic levels respectively,

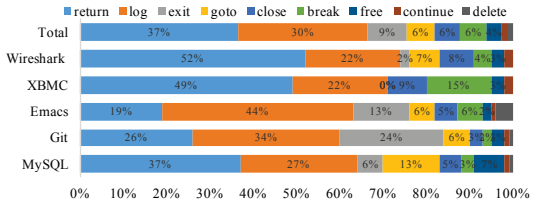

Fig. 3. Distribution of different error-handling actions.

while 7% of them are not equivalent. The average distribution across the five applications is 26.2%, 52.4%, 14.4% and 7%. Moreover, in cross-application samples, the percentage of pragmatic equivalence increases from 14.4% to 28%.

> The percentages of semantic and pragmatic equivalence are 55% and 28% in cross-application samples. It is important to judge the equivalence beyond the syntactic level.

B. Characteristics of error-handling actions

As developers usually use custom functions to handle errors, it is difficult to directly compare error-handling actions across different applications. All projects shown in Figure 1 use custom logging functions, *e.g.*, *sql_print_error* in MySQL and *ap_log_error* in Httpd. To study the existing actions, we randomly selected 100 error-handling code snippets in each of the five applications (*i.e.*, 500 in total) and found nine kinds of error-handling actions via manual analysis. According to how the actions are implemented, these actions can be classified into two categories: unconditional branch statements and custom error-handling functions. The former is implemented by keywords in C/C++ language, while the latter is implemented by custom functions.

Unconditional branch statements. Four kinds of actions are implemented by unconditional branch statements: *return*, *break*, *continue* and *goto*. Among these, the *return* action is most frequently used in the error-handling code snippets. As outlined in Figure 3, on average, 37% of error-handling code snippets use return statements. The *goto* action is another important error-handling mechanism in C/C++ applications, which is used in 6% of snippets on average. In addition, the *break* (6%) and *continue* (1%) actions are used to handle errors in loops such as *for* statements and *while* statements. In general, these kinds of error-handling actions can be recognized by simply searching the keywords.

Custom error-handling functions. Five kinds of actions are implemented by custom error-handling functions: *log*, *exit*, *close*, *delete* and *free*. Among these, the *log* action prints error messages to any target, including console, file or socket. The *exit* action exits the application immediately by wrapping the functions such as *exit* or *abort*. Figure 3 shows 30% and 9% of error-handling code snippets on average perform *log* and *exit* actions, respectively. In addition, the *close* and *delete* actions refer to closing or deleting a file or directory. These two actions are found in 6% and 1% of total snippets respectively. The final action, *free*, is used to free memory and appears in 4% error-handling code snippets.

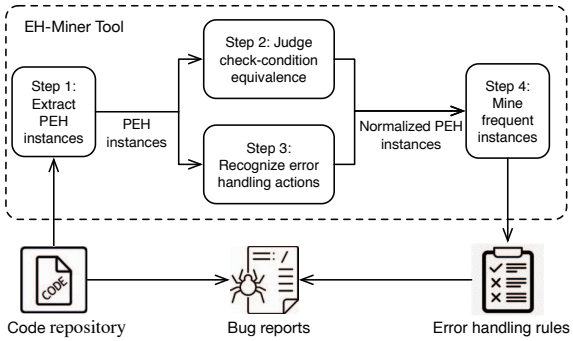

Fig. 4. Overview of *EH-Miner*.

> There are nine common used error-handling actions, which can be classified as unconditional branch statements (*return*, *goto*, *break*, *continue*) and custom error-handling functions (*log*, *exit*, *close*, *delete*, *free*).

IV. *EH-Miner* Design

In this section, we present the detailed design of *EH-Miner*. As shown in Figure 4, *EH-Miner* takes source code as input and generates error-handling rules for library functions. It then detects violations of these rules and identifies them as error-handling bugs. In *EH-Miner*, the first step is to extract PEH instances, which transfer the source code into structured data. *EH-Miner* then judges the equivalence of check conditions across PEH instances, and recognizes error-handling actions of each instance. For ease of comparison, *EH-Miner* normalizes these instances according to the equivalence of check conditions and the type of error-handling actions. Finally, *EH-Miner* mines error-handling rules from the normalized PEH instances.

A. Extracting PEH Instances

EH-Miner extracts PEH instances according to the algorithm described in Algorithm 1. For each function definition, EH-Miner scans its source code and collects all call statements (line 2). For each call statement, EH-Miner checks whether or not it is calling a library function(line 4), and if so, regards the call statement as an *lc* (library-function call) (line 5). From the *lc*, EH-Miner searches forwards to collect all branch statements (line 7), and chooses the ones that are flow-dependent on *lc*. Conditions of the chosen branch statements will be regarded as *cc* (check condition) (line 8 and 9). EH-Miner then explores all branch paths of the chosen branch statements to find call statements and unconditional branch statements (line 10). For each call statement, EH-Miner records the statement as an *hc* (handling-function call) (line 12), and adds a new PEH instance <*lc*, *cc*, *hc*> to M (line 13 and 14). For each unconditional branch statement, EH-Miner regards it as a special handling-function call, and adds a new PEH instance <*lc*, *cc*, return/goto/break/continue> to M (line 17 and 18). Finally, EH-Miner returns M, which is the set of PEH instances of the input function definition (line 24).

For example, the first code snippet in Figure 5 calls a library function *foo*, followed by two check conditions: one

Algorithm 1 PEH Instances Extraction Algorithm.

Require: Function definition F
Ensure: Instances set $M = \{m | m = < lc,\ cc,\ hc >\}$

```
M = Φ
Scan F's source code, and get all call statements
for each call statement f in F do
    if f calls a library function then
        Let lc = f
        From lc, scan forwards, and get all branch statements
        for each branch statements bs after lc do
            if bs is flow dependent on lc then
                Let cc = condition of bs
                Search paths of bs, and get calls and
  unconditional branches.
                for each call statement f' do
                    Let hc = f'
                    Let m = < lc, cc, hc >
                    Add m to M
                end for
                for each unconditional branch statement do
                    Let m=<lc,cc,return/goto/break/continue>
                    Add m to M
                end for
            end if
        end for
    end if
end for
Return M
```

is checking for a return value, while the other is checking for a pointer argument. Each check condition has one handling-function call. Therefore, *EH-Miner* extracts two PEH instances, m_1 and m_2. In addition to the above, *EH-Miner* also needs to properly deal with the following situations: (a) *Else branch*: When a function is called in the else branch of an if statement, *EH-Miner* negates the check condition; for example, the second code snippet in Figure 5 generates m_3. For $switch$ statements, *EH-Miner* records the switch case label together with the conditional expression. (b) *Non-reaching definition*: A library-function call will be ignored when its return value is reassigned. For example, in the third code snippet in Figure 5, Since the return value of foo is reassigned by "rv = a"; accordingly, "rv = foo(ptr)" is considered a non-reaching definition [17] for rv in "if(rv == 0)", and will be ignored. (c) *Nested branches*: Two check conditions will be jointed when their branch statements are nested. For the fourth code snippet in Figure 5, "if(rv != 0)" is located in the else branch of "if(rv == -1)"; thus, the check condition of "exit()" will be "!(rv == -1) && (rv != 0)". This code snippet finally generates m_4 and m_5.

B. Judging Equivalence of Check Conditions

For each pair of identified PEH instances, EH-Miner judges the equivalence of their check conditions. According to Section III-A, there are three levels of equivalence between check conditions: syntactic level, semantic level and pragmatic level. In this part, we will outline the procedure for judging equivalence of these levels.

```
rv = foo(ptr);
if(rv == -1)
  log();
if(ptr == NULL)
  return;
```

```
rv = foo(ptr);
if(rv == 0)
  ...
else
  log();
```

```
rv = foo(ptr);
rv = a;
if(rv == 0)
  exit();
```

```
rv = foo(ptr);
if(rv == -1)
  log();
else if(rv != 0)
  exit();
```

PEH instances:
m_1 = <rv=foo(ptr), rv==-1, log()>
m_2 = <rv=foo(ptr), ptr==NULL, return>
m_3 = <rv=foo(ptr), !(rv==0), return>
m_4 = <rv=foo(ptr), rv==-1, log()>
m_5 = <rv=foo(ptr), !(rv==-1)&&(rv!=0), exit()>

Fig. 5. Mapping of code snippets and PEH instances.

Syntactic equivalence. It is easy to judge the syntactic equivalence between check conditions. The most intuitive approach is to compare two check conditions as strings.

Semantic equivalence. Compared with the syntactic level, this equivalence level is much more powerful. We convert the judgment of this equivalence into a satisfiability problem, as described in the following theorem.

Theorem 1. *Given two check conditions X and Y, X and Y are semantically equivalent if the expression $(X \wedge \neg Y) \vee (\neg X \wedge Y)$ is not satisfiable.*

Proof. If $(X \wedge \neg Y) \vee (\neg X \wedge Y)$ is not satisfiable, then both $(X \wedge \neg Y)$ and $(\neg X \wedge Y)$ are not satisfiable.

If $(X \wedge \neg Y)$ is not satisfiable, then there is no case that satisfies X but does not satisfy Y. That is to say, every case that satisfies X will also satisfy Y, namely, $Y \rightarrow X$.

Similarly, if $(\neg X \wedge Y)$ is not satisfiable, then $X \rightarrow Y$.

If $Y \rightarrow X$ and $X \rightarrow Y$, then X and Y are equivalent. □

For example, suppose there are two equivalent check conditions X and Y, where X is "rv != 0" and Y is "rv $>$ 0 || rv $<$ 0". *EH-Miner* calculates the satisfiability of $(\neg X \wedge Y)$ and $(X \wedge \neg Y)$ through a Satisfiability Modulo Theories (SMT) [18] solver, and finds that both of them are not satisfiable; thus, $(X \wedge \neg Y) \vee (\neg X \wedge Y)$ is not satisfiable. Therefore, X and Y are equivalent.

Pragmatic equivalence. Given two check conditions "rv == -1" and "rv != 0", they are not equivalent according to Theorem 1, but have the same behavior when rv is the return value of $chroot$ ($i.e.$, belongs to {-1, 0}). To solve this problem, we have to obtain the return-value ranges of library functions. *EH-Miner* only analyzes constant error returns. According to [10], in the libraries on standard Linux systems, the non-constant return values are used only to indicate success conditions (e.g., the number of bytes read).

For each library function, we first find all its return statements; from each return statement, we recursively search backwards along the use-define chain [19] to find the definitions of the returned variables, then record all the integer definitions as the return-value range. Taking $chroot$ as an example, the range expression of its return value will be "ret == -1 || ret == 0". With the return-value ranges R available, we have the following theorem.

Theorem 2. *Given two check conditions X and Y, X and Y are pragmatically equivalent if the expression $((X \wedge \neg Y) \vee (\neg X \wedge Y)) \wedge R(rv_1) \wedge R(rv_2) \wedge ... \wedge R(rv_n)$ is not satisfiable,*

where $rv_1, rv_2, ..., rv_n$ are return-value variables in X and Y, and $R(rv_i)$ $(i \in [1, n])$ is the range expression of rv_i.

Proof. If the above expression is not satisfiable, then both (i) and (ii) below are not satisfiable.

$$X \wedge \neg Y \wedge R(rv_1) \wedge R(rv_2) \wedge ... \wedge R(rv_n) \quad (i)$$
$$\neg X \wedge Y \wedge R(rv_1) \wedge R(rv_2) \wedge ... \wedge R(rv_n) \quad (ii)$$

If (i) is not satisfiable, the satisfiability of $(X \wedge \neg Y)$ has two cases: (a) $(X \wedge \neg Y)$ is not satisfiable, in which case $Y \rightarrow X$ (Theorem 1); (b) $(X \wedge \neg Y)$ is satisfiable, in which case, let S denote the cases that satisfy X but do not satisfy Y. Since (i) is not satisfiable, $S \wedge R(rv_1) \wedge R(rv_2) \wedge ... \wedge R(rv_n)$ must not be satisfiable (S is a subset of $(X \wedge \neg Y)$). That is to say, the cases in S are impossible, since they are contradictory to the ranges of return values. Thus, S is empty, and every case that satisfies X will also satisfy Y; in short, $Y \rightarrow X$.

Similarly, if (ii) is not satisfiable, then $X \rightarrow Y$.

If $Y \rightarrow X$ and $X \rightarrow Y$, then X and Y are equivalent. □

Name normalization. When the check conditions have different variables, we can apply neither Theorem 1 nor 2 directly to judge their equivalence. For example, given two check conditions X and Y, where X is "rc==-1" and Y is "err!=0", the satisfiability of rc and err can not be judged. To solve this problem, *EH-Miner* normalizes the variables.

Besides the exception-handling mechanism, there are three common ways for library functions to communicate errors with their callers. The first is return values, $e.g.$, $chroot$ returns -1 when failing. The second is global variables, $e.g.$, $errno$ in $libc$ is widely used to record the error code. The third is pointer arguments; for example, in $glib$, many functions use pointer arguments to pass errors, $e.g.$, "g_xxx(..., GError **error)" communicates errors with its caller through the argument $error$. In light of the above, we only analyze three kinds of variables, $i.e.$, return values, global variables and pointer arguments, while other variables will be ignored.

In *EH-Miner*, global variables are used directly, since their names are the same even in different applications. Return values are renamed to the function names with a 0 attached; for example, rv will be renamed to $chroot\_0$ when "rv = chroot()". Pointer arguments are renamed to the function name with an index attached; for example, for function call "foo(x, y)", x and y will be renamed to foo_1 and foo_2 respectively. Moreover, when a check condition contains a function call, the call will be regarded as a return value, $e.g.$, "chroot()==-1" will be normalized to "chroot_0==-1".

After name normalization, *EH-Miner* applies Theorem 2 or Theorem 1 (if the library source code is not available) to judge the equivalence of the check conditions. For ease of comparison, all check conditions are partitioned into clusters, and check conditions in the same cluster are equivalent. *EH-Miner* normalizes the PEH instances by replacing their check conditions with the cluster IDs.

C. Recognizing Error-Handling Actions

EH-Miner needs to recognize the error-handling action from each identified PEH instance. In solving this problem, existing tools [5] only deal with three most common error-handling actions: log, $return$ and $exit$, and require users to input logging functions. We propose a more practical technique that can not only deal with more error-handling actions, but also automate the recognition process.

Recalling the characteristics outlined in Section III-B, there are nine error-handling actions that can be classified into two categories. For actions in the first category, $i.e.$, unconditional branch statements, we search those statements mechanically. For actions in the second category, $i.e.$, custom error-handling functions, we use two heuristic rules: (a) The custom error-handling function always wraps certain library functions. Taking action log as an example, its custom error-handling functions always wrap functions such as $printf$, $write$ and $send$. This is not surprising since the custom function has to implement the functionality of the action eventually. (b) There is a call chain from the custom error-handling function to the wrapped library function, and each function in the call chain contains certain keywords. For example, in MySQL, the custom logging function sql_print_error calls library function $fprintf$ through the call chain sql_print_error, $error\_log\_print$, $print\_buffer\_to\_file$, $fprintf$. Each function in the chain contains keywords such as "print", "error" or "log" that refer to printing error messages.

For each action act, we empirically define its wrapped standard-library function set SF_{act} ($e.g.$, {printf, write, send ...} for action log) and its keyword set KW_{act} ($e.g.$, {print, log, fatal ...} for action log). Given an error-handling function, *EH-Miner* builds its call graph, then applies depth first search on the call graph. (If the function invokes a third-part library, *EH-Miner* also builds the call graph of the third-part library function.) Each path p of the call graph may perform one error-handling action. *EH-Miner* determines that p performs act if: (a) p ends up with a standard-library function in SF_{act}; (b) every node in p contains at least one keyword in KW_{act}. *EH-Miner* collects the actions of all paths.

For ease of comparison, *EH-Miner* normalizes the PEH instances by replacing their handling-function calls with error-handling actions (if any). When a handling-function call performs multiple actions, *EH-Miner* will split the instance into multiple separated ones that share the same library-function call and check condition, but have different actions.

D. Mining Error-Handling Rules

EH-Miner mines error-handling rules by using the normalized PEH instances. Each rule contains three elements: a library function, a check condition and an error-handling action. For a library function, *EH-Miner* first selects its check conditions that appear frequently, then determines the corresponding error-handling actions for each check condition.

Mining check conditions. In some applications, developers use wrapper functions that wrap library functions and add

certain error-handling code. While other applications may use the wrapped library functions directly. Mixing the usage of library functions and wrapper functions may cause inaccuracy in cross-application mining. To avoid this, we propose a two-step method to mine the check conditions:

- *Intra-application mining step.* For a library function f, we refer to the number of its invocations in application p as $N(p, f)$. For a check condition c of f, we refer to the number of its appearance in p as $M(p, f, c)$. We say an application p supports c if $M(p, f, c)$ / $N(p, f) > TH_{intra}$, where TH_{intra} is a pre-defined threshold. This formula indicates that the application p supports c when c frequently appears after f.
- *Inter-application mining step.* Let $P(f)$ denote the number of applications that invoke f, and $Q(f, c)$ denote the number of applications that support c. A check condition c will be chosen if $Q(f, c)$ / $P(f) > TH_{inter}$, where TH_{inter} is a pre-defined threshold. This formula indicates that a check condition will be selected if most applications support it.

Mining error-handling actions. In this step, *EH-Miner* determines the error-handling actions. *EH-Miner* counts the number of every action used after above check conditions, and chooses the actions satisfying both the following rules: (a) the number of the action is the largest among all actions; (b) the number of the action is at least one standard deviation larger than the average number of all actions.

With reference to the error-handling rules available, *EH-Miner* scans the source code again to detect violations of these rules. *EH-Miner*, however, may report tens of thousands of violations when applying the rules directly. In this regard, *EH-Miner* further ranks the violations. For a library function f, we refer to the number of its total call sites as C_f and the number of its violations as V_f. Let R_f denote V_f/C_f. Violations of f with smaller R_f are more likely to be true positives.

V. Implementation

We implement *EH-Miner* using Clang [20] (front-end analysis), LLVM IR [21] (data flow analysis), Z3 [22] (SMT solving) and SQLite3 [23] (data management). To extract PEH instances, we implement a standalone Clang tool and leverage the use-define chain in LLVM IR. The extracting step of *EH-Miner* is written in C++, while all other steps are written in Python. To judge the equivalence of the check conditions, we apply Z3 to solve the satisfiability problem. The intermediate results across different steps are stored in a SQLite3 database.

To build the code repository, we carefully select 15 software domains that are widely used in the open source community. For each software domain, we choose the mature applications that are open source, written in C/C++, and have a long development history. Most of these applications are configurable and have a range of optional features. To analyze the libraries used by the optional features, we enable as many features as possible rather than using the default configuration when building. In total, we choose 117 applications that are successfully built, which include 26.2M lines of code (LOC) and 51,264 source files. We get the data by using $cloc$ [24] tool; the LOC number excludes the lines of blank, comment or code in header files. These applications depend on hundreds of libraries, which are all automatically analyzed by *EH-Miner*. In the interests of saving space, the applications and libraries will not be listed here[1].

All our analysis was conducted on 64-bit Ubuntu 16.04, with 2.7 GHz Intel i5 and 4 GB DDR3. *EH-Miner* spent ten hours in total and generated 536K PEH instances.

VI. Experimental Results

In this section, we evaluate *EH-Miner* by answering the following research questions (RQs).

RQ1: What are the precision and recall of *EH-Miner*?

RQ2: Can *EH-Miner* find real-world bugs effectively?

RQ3: Does *EH-Miner* outperform the state-of-the-art tools?

A. Precision and Recall of EH-Miner

EH-Miner requires two parameters to be pre-defined, $i.e.$, TH_{intra} and TH_{inter}. These parameters may obviously affect the precision (P) and the recall (R) of *EH-Miner*. In this part, we first tune the parameters, then evaluate P and R under the optimal parameters.

Experiment setup. We use cross-validation to tune the parameters so that the parameters can generalize well to any dataset. Each of the 117 applications is randomly assigned to one of the two sets: a training set and a test set. This process is independently repeated 20 times, so we have 20 training sets and 20 corresponding test sets. We carefully tune the parameters to fit the training sets, then evaluate the averaged P and R in the test sets. It is hard to evaluate P and R of error-handling bugs due to lack of the ground truth. Being similar to the existing works [13], [14], we evaluate P and R of the reported error-handling rules. We studied all POSIX functions used by all applications, and manually generated their error specifications (recorded in Z3 format), serving as oracles for confirming the reported rules. A rule will be regarded as correct if its check condition is equivalent to (or is a subset of) the error condition of the error specification. The equivalence judgement is performed by the Z3 solver, similar to the process in Section IV-B.

Suppose the expected number of error-handling rules is N. Let M denote the number of reported rules, and M' denote the number of correct rules. The precision is the ratio of $|M'|$ to $|M|$, while the recall is the ratio of $|M'|$ to $|N|$. For *EH-Miner*, the expected number of rules is the number of correct rules which appear at least one time in any application. Moreover, we prefer an overall metric that considers both P and R for the ease of comparison. The most common practice is to calculate F_{score}, which is the harmonic mean of P and R. For static tools of bug detection, however, P is much more important than R. Brittany $et\ al.$ [26] conducted interviews with 20 developers to ascertain why software developers do not use

[1]Please refer to our project homepage [25] for the target applications, source code and bug reports involved in this paper.

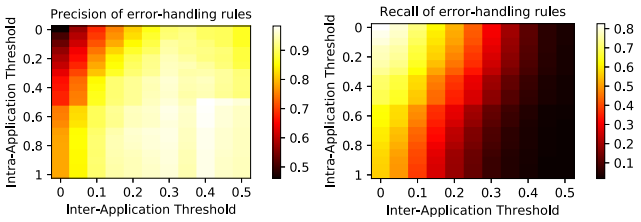

Fig. 6. Different metrics of reported error-handling rules.

static analysis tools to find bugs and found that false positives constitute the most significant barrier. With this in mind, we used the weighted harmonic mean of P and R:

$$F'_{score} = \frac{w_1 + w_2}{\frac{w_1}{Precision} + \frac{w_2}{Recall}},$$

where w_1 and w_2 are the weights of P and R, respectively. The weights of precision and recall are subjective for different users, thus *EH-Miner* provides user interfaces to change these parameters, while the default values of w_1 and w_2 are 0.7 and 0.3, respectively.

Optimal thresholds. We apply *EH-Miner* to the training sets, and evaluate the rules that are related to POSIX functions. We iterate the parameters TH_{intra} and TH_{inter} from 0 to 1 with a step length of 0.05. The results of averaged P and R are shown in Figure 6. It is clear that P increases with the corresponding increase in the thresholds, while R is reduced as the thresholds increase. The averaged P ranges from 0.45 to 1, while the averaged R ranges from 0.83 to 0. We normalize both P and R by using min-max normalization [27] so that each metric contributes approximately proportionately to F'_{score}. As the two thresholds are increased, F'_{score} initially increases and subsequently decreases. When TH_{intra} equals to 0.35 and TH_{inter} equals to 0.15, *EH-Miner* achieves its highest averaged F'_{score} in the training sets, while the averaged P and R are 90.2% and 48.3%, respectively.

The precision and recall. We apply *EH-Miner* to the test sets by using (0.35, 0.15) as the optimal parameters. The averaged P and R of the 20 test sets are 91.1% and 46.9%, while the standard deviation of P and R are 1.5% and 4.0%. The results indicate that *EH-Miner* can achieve a higher precision compared with [13], and a much higher recall compared with [14]. Meanwhile, the recall and the precision are similar to [13] and [14], respectively. In Section VI-C, we will compare these tools in detail.

B. Real-World Bugs Reported by EH-Miner

In this part, we evaluate the effectiveness of *EH-Miner* through its ability of detecting real-world bugs.

Bugs in applications. We apply *EH-Miner* to all 117 applications. *EH-Miner* found 1036 error-handling rules (180 for POSIX functions and 856 for other third-party libraries). With these rules available, *EH-Miner* found 2266 violations, which are from 189 library functions. We sampled and analyzed the violations until the confidence interval is less than or equal to 3 at 95% confidence level. We skipped the violations that we can not determine the correctness, and finally sampled 380 violations, of which 335 are true positives, and 45 are false positives. At 95% confidence level, the precision is 88.16% with a confidence interval of 2.96, *i.e.*, [85.20%, 91.12%].

TABLE I
NUMBER OF CONFIRMED OR FIXED BUGS.

Application	Bug	Application	Bug	Application	Bug
Amarok*	2	Ayttm	2	Balsa	1
Bftpd	12	BitKeeper*	3	Bluefish	4
Citadel*	6	Claws-mail*	7	Cubrid	10
Darktable*	4	Ettercap*	6	Fossil	3
GFTP	3	GIMP*	4	Httpd*	4
LFTP	3	Mariadb*	2	MonetDB*	5
MPlayer*	2	Mutt	1	Netsniff-ng	7
Pidgin*	3	VLC*	1	W3M	5
XBMC*	3	Xplico	3		

* The application is regularly checked by Coverity or Fortify.

We found the true positive violations can be classified into two types. Type A (248 violations) indicates that the library functions are not checked at all. Violations in this type are definitely bugs which may cause serious results like crash or data loss. Type B (87 violations) means developers have checked the library functions and performed normal actions in normal paths, while the error paths are ignored. This type of violation may not lead to a crash in most test cases, but we still believe these violations should be fixed conservatively to make the program more robust.

The false positive cases are mainly caused by the following reasons: (i) The error-handling rule is wrong. For example, the function $ferror$ should not fail, but *EH-Miner* reports error-handling bugs for $ferror$. (ii) The library function is checked by another way. For example, the return value of $notify\_init$ (in library $Libnotify$) is normally checked before calling other $Libnotify$ functions (e.g., $notify\_notification\_new$). The $Balsa$ program, however, calls $notify\_is\_initted$ before calling other $Libnotify$ functions to make sure $notify\_init$ is successfully initialized. In this situation, *EH-Miner* reports a false positive about $notify\_init$. (iii) The error-handling code is located in complex context, and *EH-Miner* fails to detect.

We are now in the process of reporting the Type A bugs to the developers due to their importance. So far, we have submitted patches for 142 bugs according to the ranking R_f (defined in Section IV-D), and patches for 106 bugs have been confirmed or fixed by developers at the time of writing. On the other hand, seven patches were rejected. Among them, five were regarded as unnecessary, since developers believe that the bugs will never happen. One patch was rejected since it will result in another bug. For the last rejected patch, the developer believes the whole system will down when the bug happens, and it is useless to fix the bug in an application. In Table I, we list all the bugs which have been confirmed or fixed by developers.

Case study. In this part, we study a real-world bug detected by *EH-Miner* — the MonetDB Database Server daemon, $monetdbd$, will crash if a wrong host was set to the configuration item $listenaddr$. If a user set a wrong host to $listenaddr$ and started the server, the host will be sent directly

to the library function $gethostbyname$. In this situation, $gethostbyname$ will fail and return a NULL pointer. Further use of the return value will cause a NULL pointer dereference. Monetdbd will abort if this bug is triggered; thus, developers fixed the bug once it was reported and labelled it as a critical bug.

To detect this bug, *EH-Miner* found that $gethostbyname$ is used in 37 different applications, 23 of which have checked whether or not its return value is NULL; if so, performed the $return$ action. Therefore, *EH-Miner* got the error-handling rule that "When $gethostbyname$ returns NULL, its caller function should return". Finally, *EH-Miner* scanned the source code again to detect violations of this rule.

Bugs in Linux kernel. We further apply *EH-Miner* to Linux kernel to evaluate the effectiveness of *EH-Miner* on a different dataset. The error-handling rules of API functions can not be used since kernel does not use any library. As *EH-Miner* performs cross-project mining, each kernel subsystem will be regarded as a separated project, $e.g.$, fs/ext4, and we have 395 projects in the kernel dataset. *EH-Miner* gets 1590 error-handling rules and 2069 violations of the rules for 298 kernel functions. We randomly inspect 460 violations, of which 386 are true positives, while 74 are false positives. At 95% confidence level, the precision is 83.91% with a confidence interval of 2.96, $i.e.$, [80.95%, 86.87%].

As before, we are now in the process of reporting the Type A kernel bugs to developers. So far, we had reported 68 bugs (from 30 kernel functions and each function may have multiple bug instances), and 42 bugs had been confirmed or fixed by developers at the time of writing. Among these, 28 bugs were fixed by using our patches; 6 bugs are under discussing about how to improve the patches; 8 patches are rejected even though the bugs are confirmed, since developers believe the whole functions are buggy, while our patches do not help. On the other hand, 22 bugs are rejected. One reason is that there is no serious consequence even the function fails. For example, developers ignore the errors of $i2c\_new\_dummy$ (6 instances), since i2c devices are not really used. Another reason is that the error has been well handled in other way. For example, when $kmem\_cache\_create$ fails (9 instances), kernel panic happens if the function is invoked with argument SLAB_PANIC.

C. Comparison of Different Methods

In this section, we compare *EH-Miner* with the existing tools of detecting error-handling bugs. LFI [11] is a dynamic analyzing tool, while *EH-Miner* is a static one. Dynamic tools can achieve much higher accuracy compared with static tools, but they are unable to find bugs that have not been executed. Therefore, *EH-Miner* is complementary to LFI. EPEx [5] and ErrDoc [4] are static tools but require users to provide error specifications as input, while *EH-Miner* requires no such input. Some commercial tools are widely used in practice, $e.g.$, Coverity [28] and Fortify [29]. *EH-Miner* reported 106 bugs that were fixed by developers at the first time. These bugs are from 26 projects, while 14 projects (containing 52 bugs) have been regularly checked by Coverity or Fortify as shown in Table I. The fact that none of these bugs were detected by these tools demonstrates that *EH-Miner* can detect bugs that the state-of-the-art tools miss. The closest works to *EH-Miner* is APEx [13] and Acharya $et\ al.$ [14], which are static tools and mine error specifications automatically. We choose [13] and [14] as the baseline methods.

Experiment setup. We compare these tools through three metrics: the precision, the recall and the efficiency. APEx [13] and Acharya $et\ al.$ [14] manually evaluated both the precision of bugs and the precision of error specifications. The precision of bugs, however, might be biased, since the authors might have limited domain knowledge and simply regard violations of correct rules as true positives. This assumption will lead to overestimation. For example, all rejected bugs in Section VI-B are violations of correct rules, but they are not true positives. In this regard, we use P and R of error specifications, since the standard of the library functions can serve as an oracle.

EH-Miner mines error-handling rules instead of error specifications. For the ease of comparison, let I denote the expected number of functions covered by error specifications or error-handling rules. Let J denote the number of functions that covered by reported error specifications or error-handling rules, and J' denote the number of functions that covered by correct error specifications or correct error-handling rules. The precision is the ratio of $|J'|$ to $|J|$, while the recall is the ratio of $|J'|$ to $|I|$. For functions covered by multiple rules, we calculate the proportion of correct rules. For example, if a function is covered by three rules — one correct rule and two wrong rules — the number of function covered by correct rules is 0.33.

Comparing above metrics directly may also cause bias since *EH-Miner* learns from more applications. To remedy this, we apply *EH-Miner* to 28 applications which are used by APEx [13]. For space reason, we do not list these applications. As for Acharya $et\ al.$ [14], the authors did not provide their application list, and the tool is not available, so it is hard to compare [14] with *EH-Miner* by using the same dataset.

Comparison with [14]. Under optimal parameters, *EH-Miner* mines 120 error-handling rules for POSIX library covering 109 functions. Among them, rules of 98.58 functions are correct (some functions are covered by both correct and wrong rules), while the expected number of functions is 138 (POSIX functions that appear in any of the 28 programs). Thus the precision and recall are 90% and 71%, respectively. As shown in Table II, *EH-Miner* learns from fewer applications, while the precision is similar to [14]. Acharya $et\ al.$ reported that their tool spent one hour on Postfix, which has 111K lines of code. The efficiency is about nine hours per million lines of code (MLOC). According to Section V, the efficiency of *EH-Miner* is 0.38 hours per MLOC.

Acharya $et\ al.$ mine API-Error Check from error traces when the check conditions are equivalent in the syntactic level. In our study, the syntactically equivalent cases only account for 26% (Figure 2). Additionally, they assume that there is an explicit return/exit statement in an error trace. Our study

TABLE II
COMPARISON OF DIFFERENT TOOLS.

	#Appl.	Precision	Recall	Efficiency
Acharya *et al.*	83	90%	28*	9 H/MLOC
APEx	28	77%	47%	7.7 H/MLOC
EH-Miner	28	90%	71%	0.38 H/MLOC

* The authors did not report the recall.

shows that these two statements only account for 46% (Figure 3). As a result, [14] found error specifications of 28 functions for both POSIX and X11 libraries. While *EH-Miner* found rules of 98.58 functions for POSIX library only, in spite of learning from fewer applications.

Comparison with [13]. Acharya *et al.* [14] is the prior work of APEx [13], which claims that APEx outperforms [14], thus we regard APEx as the state-of-the-art tool. The precision and recall of APEx are 77% and 47%, respectively. APEx spent 11.5 hours on five applications: ClamAV, Pidgin, Grep, GnuTLS and Coreutils, which have 1.5M lines of code in total. The efficiency is about 7.7 hours per MLOC. *EH-Miner* is about 20 times faster than APEx. Though *EH-Miner* uses a constraint solver, the number of variables in each constraint is very limited (only from two check conditions), and *EH-Miner* will not suffer the path explosion problem.

APEx first distinguishes error paths and non-error paths, then analyzes the conditions of error paths and mines error specifications like "for foo(), -1 is an error return value". In contrast to APEx, *EH-Miner* does not need to know whether -1 means error or not, but directly mines rules like "foo() should be logged when returning -1". This feature enables *EH-Miner* to improve the precision by (90%-77%) / 77%, *i.e.*, 17%. On the other hand, the features of judging equivalence of check conditions and recognizing error-handling actions improve the recall from 47% to 71%.

VII. DISCUSSION

Exception-handling mechanism. There are two common practices to handle errors in software. The first practice is to use explicit exception handling mechanism, i.e., try-catch block, which are built-in features in many languages like Java and Python. This mechanism has fixed syntax structure, and has been well studied by many existing works. While EH-Miner is designed as a complement tool to the existing works. EH-Miner focuses on the second practice, i.e., function-return-check, which is wide-used in low-level languages such as C/C++. C does not provide exception-handling mechanisms, while C++ projects contain far fewer try blocks than projects developed in the other languages.

Alternative error-handling method. Sometimes, developers may can use an alternative method to handle an error of an API. For example, in Balsa, developers do not check the return value of the API $notify\_init$. Instead, they use $notify\_is\_initted$ to ensure $Libnotify$ is successfully initialized by $notify\_init$. Another example is $open$ and $stat$. The developers use $stat$ to check the state of a file that is going to be opened by $open$. When $stat$ returns success, $open$ will succeed in most cases. *EH-Miner*, however, can not cope with this relationship, and reports two false-positive cases. To remedy this situation, we will need to study the relationship between API functions further.

Impact-oriented bug detection. *EH-Miner* can find bugs that cause severe results like crash or abort. These results, however, have different impacts. For example, in MonetDB, an error-handling bug of $gethostbyname$ causes a crash, and the bug is classified as a critical bug. On the other hand, in Fossil, a bug of $fopen$ crashes a component $mkversion$. The developer, however, believes this bug is not important, since $mkversion$ is a one-off utility used during the build process, and is called only by Makefile. *EH-Miner* can not avoid the low impact bugs as yet.

VIII. THREATS TO VALIDITY

Threats to external validity. The first threat to external validity is the representativeness of the programs. In the implementation of EH-Miner, we choose as many as possible software domains, and representative programs from those domains to build dataset. The programs use APIs from various libraries, thus EH-Miner can mine error-handling rules of the APIs. The dataset, however, may not be generalized to other datasets, e.g., the Linux kernel do not use libraries. To control this threat, we evaluate EH-Miner on a different dataset – Linux kernel. The second threat to external validity is the limitations of error-handling actions. In Section III, we summarize nine actions from five programs. There might be another action that is not used by any of the five programs, thus our action set may not be generalized to other programs. The impact the this threat is limited. We mine error-handling rules from frequent actions; others will be ignored anyway.

Threats to internal validity. Beyond the factors in PEH model, there are two other factors that can impact error-handling rules: context and argument, which pose threats to internal validity. (a) The context of error-handling actions is the first threat. The choice of error-handling actions might be determined by some application-specific contexts besides check conditions. For example, the $break$ and $continue$ actions should be only used in loops; the $delete$ and $close$ actions should be used when dealing with files or directories; the $return$ action usually will not be used when the caller function returns void. (b) The second threat is argument. Taking $fopen$ as an example, $fopen$ has two arguments: file name and operation mode. Whenever the file name is tainted by any form of input, $fopen$ becomes more likely to fail. For example, the invocation $fopen(input\_file, `r')$ is more likely to fail than $fopen(\text{"}/etc/passwd\text{"}, `r')$, since the $input\_file$ may be a file that does not exist. As for the second argument, 'r' is more likely to fail than 'w'. Given a non-existent file, $fopen$ fails when reading, and creates a new file when writing. We leave the contexts and arguments in future work.

Threats to construct validity. A common practice to evaluate static tools is using precision and recall. For EH-Miner, however, it is hard to measure its recall, since lack of ground truth of false negative. Measuring precision is also

non-trivial, since the results should be inspected by developers manually. In this regard, we use the precision and recall of error-handling rules instead of the precision and recall of error-handling bugs to evaluate EH-Miner. A violation of a right error-handling rule does not have to be a bug, e.g., developers may use an alternative way to handling the error.

IX. Related Works

Error-handling bugs. Existing works of detecting error-handling bugs can be classified into two kinds, $i.e.$, static approaches and dynamic approaches. Many works use static analysis techniques for detecting bugs. Tian $et\ al.$ [4] conducted empirical study on error-handling bugs, and proposed ErrDoc, a tool that can automatically detect, categorize, and fix error-handling bugs. Jana $et\ al.$ [5] designed EPEx, which uses under-constrained symbolic execution to explore error paths and detect error handling bugs. Lawall $et\ al.$ [7] built a static bug finding tool for finding error-handling bugs in SSL. Rubio-Gonzlez $et\ al.$ [2] proposed an inter-procedural static analysis that tracks errors as they propagate through file system code. Gunawi $et\ al.$ [6] developed a static analysis technique that analyzes how file systems and storage device drivers propagate error codes. As error conditions rarely hold feeding regular input, thus the error-handling code rarely gets to run and error-handling bugs are hard to test. To fill this gap, researchers have used dynamic fault injection to execute error-handling code by simulating failures. Marinescu $et\ al.$ [9]–[11] proposed a dynamic analysis technique and developed a fault injection tool. They simulate API failures according to the error specifications and check the response of the test program. Jia $et\ al.$ [30] designed and implemented SmartLog, an intention-aware log automation tool to place log statements for error-prone functions.

Both static and dynamic works depend heavily on correct error specifications, while *EH-Miner* can detect error-handling bugs without error specifications.

Exception-handling bugs. There is a long line of research focuses on exception-handling mechanisms, which are built-in features in many languages like Java and Python. Many researchers study the characteristics of exception-handling bugs. Ebert $et\ al.$ [31] conducted a survey of 154 developers and an analysis of 220 exception handling bugs from the repositories of Eclipse and Tomcat. Oliveira $et\ al.$ [32] presented an empirical study on the relationship between the usage of Android abstractions and uncaught exceptions. Filho $et\ al.$ [33] presented an in-depth study of the adequacy of the AspectJ language for modularizing exception handling code. Jakobus $et\ al.$ [12] contrasted exception handling code across languages from 50 open source projects. Some works are aimed to detect and repair exception-handling bugs. Barbosa $et\ al.$ [34] presented a tool to recommend repairs of exception handling violations with aware of the global context. Gu $et\ al.$ [35] expanded the intrinsic capability of runtime error resilience in software systems to recovery unexpected errors at runtime. Yan $et\ al.$ [36], [37] proposed a slice-based statistical fault localization approach to improve fault localization effectiveness. Weimer $et\ al.$ [38], [39] presented a data-flow analysis for finding a certain class of exception-handling bugs in Java programs. Jia $et\ al.$ [40] designed a tool that can automatically detect and pinpoint the root causes of the problems caused by ungraceful exits.

These works focus on exception-handling mechanisms like $try\text{-}catch$ statements. While *EH-Miner* detects bugs for error handling of function return, which is widely used in languages like C/C++.

API specifications. Another line of research related to *EH-Miner* is API specifications mining. Some works are aimed to automatically mine error specifications. DeFreez $et\ al.$ [41] proposed a path-based embedding method to determine function synonyms, and predicted the functions that should be executed when an error occurs. This method, however, can not predict error conditions. Kang $et\ al.$ [13] studied the characteristics of error paths and proposed APEx, which can automatically infer error specifications for C API functions. Acharya $et\ al.$ [14] automatically inferred error handling specifications of APIs by mining static traces of their runtime behaviors. Rubio-Gonzlez $et\ al.$ [15] examined mismatches between documented and actual error codes, and revealed undocumented error-code instances. Magiel $et\ al.$ [42] focused on a specified embedded system, and presented a fault model accompanied by an analysis tool. Several prior research projects have mined different types of specifications from source code to help software developers. Nguyen $et\ al.$ [43] mined large-scale repositories of existing open-source software to derive potential preconditions for API methods. Zhong $et\ al.$ [44] and Tan $et\ al.$ [45] tried to infer API specifications from user manuals and comments. Acharya $et\ al.$ [46] proposed a framework to automatically extract usage scenarios among user-specified APIs as partial orders. Engler $et\ al.$ [47] demonstrated techniques that automatically extract programing rules from the source code.

EH-Miner is closer to works of mining error specifications: [13] and [14]. Compared with these works, *EH-Miner* has three advantages: higher accuracy, better scalability and easier to repair.

X. Conclusions

The existing methods of detecting error-handling bugs are limited by the need for correct error specifications input. This prerequisite is hard to satisfy, since manually generating error specifications is error-prone and tedious, while mining error specifications automatically is often inaccurate. To fill the gap, we designed and implemented *EH-Miner*, a tool that can automatically detect error-handling bugs without requiring error specifications as input. We applied *EH-Miner* to 117 mature applications. It was found that *EH-Miner* mined error-handling rules with precision of 91.1% and recall of 46.9%. We reported 142 bugs mined by *EH-Miner*; of these, 106 had been confirmed and fixed by developers. We also reported 68 bugs Linux kernel-4.17, of which 42 were confirmed or fixed by developers.

REFERENCES

[1] M. P. Robillard and G. C. Murphy, "Designing robust java programs with exceptions," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, ser. SIGSOFT '00/FSE-8. New York, NY, USA: ACM, 2000, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/355045.355046

[2] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 270–280. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542506

[3] "Owasp," https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf, 2018.

[4] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 752–762. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106300

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

[6] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "Eio: Error handling is occasionally correct." in *FAST*, vol. 8, 2008, pp. 1–16.

[7] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller, "Finding error handling bugs in openssl using coccinelle," in *Dependable Computing Conference (EDCC), 2010 European*. IEEE, 2010, pp. 191–196.

[8] P. Broadwell, N. Sastry, and J. Traupman, "Fig: A prototype tool for online verification of recovery mechanisms," in *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[9] P. Marinescu, R. Banabic, and G. Candea, "An extensible technique for high-precision testing of recovery code," in *Proceedings of the USENIX annual technical conference*, no. EPFL-CONF-147244, 2010.

[10] P. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 11:1–11:38, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063509.2063511

[11] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 379–388.

[12] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. de Lucena, "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)*, Nov 2015, pp. 183–193.

[13] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 472–482. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970354

[14] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *Fundamental Approaches to Software Engineering*, M. Chechik and M. Wirsing, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 370–384.

[15] C. Rubio-González and B. Liblit, "Expect the unexpected: Error code mismatches between documentation and the real world," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: ACM, 2010, pp. 73–80. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806687

[16] D. analysis, https://en.wikipedia.org/wiki/Dependence_analysis, 2018.

[17] "Reaching definition," https://en.wikipedia.org/wiki/Reaching_definition, 2018.

[18] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/1995376.1995394

[19] "Use define chain," https://en.wikipedia.org/wiki/Use-define_chain, 2018.

[20] "Clang," https://clang.llvm.org, 2018.

[21] "Llvm," https://www.llvm.org, 2018.

[22] "Z3," https://github.com/Z3Prover/z3, 2018.

[23] "Sqlite3," https://www.sqlite.org, 2018.

[24] "Cloc," http://cloc.sourceforge.net, 2018.

[25] "Eh-miner," https://github.com/ZhouyangJia/EH-Miner, 2019.

[26] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[27] F. scaling, https://en.wikipedia.org/wiki/Feature_scaling, 2018.

[28] Coverity, https://scan.coverity.com, 2018.

[29] Fortify, https://software.microfocus.com/en-us/solutions/application-security, 2018.

[30] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 61–71.

[31] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82 – 101, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215000862

[32] J. Oliveira, D. Borges, T. Silva, N. Cacho, and F. Castor, "Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions," *Journal of Systems and Software*, vol. 136, pp. 1 – 18, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217302558

[33] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, "Exceptions and aspects: the devil is in the details," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT'06/FSE-14*. ACM New York, NY, USA, 2006, pp. 152–162.

[34] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 858–858. [Online]. Available: http://doi.acm.org/10.1145/3180155.3182539

[35] T. Gu, C. Sun, X. Ma, J. L, and Z. Su, "Automatic runtime recovery via error handler synthesis," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 684–695.

[36] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *J. Syst. Softw.*, vol. 89, no. C, pp. 51–62, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.08.031

[37] Y. Lei, C. Sun, X. Mao, and Z. Su, "How test suites impact fault localisation starting from the size," *IET Software*, vol. 12, no. 3, pp. 190–205, 2018.

[38] W. Weimer and G. C. Necula, "Exceptional situations and program reliability," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, pp. 8:1–8:51, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1330017.1330019

[39] W. Weimer and G. C. Necula, "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*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 419–431. [Online]. Available: http://doi.acm.org/10.1145/1028976.1029011

[40] Z. Jia, S. Li, T. Yu, X. Liao, and J. Wang, "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*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 751–762. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338938

[41] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 423–433. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236059

[42] M. Bruntink, A. van Deursen, and T. Tourwé, "Discovering faults in idiom-based exception handling," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06.

New York, NY, USA: ACM, 2006, pp. 242–251. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134320

[43] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 166–177. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635924

[44] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–318. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.94

[45] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 145–158. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294276

[46] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287630

[47] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 57–72. [Online]. Available: http://doi.acm.org/10.1145/502034.502041