Automatically Diagnosing and Repairing Error Handling Bugs in C

Yuchi Tian, Baishakhi Ray
University of Virginia
Charlottesville, VA, USA 22903
{yuchi,rayb}@virginia.edu

ABSTRACT

Correct error handling is essential for building reliable and secure systems. Unfortunately, low-level languages like C often do not support any error handling primitives and leave it up to the developers to create their own mechanisms for error propagation and handling. However, in practice, the developers often make mistakes while writing the repetitive and tedious error handling code and inadvertently introduce bugs. Such error handling bugs often have severe consequences undermining the security and reliability of the affected systems. Fixing these bugs is also tiring—they are repetitive and cumbersome to implement. Therefore, it is crucial to develop tool supports for automatically detecting and fixing error handling bugs.

To understand the nature of error handling bugs that occur in widely used C programs, we conduct a comprehensive study of real world error handling bugs and their fixes. Leveraging the knowledge, we then design, implement, and evaluate ErrDoc, a tool that not only detects and characterizes different types of error handling bugs but also automatically fixes them. Our evaluation on five open-source projects shows that ErrDoc can detect error handling bugs with 100% to 84% precision and around 95% recall, and categorize them with 83% to 96% precision and above 90% recall. Thus, ErrDoc improves precision up to 5 percentage points, and recall up to 44 percentage points *w.r.t.* the state-of-the-art. We also demonstrate that ErrDoc can fix the bugs with high accuracy.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Empirical software validation*; • **Theory of computation** → *Program analysis*;

KEYWORDS

error handling bugs, bug fix, API errors, bug detection

ACM Reference format:
Yuchi Tian, Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106300

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106300

1 INTRODUCTION

Secure and reliable software must handle all possible failure conditions correctly. Modern programming languages, therefore, provide exception handling features so that software can behave gracefully even when lower-level functions fail. Unfortunately, low-level languages like C do not have any native error handling primitives. Therefore, in C, the developers are forced to create their own error handling conventions, often in ad-hoc program-specific manner. Such error handling code is usually repetitive and tedious to write correctly, resulting in error handling bugs. These bugs often lead to serious security and reliability flaws (*e.g.,* CVE-2014-0092 [4], CVE-2015-0208 [5], CVE-2015-0285 [6], CVE-2015-0288 [7], and CVE-2015-0292 [8]). In fact, improper error handling is one of the top 10 causes of security vulnerabilities according to The Open Web Application Security Project (OWASP) [26]. Common testing techniques usually fail to detect these bugs as most of the errors do not show up during regular executions. Moreover, manually creating test cases to cover all possible error paths is not scalable for large real-world programs.

Due to severe implications of error handling bugs, prior researchers have put significant effort in detecting them automatically. However, we still see a steady influx of error handling bugs several of which result in security vulnerabilities. For example, there are already two Common Vulnerability and Exposure (CVE) reports in 2017 (CVE-2017-3318 [9], CVE-2017-5350 [10]) about error handling bugs. In order to understand the true nature of these bugs in practice, we begin with a comprehensive study of real error handling bugs from 6 large open-source software. We create a taxonomy of these bugs to understand their underlying causes. Our results indicate that error handling bugs usually occur due to four causes: Incorrect/Missing Error Checks, Incorrect/Missing Error Propagation, Incorrect/Missing Error Outputs, and Incorrect/Missing Resource Release. While existing bug finding tools can partially detect some of these bugs, a large proportion of them remains yet undetected.

Leveraging the findings of our study, we design and implement a tool, ErrDoc, that can detect and categorize all classes of error handling bugs. Using under-constrained symbolic execution, ErrDoc first explores all the error paths—the paths along which a function can fail. If a function fails, the error needs to be handled properly along the error path. To ensure that, ErrDoc uses a combination of static analysis techniques and verify whether the error value returned from the failing function is checked, propagated upstream, or logged. If none of these happen, ErrDoc reports error handling bugs. ErrDoc further ensures that a program can fail gracefully by releasing all the allocated resource. If resources are not freed along an error path, ErrDoc reports those cases as bugs. We find that ErrDoc can detect error handling bugs with 100% to 84% precision

and around 95% recall. To provide developers more information about the underlying causes of the bugs, ErrDoc also categorizes these bugs with 83% to 96% precision and above 90% recall.

In our case study, we further notice that fixing error handling bugs is a repetitive and tenuous process. Most functions have their own error handling protocol. To fix an error handling bug, the developer not only has to fix the bug correctly but also follow the existing protocol to improve code readability and decrease maintenance overhead. Therefore, the fixes to error handling bugs themselves often introduce new bugs. In this paper, we propose an algorithm to automatically generate patches for fixing different types of error handling bugs. This algorithm modifies the Abstract Syntax Tree (AST) of the buggy code to introduce the bug-fixes. One of the key characteristics of our patch generation algorithm is that the generated patches not only fix the corresponding error handling bugs but also blend into the existing error handling code seamlessly. Therefore, such patches are more useful to the developers as demonstrated by the fact that several of our automatically generated patches are already accepted by OpenSSL developers as bug-fixes without any further modifications. Overall, the bug-fixing phase generates acceptable patches with 72% to 84% precision. To this end, we make the following contributions:

- We conduct a comprehensive study to understand the characteristics of error handling bugs. We present, to the best of our knowledge, the first classification of error handling bugs into four different categories—Incorrect/Missing Error Propagation, Incorrect/Missing Error Checks, Incorrect/Missing Error Outputs, and Incorrect/Missing Resource Release.
- We design and implement ErrDoc, a tool to automatically detect, categorize, and fix error handling bugs. To our knowledge ErrDoc is the first tool that can automatically categorize and fix error handling bugs.
- We present an extensive evaluation of ErrDoc on five large open-source software including OpenSSL, GnuTLS, WolfSSL, Curl, and Httpd. We find 106 new bugs in OpenSSL. We are in the process of reporting them. Three of the patches generated by ErrDoc have already been accepted by the developers.

The rest of the paper is organized as follows. Section 2 presents the comprehensive study on real-world error handling bugs. Section 3 introduces the tool ErrDoc, followed by its implementation and evaluation in Section 4 and Section 5 respectively. We discuss related work in Section 7 and conclude our work in Section 8.

2 MANUAL STUDY

To better understand what kind of error handling bugs occur in real C code and how developers fix them in practice, we manually studied 145 real-world error handling bugs and their fix patches. In this section, we will discuss our study method and findings in detail.

Study Subject. We collected version history from 6 open source projects written in C. Table 1 summarizes the study subjects. In total, we have studied around 13M LOC, and 30K commits with a parallel development history of up to 8 years.

Study Method. For each studied project, we begin with extracting all the changes committed within the studied time period. A change typically contains a commit log and a patch (see Table 3) along with other meta information. From these changes, we try to identify error handling bug-fix related commits. First, from the commit messages, we remove stopwords, punctuations, and other program-identifier-related information, and stem the commit message using standard natural language processing (NLP) techniques. We mark the corresponding commit to be bug-fix-related if the processed commit messages contain at least one of the following keywords: 'bug', 'fix', 'check', 'issue', 'mistake', 'incorrect', 'fault', 'defect', and 'leak'. Among the identified bug-fix commits, we further search for some combinations of error handling bug-related keywords: 'error path', 'memory leak', 'unchecked return', 'error handl', 'error check', and 'check return'. For example, we identify an OpenSSL commit (sha: 0a618df) with commit message :"*Fix a mem leak on an error path*" as a error handling bug-fix commit.

Table 1: Study Subjects

Project	LOC	Studied period	Total commits	Total bug-fixes	EH bugs
OpenSSL	469,525	2016-01-01 to 2017-01-01	3925	924	126
GnuTLS	168,777	2002-01-01 to 2010-01-01	7035	760	29
WolfSSL	166,667	2016-01-01 to 2017-01-01	1240	297	31
Curl	153,732	2008-01-01 to 2017-01-01	11654	2853	190
Httpd	1,832,007	2012-01-01 to 2017-01-01	6781	1049	70
Linux	10,462,319	2016-12-01 to 2017-01-01	3234	1377	263
Total	13,253,027		33,869	7260	709

To evaluate the precision of the above classification, we randomly choose 120 commits (20 from each project) that are marked as error handling bug-fix. We manually verify the commit messages and their corresponding patches. Out of 120 commits, 115 are classified correctly; thus, giving a precision of 95.83%. Only 5 commits are misclassified. In all these misclassified commits, although the relevant keywords are present in their commit messages, they were not related to error handling bugs. For example, commit message of one misclassified Httpd change (sha: 068854a; message: "*APR_HAVE_foo is checked via #if, not #ifdef ... This fixes a compile error*") contains 'fix', 'check', and 'error' keywords; however, it was referring to a compilation problem.

Table 2: Studied Error Handling Bugs & Patches

Projects	EC	EP	RR	EO	Others	Total
OpenSSL	3	2	10	2	3	20
GnuTLS	10	1	4	0	2	17
Curl	12	2	16	5	5	40
Httpd	6	2	7	3	2	20
WolfSSL	9	1	5	2	1	18
Linux	9	2	9	6	4	30
Total	49 (33.79%)	10 (6.90%)	51 (35.17%)	18 (12.41%)	17 (11.72%)	145

Characterization of the Bugs & Fixes. In our study, we found total 709 error handling bugs across all the 6 projects (see Table 1). Among these bugs, we randomly choose 145 bugs and manually investigate them (both commit messages and patches) to understand the reason behind these errors. Based on this study, we identify four different categories of error handling bugs: Incorrect/Missing Error Checks (EC), Incorrect/Missing Error Propagation (EP), Incorrect/Missing Error Outputs (EO), and Incorrect/Missing Resource Release (RR) (see Table 2). We use the code snippets in Table 3 to discuss each of the categories below.

1. **EC: Incorrect/Missing Error Checks**: When an API function, say API_{test}, fails, it usually indicates the failure to its caller by returning an appropriate error code. Thus, after calling API_{test}, the caller should check its return value; in case, API_{test} returns an error code (as per API_{test} specification), the caller should handle the failure properly. However, we find that developers often forget to check

Table 3: Examples of Error Handling Bugs of different types

1. EC-A : Missing Error Checks

Project: OpenSSL
Commit ID: b197257d71694fd52ab61d173f77c8a120d3eead
File: crypto/ocsp/ocsp_ext.c
Author: Matt Caswell ⟨matt@openssl.org⟩
Date: 08/22/2016
Log: Check for error return from ASN1_object_size

```
   os.length = ASN1_object_size(0, len, ...);
+  if (os.length < 0)
+    goto err;
   ...
```

EC-B : Incorrect Error Checks

Project: Curl
Commit ID: 520833cbe1601feed1c6473bd28c4c894e7ee63e
File: lib/ssluse.c
Author: Mike Giancola ⟨mikegiancola@gmail.com⟩
Date: 05/22/2013
Log: ossl_recv: SSL_read() returning 0 is an error too

```
   nread = (ssize_t)SSL_read(...);
-  if(nread < 0) {
+  if(nread <= 0) {
     int err = SSL_get_error(...);
```

2. EP : Incorrect/Missing Error Propagation

Project: OpenSSL
Commit ID: e0670973d5c0b837eb5a9f1670e47107f466fbc7
File: ssl/ssl_ciph.c
Author:
Date: 02/05/2017
Log: mem leak on error path and error propagation fix

```
int SSL_COMP_add_compression_method() {
  ...
  if (id < 193 || id > 255) {
    SSLerr(...);
-     return 0;
+     return 1;
  }
```

3. EO : Incorrect/Missing Error Outputs

Project: WolfSSL
Commit ID: fa5dd0100146222a43d7562fdb2c600f481eaecf
File: wolfcrypt/test/test.c
Author: David Garske ⟨david@wolfssl.com⟩
Date: 05/04/2016
Log: ... Added error message for ECC test failures, to show the curve size used ...

```
  ret = ecc_test_curve_size(rng, keySize,...)
  if (ret < 0) {
+   printf("ecc_test_curve_size %d failed!:%d",
+                 keySize, ret);
    return ret;
  }
```

4. RR : Incorrect/Missing Resource Release

Project: OpenSSL
Commit ID: 85d6b09ddaf32a67a351521f84651c3193286663
File: crypto/srp/srp_lib.c
Author: Matt Caswell ⟨matt@openssl.org⟩
Date: 08/22/2016
Log: Fix mem leak on error path

```
BIGNUM *SRP_Calc_u(...)
{
  ...
  if ((cAB = OPENSSL_malloc(2 * longN)) == NULL)
    goto err;
  ...
  err:
+   OPENSSL_free(cAB);
    EVP_MD_CTX_free(ctxt);
    return u;
}
```

the API return values. Table 3 EC-A shows such an example from OpenSSL. Here, the caller function calls API `ASN1_object_size` that returns < 0 on failure. The caller function did not check whether the API returns an error code, which causes an error handling bug. Developers fixed the bug in subsequent commit as shown in the example.

Moreover, developers sometimes check some return values, but not the entire range of possible error codes that API_{test} can return, as shown in example EC-B in Table 3. The API function SSL_read can return ≤ 0 as error code. However, the developer only checked for < 0, which causes the error handling bug. Developer fixed the bug in the commit shown in the example by adding the check for zero. As per Table 2, 33.79% of the studied bugs belong to this EC category.

Fix: The fixed patches usually include the missing check of the API return value using if statement or correcting the if condition to check all possible error codes. In the corresponding error handling code (*i.e.* the code within if block), developers usually output an error message notifying users about the failing condition, or propagate an error value upstream using an appropriate error return code.

2. **EP: Incorrect/Missing Error Propagation**: To handle an API failing condition correctly, a caller may propagate the error code upstream using an appropriate return value to inform the rest of the system about the failure. However, we find that developers often forget to propagate the correct error code. Also, due to complicated logic in the caller function, the return value with correct error code may get overwritten with non-failure values, or the caller may return from another successful path by ignoring previous failing conditions. For example, see EP case in Table 3. The caller function `SSL_COMP_add_compression_method` should return 1 on error, and 0 on non-error. However, the developer returned 0 from an error path by mistake, which was fixed in the subsequent patch. Table 2 shows that only 6.90% of the studied bugs belong to this category.

Fix: The fixed patches ensure to return the correct error values, as per the callers' error specifications.

3. **EO: Incorrect/Missing Error Outputs**: An API failure can be handled by logging/outputting an appropriate error message so that the users become aware of the failure. However, in error handling code developers often forget to output an appropriate error message—either no error message is logged or outputted, or the error message is incorrect, unclear or lacks details. Consider the EO example in Table 3, the WolfSSL developer forgot to output an error message; he later corrected it by showing the `keySize` and return value. 12.41% of the studied bugs belong to this category (see Table 2).

Fix: The fixes often include adding statements to output error messages or changing error messages with more specific details.

4. **RR: Incorrect/Missing Resource Release**: Even if developers have logged or propagated error code correctly, they often forget to release locally allocated resources (*e.g.,* free memory) in the error handling code, causing RR bugs. As shown in example RR in Table 3, developer allocated *cAB* using openssl_malloc but forgot to free that in the corresponding error handling code under *err* label. The highest number of error handling bugs (35.17%) come from Incorrect/Missing Resource Release category (see Table 2).

Fix: To fix RR bugs, the patches usually include adding the specific resource-free function calls in the error handling code. Sometimes, developers put all the free statements in one block and call them using `goto` statement from the error handling code to free all the resources together.

5. **Other error handling bugs**: 17 out of 145 (11.72%) studied bugs arise due to some other issues in error handling code, which may not be directly related to error handling bugs. These include fixing double free mistakes, refactoring or changing styles of error handling code, and improving/cleaning error handling code.

Generic observation on bug-fix patches. Although the error handling bug-fix patches, as discussed above under each category, may seem to be quite straightforward to fix, we notice that a real patch often shares similar characteristics of other error handling code in its immediate context. For example, if a caller function implements a common error handling code using `err` label (see example RR in table 3), a fix patch often calls `goto err` and implement the relevant error handling code within the common error handling block under label `err`. Even if semantically correct, a fix-patch does not get accepted by the project developers, if the patch does not follow the protocol followed by other error handling code in the same local context.

Threats to Validity. We investigate the error handling bugs in the studied projects for a given time period, so our study may not cover all the error handling bugs and patches in these projects. The collection of error handling bugs and patches is based on NLP techniques and keyword matching, developers may not put the keywords that we look for in the commit messages. Also, our manual categorization may be affected by inspectors' biases.

3 APPROACH

Here, we introduce how ErrDoc (Error Doctor) detects and diagnoses potential error handling bugs in C programs and also fixes them. If a function call, say f, fails for some reason and returns an error to its caller, say f_{caller}, the caller must handle the error properly; otherwise, an error handling bug may occur. If f_{caller} fails to do so, an error handling bug may occur. We begin with defining several key terms used in this work.

Definition 3.1. **Control Flow Graph (CFG):** A CFG of a function in the program is a directed-graph represented by a tuple ⟨ N, E ⟩. N is the set of nodes, where each node is labeled with a unique program statement. The edges, E ⊆ N × N, represent possible flow of execution between the nodes in the CFG. Each CFG has a single begin, n_{begin} , and end, n_{end}, node. All the nodes in the CFG are reachable from the n_{begin} node and the n_{end} node is reachable from all nodes in the CFG. [27]

Definition 3.2. **Path :** A path P is a sequence of nodes ⟨ $n_0, n_1, ...n_j$ ⟩ in a CFG, such that there exists an edge between n_k and n_{k+1}, *i.e.* (n_k,n_{k+1}) ∈ E, for k = 0,...,j-1 [25].

Definition 3.3. **Error Path:** For a function pair f and its caller f_{caller}, an error path $Path_{err}(f_{caller},f)$ is a path that starts from n_{begin} of f_{caller}'s CFG, contains the function call to f, follows a branch in f's CFG along which f's error conditions are satisfied, and ends at n_{end} of f_{caller}'s CFG.

Note that, an error path spans over the CFGs of both f_{caller} and f. In Figure 1, callee `EVP_Digest` returns error along path `S2, S4, S5`; The corresponding error path (marked in **red**) spans over the CFGs of both caller `ASN1_Digest` and `EVP_Digest`: ⟨`L0, L1, S2, S4, S5, L3, L4`⟩.

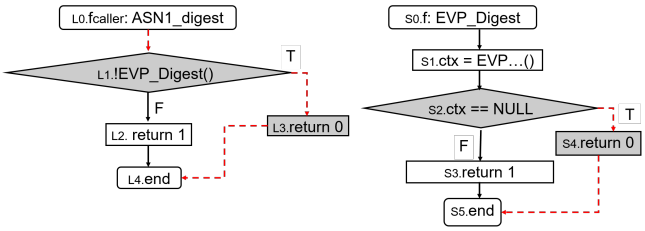

Figure 1: sample error path (marked in red) and error handling code (marked in gray) in the CFGs of f_{caller} and f. Here 0 indicates failure, while 1 indicates success.

Definition 3.4. **Non-Error Path:** For a function pair f and its caller f_{caller}, a non-error path $Path_{nerr}(f_{caller},f)$ is a path that starts from n_{begin} of f_{caller}'s CFG, contains the function call to f, follows a branch in f's CFG along which f's error conditions are not satisfied, and ends at n_{end} of f_{caller}'s CFG.

In the above example of Figure 1, $Path_{nerr}(f_{caller},f)$ is marked as ⟨`L0, L1, S2, S3, S5, L2, L4`⟩.

Definition 3.5. **Error Handling Code:** For a function pair f and its caller f_{caller}, error handling code is a sub-graph in f_{caller}'s CFG along the error path $Path_{err}(f_{caller},f)$, that explicitly checks the error value returned by f and performs a special processing.

Here, f can return error code by explicit `return statement` or changing the value of its function arguments if called by reference. In Figure 1, gray nodes indicate error handling code.

3.1 Overview

An overview of ErrDoc's workflow for a target function f is shown in Figure 2. ErrDoc takes five inputs: the signature of a fallible function (f) and their caller functions (f_{caller}), f's error specifications ($f_{errSpec}$), global non-error specification ($Global_{nerrSpec}$) and a list of logging functions (Loggers). A list of fallible functions that need to be tested and loggers are created manually, $f_{errSpec}$ and $Global_{nerrSpec}$ are generated by either APEx [16] or entered manually, and the caller functions are automatically detected by project specific call-graph.

ErrDoc then works in three phases. In Phase-I, given each caller and callee pair, ErrDoc detects all possible error paths (Step I-a), identifies error handling code along each error path (Step I-b), and looks for function pairs that often occur together along error paths (Step I-c). ErrDoc then detects and categorizes different types of error handling bugs in Phase-II. Finally, in Phase III, ErrDoc leverages the knowledge from previous phases to fix the bugs. The rest of the section discusses each of the phases in details. Listing 1 serves as a motivating example to illustrate each step.

3.2 Phase-I. Explore Error Paths

This phase works in three steps: identifying error paths, detecting error handling code and function pairs along those paths.

Step I-a. Identify Error Paths. For a given function under test (f), its caller function (f_{caller}), and error specifications ($f_{errSpec}$) as input, ErrDoc symbolically executes f_{caller} and checks if f is

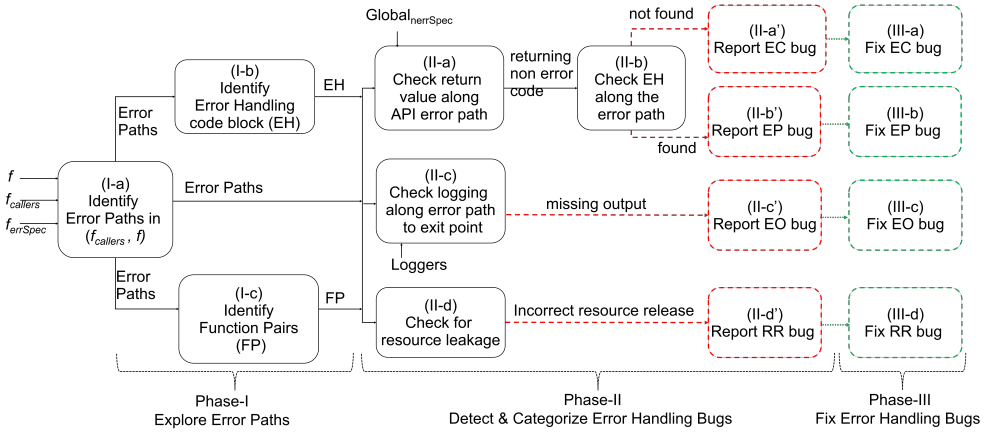

Figure 2: ErrDoc Workflow

called at each method call. If found, ErrDoc symbolically executes f and checks its return value (f_{ret}) at the call site along all the possible return paths. If $f_{ret} \wedge f_{errSpec}$ is satisfiable, then the corresponding path is marked as an *error path*, otherwise it is marked as a non-error path. ErrDoc marks all the corresponding CFG nodes and edges along the error path. This step is adapted from our previous tool [14].

Step I-b. Identify Error Handling Code (EH) along an Error Path. Along an error path $Path_{err}(f_{caller}, f)$, ErrDoc determines the corresponding error handling code in f_{caller}. This is done at AST level, *i.e.* ErrDoc identifies an AST subtree corresponding to the error handling code. First, ErrDoc identifies the AST sub-tree of f_{caller} that constitutes the error path. It then identifies the AST nodes related to f method call. In particular, ErrDoc matches the node type to be 'Call Expression' and node value to be f. Let's call such node as f_{ast}. Next, ErrDoc identifies the error handling code based on three heuristics.

- Along the error path, if f_{ast} has an immediate parent `if` statement and f_{ast} is inside the condition part of the if statement, then ErrDoc considers the corresponding `then` or `else` part (whichever falls in the error path) of the AST sub-tree as potential error handling code. For example, `if(f()) then {EH code}` (see lines 22 to 29 in Listing 1).
- Along the error path, if the variable storing return value of f is inside the condition part of a f_{ast}'s subsequent if statement, then ErrDoc considers the corresponding `then` / `else` part (whichever falls in the error path) of the AST sub-tree as potential error handling code. For example, `int b = f(); if (b < 0) then {EH code}` (see lines 9 to 13 in Listing 1).
- Since error paths usually contain significantly fewer statements than non-error paths [16], ErrDoc considers the number of statements in the identified error handling code should be ≤ 5. Otherwise, ErrDoc disregards the code block as error handling code.

Step I-c. Identify Function Pairs (FP) along Error Paths. Certain resource related functions like `malloc` and `free`, or `lock` and `unlock` usually occur in pairs. Researchers identified such function pairs by mining source code [20, 34]. However, identifying such pairs from all the program paths may introduce noise. Here, we focus on only the error paths. Our intuition is that if a resource is allocated in a function path, that resource is, in general, deallocated before the function exits or returns, for both error and non-error paths. Since an error path is usually shorter than other program paths and contains less number of function calls [16], our search space of identifying function pair will be significantly reduced. In addition, along a non-buggy error path, the path before function call f corresponds to a regular path and can allocate a resource; In contrast, the path after f contains error handling code (see Figure 1) and usually deallocates the resource before it returns or exits. Thus, f gives a natural separation between functions related to resource allocation and de-allocation, and thus, facilitates pair-wise analysis.

In this step, ErrDoc collects the function call sequences along an error path that belong to f_{caller} CFG. ErrDoc divides the call sequence into two sub-sequences based on whether a function is called before or after f. For example, the call sequence ⟨`f1 f2 f2 f3` f `f4 f5 f3`⟩ is divided into before and after subsequences ⟨`f1 f2 f2 f3`⟩ and ⟨`f4 f5 f3`⟩ respectively. Next, all the functions of the two subsequences are paired with each other. However, if the occurrences of a function and its pair vary in the two subsequences, ErrDoc disregards them, as function pairs are unlikely to have different frequencies in a program path. Thus, all `f2` related pairs from the above example are disregarded, as `f2` occurred twice in before subsequence. Also, if a function, *e.g.*,`f3`, occurs both before and after subsequences, it is filtered out. At the end of this step, a potential set of function pairs are identified, *e.g.*,⟨`f1,f4`⟩ and ⟨`f1,f5`⟩.

ErrDoc collects all the function pairs from all the error paths in a studied project and note their corresponding occurrence frequency. ErrDoc then discards all the function pairs with frequency 1, because these pairs are probably accidental. Next, for any given function, ErrDoc tries to find top-n function pairs. Thus, for a given right hand side function in the function pairs, ErrDoc sorts its pairs based on their frequency in descending order and selects top-n frequent function pairs. The same process is repeated on the left hand side function element of the function pairs.

Note that ErrDoc identifies function pairs with one-to-one, one-to-many, many-to-one or even many-to-many mappings. For example, in OpenSSL project, both functions `BIO_new` and `BIO_new_file`

are paired with `BIO_free`. Similarly, function `CRYPTO_malloc` is matched with both `CRYPTO_free` and `CRYPTO_clear_free`.

Next, ErrDoc checks how the two functions in a pair are related by analyzing data-dependency between their arguments and return values along any related error path and represents the relation using *function pair signature*. For example, if for a function pair FP, the first argument of the right function is data dependent on the return value of the left function, the signature will be: $\langle \mathtt{FP}_{left}, \ \mathtt{ret}, \ \mathtt{FP}_{right}, \ \mathtt{arg1} \rangle$. If the paired functions are not data-dependent they are eliminated from further considerations, because they are unlikely to handle the same resources and thus, are not related to our purpose.

3.3 Phase-II. Detect & Categorize Error Handling Bugs

Leveraging information gathered from previous steps, this phase outputs buggy program location and corresponding bug category. To detect EP and EO bugs, ErrDoc extends our previous tool [14].

Incorrect/Missing Error Propagation (EP): If along an error path $Path_{err}(f_{caller}, f)$, ErrDoc finds a return statement in f_{caller}, ErrDoc checks the return value to ensure an error value is pushed upstream. ErrDoc takes the global non-error specification ($\text{Global}_{nerrSpec}$) of the program under study as input. While symbolically executing the error path, ErrDoc checks whether the return value at f_{caller} can contain non-error values according to $\text{Global}_{nerrSpec}$, *i.e.* constraints on return value satisfies $\text{Global}_{nerrSpec}$. If it can contain only non-error values, then ErrDoc marks the corresponding return location as a source of EP bug. If the return statement can contain both error or non-error values, ErrDoc marks the location as *maybe* EP bug, while if it correctly contains only error values, it is marked as not a bug. In the case of a maybe bug, ErrDoc further checks whether the return statement is part of an error handling code; If true, it marks a maybe bug to EP bug, since error handling code should push an error value upstream.

For example, in Listing 1, ErrDoc finds that at line 27, the function is returning a success code (according to error specification shown in line 1-4). However, it is along an error path of `EVP_Digest1` routine and also within the corresponding error handling code (line 22 to 29). Thus ErrDoc marks the return statement at line 27 as potential source of EP bug and the corresponding error path as buggy.

Incorrect/Missing Error Checks (EC). If for functions f and its caller f_{caller}, an EP bug is diagnosed in the previous step and along the corresponding buggy error path in f_{caller} no error handling code for f is found, ErrDoc reports it as a potential missing EC bug.

For example, in Listing 1, corresponding to function call `EVP_Digest` at line 15, an error path (in the buggy version) is along the line `5,7,8,9,15,22,31,32`. However, there was no error check in this path. Thus, ErrDoc reports line 15 as missing EC bug.

An error check can be incorrect. If for functions f and its caller f_{caller}, an EP bug is diagnosed in the previous step and also an error handling code is found along an error path, but the buggy EP location reported in previous step is not part of the error handling code, then the only reason behind the EP bug is an incorrect error check. ErrDoc diagnoses such cases as incorrect EC.

Incorrect/Missing Error Outputs (EO). If along an error path $Path_{err}(f_{caller}, f)$, an exit statement is encountered, ErrDoc makes sure the error situation is logged before exiting. In this step, program specific logging utilities are taken as input. ErrDoc then checks whether a logging function is called before exiting. ErrDoc also checks the constraints on the symbolic arguments of the exit function to ensure they can have error values [14].

For example, in Listing 1, before exiting at line 12 on `OPENSSL_malloc` failure, nothing was logged in the buggy version. ErrDoc diagnoses such case as EO bug.

Listing 1: Motivating Example: Adapted from OpenSSL. The comments and lines in red show the bugs and their categories; the corresponding fixes are shown in green.

```
1   /*
2   Error Spec: returns 0 on error
3                       1 on success
4   */
5   int ASN1_digest(...) {
6
7    int i = i2d(data, NULL);
8    unsigned char *str = OPENSSL_malloc(i);
9    if (str == NULL) {
10     /* EO bug: exiting without logging */
11  +  ASN1err(..., ERR_R_MALLOC_FAILURE);
12     exit(1);
13   }
14
15  - EVP_Digest(...);  /* EC bug: No error check */
16  + if (!EVP_Digest(...)) {
17  +  OPENSSL_free(str);
18  +  return (0);
19  + }
20
21
22   if (!EVP_Digest1(...)) {
23     /* RR bug: Not freeing str */
24  +  OPENSSL_free(str);
25
26     /* EP bug: return success code on error path */
27  - return (1);
28  + return (0);
29   }
30
31   OPENSSL_free(str);
32   return (1);
33  }
```

Incorrect/Missing Resource Release (RR). Given a list of function pairs (FP) and function pairs signatures, as identified in Section 3.2, ErrDoc checks for potential violation of the pairs along an error path $Path_{err}(f_{caller}, f)$. In particular, along the error path in f_{caller}'s CFG, for every function call appears before f, ErrDoc checks whether there is any corresponding paired function(s) from the FP list. If found, ErrDoc then looks for those paired functions along this error paths and checks the corresponding signatures along the error path in f_{caller}'s CFG. If none of the paired functions are found or none of the found paired functions satisfies any of the data dependency specified in the function pairs signatures, ErrDoc reports a potential RR bug. Even if ErrDoc finds an RR bug, it may happen the corresponding resource is freed later in a different function, especially for resources that may exist beyond f_{caller}'s scope. For such resources, ErrDoc simply reports warnings.

For example, in Listing 1, corresponding to `OPENSSL_malloc` function at line 8, there was no `OPENSSL_free` function along the

error path containing `EVP_Digest1` (line 22). ErrDoc reports this case as RR bug.

3.4 Phase-III. Fix Error Handling Bugs

In this step, ErrDoc automatically fixes four types of error handling bugs. Let's assume, function f_{caller} did not handle errors returned by calling function f; ErrDoc detected and reported that error. To fix the bug, ErrDoc takes three inputs: the buggy program locations, *i.e.*, source code line numbers, the bug category, and any error handling code that may pre-exist in f_{caller}. First, ErrDoc traverses f_{caller}'s AST (Abstract Syntax Tree) and locates the AST nodes and its associated subtree that match the buggy program locations. Next, depending on the bug category, ErrDoc constructs an AST subtree to fix the bug, as described below. Finally, ErrDoc applies the fix to the original buggy AST by deleting and adding existing AST nodes and edges. Figure 3 represents the fixing process. **Red** and **green** subtrees show deletion and addition operations respectively.

We observed in Section 2, a human written patch to fix error handling bug is often integrated with a pre-existing error handling code block present in the buggy function body. Thus, our main challenge is to produce a bug-fix patch that blends well with its context to resemble a human written patch. This is important as a patch that does not follow the developers' coding practice may not get accepted [11].

ErrDoc looks for an existing error handling code that is already present in f_{caller}. If multiple such error handling code is found, ErrDoc chooses the one closest to the buggy location, in terms of source code line number. We measure the distance between the buggy program locations and an error handling code using three heuristics: (i) If an error handling code block or the buggy locations span over multiple source code lines, we choose the first line(s) as their respective starting points. The distance between the two is then measured based on their starting points. (ii) If multiple error handling code blocks exist in the context, we choose the one having minimum distance *w.r.t.* the bug location. (iii) If the same target function (f) is called multiple times from the same caller (f_{caller}), and at least in one call site the error is handled correctly, we follow its way to fix the bugs at other call sites within f_{caller}. Here, even if other error handling code exist for different functions at nearer context, we give priority to the same callee. If an error handling code block is found following either (ii) or (iii), we call such pre-existed block as *contextual error handling code*.

Following this above generic strategy, the rest of the section elaborates our big-fixing process for each bug category.

Incorrect/Missing Error Checks (EC): Given an EC bug, we know the function f's call site and the specific return value that is not checked correctly from EC bug detection step. To fix the bug, ErrDoc first adds the missing check (*i.e.*`if statement`) on the return value of f after its call site, as shown in 3(a). Then, if there is no contextual error handling code, ErrDoc constructs an error handling code block that consists of logging function call and a return statement with $Global_{err}$ value. Otherwise, ErrDoc refers to the contextual error handling code. For example, if the contextual code jumps to an error label using a `goto statement`, ErrDoc also follows the same.

For example, in Listing 1, ErrDoc reports line 15 as missing EC bug. To fix the EC bug, ErrDoc inserts an `if statement` at line 16 on the return value of `EVP_Digest`. In addition, ErrDoc leverages

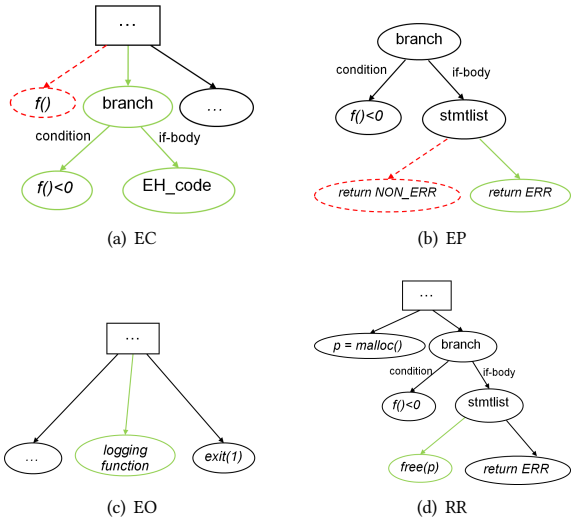

Figure 3: Bug Fixing Examples

the contextual error handling code (line 22 to 28) and constructs the error handling code, as shown in line 16 to 19. Note that, these two error handling code blocks follow the same patterns.

Incorrect/Missing Error Propagation (EP): As discussed in the previous sections, an EP bug returns an incorrect value from f_{caller} along an error path $Path_{err}(f_{caller}, f)$. To fix the bug, ErrDoc considers following three scenarios:

(1) *Outside f's error handling code.* If the buggy return statement returns a non-error constant, ErrDoc simply replaces it with an error-specific constant value as per $Global_{err}$. Otherwise, if it returns a variable, say v, ErrDoc constructs an assignment statement: $v = Global_{err}$. ErrDoc then inserts the assignment statement at the beginning of f's error handling code.
(2) *Inside f's error handling code with a contextual error handling block.* ErrDoc replaces the buggy return statement with the return statement contained in contextual error handling block.
(3) *Inside f's error handling code without contextual error handling block.* ErrDoc updates the return value of the buggy return statement with $Global_{err}$ value.

For example, in Listing 1, ErrDoc marks the return statement at line 27 as a potential source EP bug. The buggy statement is within `EVP_Digest1`'s error handling code (line 22 to 29). To fix this EP bug, ErrDoc replaces the buggy return statement with the return statement from contextual error handling block (line 17 to 18).

Incorrect/Missing Error Outputs (EO): Since EO bug does not call a logging function before exit along an error path, to fix the bug, ErrDoc inserts an extra logging function call statement before exit, as shown in Figure 3(c). For example, in Listing 1, before exiting at line 12 on `OPENSSL_malloc` failure, nothing was logged in the buggy version. ErrDoc diagnoses such case as EO bug and inserts an extra logging function call statement, as per input logger utility functions, as shown in line 11.

Incorrect/Missing Resource Release (RR): Since in an RR bug, a resource is not released correctly along an error path, *i.e.* resource allocation function is called but not the corresponding release function, ErrDoc first retrieves the function pair signature as identified in Section 3.2 and leveraging the signature constructs the

corresponding paired function call statement. Then, ErrDoc inserts the call statement before the return statement at the end of the error path. For example, in Listing 1, ErrDoc reports RR bug along the error path of `EVP_Digest1` (see line 23), since `OPENSSL_free` call was missing corresponding to `OPENSSL_malloc` function (line 8). Based on the signature of these two pair functions, ErrDoc constructs a corresponding function call `OPENSSL_free(str)` and inserts this function call statement before the return statement at line 28.

4 IMPLEMENTATION

ErrDoc is composed of three phases, as shown in Figure 2. The first two phases include error path exploration (Phase-I) and bug detection (Phase-II). They are implemented using Clang static analysis framework [1] (adapted from previous tool EPEx [14]) and Clang LibTooling [2]. The last part includes AST based bug fixing (Phase-III), which is implemented using Clang LibTooling [2]. In total, the implementation of ErrDoc includes almost seven thousand C++ LOC and three hundred Python LOC.

In particular, the implementation uses three Clang analysis steps: i. **Symbolic Execution:** ErrDoc uses Clang's static analysis framework to perform an under constrained symbolic execution for identifying error paths (Step I-a) and function pairs (Step I-c), and detecting and categorizing the bugs (Phase-II). We implement three Clang checkers for Phase-I and Phase-II. ii. **Data-dependency analysis:** Clang's inbuild static analyzer does not support data-dependency analysis. Hence, inspired by Zaks *et al.* [35], we leverage the symbols (persistent representation of opaque values) in Clang analyzer to implement a checker that can support simple data dependency checking to identify the function pair signatures (see Section 3.2). iii. **AST analysis:** We leverage Clang's AST analysis framework to identify error handling code (Step I-b) and related EC bug (Step II-a'), refining RR bug reports (Step II-d') and fixing the buggy programs (Phase-III). We use Clang LibTooling [2] and Clang LibASTMatchers [3] for the AST analysis.

The overall running effort of ErrDoc is similar to a clang static checker. Given a list of inputs (see Section 3.1), a python script is used to automate the whole process.

5 EXPERIMENTAL RESULTS

In this section, we evaluate ErrDoc's ability of detecting, categorizing, and fixing error handling bugs. In particular, we investigate the following research questions:

- **RQ1.** How accurately ErrDoc detects error handling bugs?
- **RQ2.** How accurately ErrDoc categorizes error handling bugs?
- **RQ3.** How accurately ErrDoc fixes error handling bugs?

Study Subject. We select five projects: OpenSSL, GnuTLS, WolfSSL, Curl, and Httpd for the evaluation. These projects were also studied by EPEx. We choose the same projects for comparison. We also create three types of dataset:

i. *Manually Injected Bugs.* We randomly choose source files from each project and inject 64 error handling bugs. To insert EC bugs, we remove existing error handling code that handles an API call. We inject EP bugs by replacing a return ERROR statement with a return NON_ERROR statement. EO bugs are injected by removing an existing logging function before exit function call along an error path. Finally, to inject RR bug, we remove a deallocation routine from an error handling code. The corresponding original code are considered as correct code to evaluate bug-fix performance. We also choose 22 instances of non-buggy cases that are representative of correct error handling code, as per our observation.

ii. *Real bugs from evolutionary data.* We further select 50 real error handling bugs and patches from the error handling bug-fix commits that we identified in the empirical study (see Section 2).

iii. *New bugs in OpenSSL.* ErrDoc also finds some new bugs. We manually verify their correctness to evaluate the tool. We are also in the process of reporting these bugs. Three of our patches are already integrated with OpenSSL code.

Study Methodology. We measure ErrDoc's capability to detect and categorize error handling bugs in terms of precision and recall. For each bug type t, suppose that E is the number of bugs detected by ErrDoc and A is the the number of true bugs in ground truth set. Then the precision and recall of ErrDoc in categorizing error handling bugs are $\frac{|A \cap E|}{|E|}$ and $\frac{|A \cap E|}{|A|}$ respectively. We use similar measurement for bug detection and fixing.

5.1 Study Results

RQ1. How accurately ErrDoc detects error handling bugs?

We evaluate ErrDoc in three settings, as described in Section 5. We further compare ErrDoc's accuracy with state of the art error handling bug detection tool EPEx [14]. Table 4 summarizes the result.

Table 4: Bug detection accuracy

	manually created dataset Total = 86 bug=64, non-bug=22		**evolutionary dataset** Total = 50 bug=50		**new bugs** Total = 105 bug=105	
	ErrDoc	**EPEx**	**ErrDoc**	**EPEx**	**ErrDoc**	**EPEx**
Detected	63	41	48	26	125	53
False Positive	2	1	0	0	20	11
False Negative	3	24	2	24	N/A	Over 63
Precision	0.97	0.98	1.00	1.00	0.84	0.79
Recall	0.95	0.63	0.96	0.52	N/A	N/A

In the manually curated dataset, ErrDoc reports 63 bugs, out of that 2 are false positives. To verify ErrDoc should not incorrectly classify a non-bug as bug, we check ErrDoc's accuracy on 22 correct error handling instances; out of them ErrDoc mistakenly identifies 2 as error handling bugs. Thus, ErrDoc's overall precision and recall are 97% and 95% respectively. In contrast, on the same dataset, we find EPEx detects error handling bugs with 98% precision and 63% recall. In the evolutionary dataset, ErrDoc detects error handling bugs with 100% precision and 96% recall, while EPEx's precision and recall are 100% and 52% respectively. Finally, we manually verify all the new bugs that both the tools have detected. Here, ErrDoc's precision is 84% while EPEx's precision is 79%. At any given point, it is not possible for us to know how many new bugs we cannot find; Therefore, we cannot detect recall in this case. However, ErrDoc can detect all the bugs that EPEx detects, but 63 bugs detected by ErrDoc cannot be found by EPEx . Thus, we conclude that ErrDoc can successfully find error handling bugs with an overall improvement of 5 to 0 percentage points in precision and 44 to 32 percentage points in recall when compared with previous tool EPEx.

We notice that a major improvement on ErrDoc's accuracy over EPEx is due to the former's capability of detecting Incorrect/Missing Resource Release (RR) bugs. Incorrect/Missing Resource Release bug detection primarily depends on the methodology to finding function pairs, as discussed in Section 3.2. Thus, to get an estimate of RR bug detection capability, we also measure ErrDoc's accuracy of detecting function pairs. In particular, we investigate:

RQ1-A. How accurately ErrDoc detects function pairs?

For this experiment, we create a ground truth set of 33 real function pairs from OpenSSL, by carefully studying the functions' specifications. As discussed in Section 3.2, we perform the function pair analysis along error paths because error paths being shorter than non-error paths will reduce the search space. The target function (f) further provides an automatic divider between resource allocation and deallocation functions. We also perform a data-dependency analysis between the arguments and return values to the function pairs. To check how effective these strategies are, we compare ErrDoc's accuracy of detecting function pairs at different settings, as shown in Table 5.

In particular, we compare the function pairs identified by ErrDoc against the ground truth set. The ErrDoc's identified function pairs that are also in the ground truth are marked as true positives. We further manually check the remaining function pairs detected by ErrDoc against OpenSSL specification to decide if they are true or false positives. To measure the false negatives, we count how many function pairs in ground truth are not identified by ErrDoc. Table 5 shows the result.

Table 5: Accuracy of Function Pair detection at various settings

	Top 1		Top 2		Top 3	
	Precision	Recall	Precision	Recall	Precision	Recall
1. **error path + f**	0.45	0.30	0.34	0.36	0.27	0.36
2. **all path + f**	0.31	0.30	0.21	0.39	0.15	0.39
3. **error path + not-f**	0.23	0.30	0.17	0.36	0.13	0.36
4. **all path + not-f**	0.13	0.27	0.12	0.39	0.08	0.39
5. **error path + f + data-dep**	1.00	0.30	0.92	0.36	0.86	0.36
6. **all path + f + data-dep**	0.71	0.30	0.50	0.39	0.39	0.39

Function pairs are computed along only error paths (settings 1,3,5) and all the paths (settings 2,4,6). In all the settings except 3 and 4, a target function, say f, is used as partitioning function. Also, settings 5 and 6 consider data-dependency analysis among the arguments of function pairs. Top 1, Top 2, and Top 3 are top pairs selected for a given function.

The result shows that identifying function pairs using only error paths (settings 1, 3, and 5) as opposed to all the paths (settings 2, 4, and 6) gives better precision and recall in all the settings. These results prove our intuition that only considering error paths reduce search space and thus introduce less noise in function pair analysis. Hence, we see better accuracy. In addition, with a partitioning function, say f, we see better precision (setting 1 vs. 3 and 2 vs. 4). For example, comparing settings 1 and 3, we see f helps in gaining precision by 22 percentage point while recall remains almost the same. These results indicate that f plays an important role in function pairing and also ordering the paired elements. Finally, we compare the impact of data dependency analysis (compare setting 1 vs. 5, 2 vs. 6). For instance, setting 1 vs. 5 helps in 55 percentage point gain in precision at Top 1 while the recall remains the same. Thus, we conclude that setting 5 (*i.e.* error path + f + data-dependency) gives the highest accuracy in detecting function pairs. Note that, the precision is highest at Top 1 setting, while the recall is lower than top 2 and 3 as we disregard some valid pairs (*e.g.,* OpenSSL two-to-one or one-to-two function pairs). Since three-to-one or one-to-three function pairs are rare, recalls at setting top 2 and top 3 are almost the same.

RQ2. How accurately ErrDoc categorizes error handling bugs?

Table 6: Evaluation for Bug Categorization

		EC	EP	EO	RR
manually created dataset	**precision**	1	0.91	1	0.95
	recall	0.95	1	1	0.91
evolutionary dataset	**precision**	1	1	-	1
	recall	0.96	1	-	0.95
new bugs	**precision**	0.90	1	0.83	0.88

We evaluate ErrDoc's categorization accuracy for all the four bug types in three settings, as shown in Table 6. Overall, ErrDoc categorizes the bugs with 83% to 96% precision and above 90% recall.

We find that the false positives mainly fall into three categories. If the caller does not follow the global error/non-error values, ErrDoc may falsely detect them as EP, EC or RR bugs. Secondly, the false positive may occur due to the limitation of Clang's static analyzer. For example, one correct code block is incorrectly detected as EP bug because the caller of this correct code block does not follow the global non-error value. False positives are also caused if ErrDoc misses to identify valid function pairs. For example, a correct code block is incorrectly detected as RR bug because ErrDoc misses alternative function pairs in case of one-to-multiple pair. In categorizing EO bugs, if a different logging function is used than expected false positives may occur.

Some false negatives occur as there are some EC and RR bugs that cannot be detected by ErrDoc. For undetected EC bug, the caller of this EC bug does not follow the global non-error value. However, this is not fundamental limitation of ErrDoc's algorithm; For the RR bugs that ErrDoc fails to detect, the function pairs do not have any data dependency (*e.g.,*`start` and `end`).

RQ3. How accurately ErrDoc fixes error handling bugs?

We evaluate the correctness of our bug fix algorithms using both manual and evolutionary bug data. For both the dataset, we already know the corresponding fix patches. Thus, to evaluate ErrDoc's performance, we check whether ErrDoc's produced patches exactly match with the original fix; If not, we check whether they are semantically equivalent. Table 7 summarizes the result across different projects. In the manually created data, 68% patches generated by ErrDoc are exactly similar to the original correct code and 16% patches generated by ErrDoc are semantically equivalent; thus giving an overall precision of 84%. However, there are 16% patches that are not matched with the original correct code. Most of the mismatch arises because of the different logging messages between the patches generated by ErrDoc and the original code.

In the second part, we apply ErrDoc to fix 50 selected real bugs retrieved from evolutionary history. As shown in Table 7, ErrDoc can fix 72% real bugs in a similar way the developers have fixed them. But for 28% bugs, our patches cannot be matched with the patches proposed by developers. Although all our produced bugs are correct, *i.e.* fixing the bugs correctly, a mismatch typically results from one of the three issues: first, developers sometimes use different logging

Table 7: Evaluation for Bug Fix

Project	Bug Type	Manually Injected Bug				Real Bug			
		Total bugs	Exact match	Semantic match	No match	Total bugs	Exact match	Semantic match	No match
Summary	EC	20	11	2	7	23	9	6	8
	EP	10	9	1	0	5	2	0	3
	RR	20	14	5	1	22	10	9	3
	Total	50	34 (68%)	8 (16%)	8 (16%)	50	21 (42%)	15 (30%)	14 (28%)

messages. Secondly, for some EP bugs, developers return a specific meaningful error value instead of a global error returned by ErrDoc. Finally, for some RR bugs, developers put all the deallocation routine calls at the end of the caller function, and insert goto statement to the error handling code so that all the resources can be deallocated together. ErrDoc fails to produce such patch if no similar instances can be found in the context.

When comparing ErrDoc's produced patches with the patches by developers, we find in some cases, the closest error handling code identified by ErrDoc is a false positive instead of the real closest error handling code. In these cases, ErrDoc cannot fix EC bugs correctly, even though the real closest error handling code can work.

6 THREATS TO VALIDITY

Some inaccuracy in RR bug detection may arise due to the inaccuracies in function pair identification. EC and EP bug detection may have false positives when a caller does not follow their global specifications. Further, imprecise error specification and the limitation from symbolic execution may introduce inaccuracies in identifying error paths and function pairs. Also, our way of identifying error handling code can affect the accuracy of EC bugs fix. However, our evaluation shows that these cases are rare.

We evaluate ErrDoc on only five projects. Thus our results may not generalize. To minimize this threat we evaluate the results by creating different types of dataset across all the projects.

7 RELATED WORK

(i) Automatically detecting error handling bugs. Researchers proposed several techniques to automatically detect some of these bugs. Lawall *et al.* used program matching and transformation to detect EC bugs in OpenSSL library [18]. Gunawi and Rubio-Gonzalez found both EC and EP bugs in file and storage system using static dataflow analysis [13, 30]. Jana *et al.* [14, 16] used symbolic execution to detect error handling paths and then detected EC and EP bugs by checking whether appropriate error values are propagated upstream along the error paths. ErrDoc extends Jana *et al.*'s work.

Saha *et al.* [31] proposed Hector to detect RR bugs in error handling code. They used an intra-procedural live-variable analysis to collect a set of exemplars of correct resource allocation and release. Then, they compare each candidate fault with close exemplars to determine if some resources need to be released. Hector has a 23% false positive rate. ErrDoc further reduces the false positive rate by analyzing the data dependency of the resource related variables along restricted error paths, which span over the callee and caller functions.

Further, there are tools to detect and analyze exception handling bugs in Java [28, 29, 32, 33]. Since Java exception handling mechanism differs significantly from C, we cannot directly use these tools, our study is complementary to them.

(ii) Automated program repair. Automatic program repair is a well-researched field, and previous researchers proposed many generic techniques for general software bugs repair [15, 17, 19, 21, 22]. However, they did not consider the unique characteristics of error handling bugs (as discussed in Section 2), thus they cannot repair error handling bugs efficiently. For example, Meng *et al.* [23, 24] proposed program repair tools that learn the program patches from examples and applied the patches in a context-aware fashion. ErrDoc also uses the identified error handling code to fix EC bugs; However, users do not have to provide any explicit examples. Lawall *et al.* [18] can detect and fix EC bugs in OpenSSL library. Since they made assumptions on the return error values, their tool suffers from a high false positive rate around 50%. In contrast, ErrDoc is designed to detect and fix different error handling bugs for general C programs with a high accuracy.

Further, the previous code repair tools mainly rely on test cases demonstrating buggy behaviors. However, it is often hard to write test cases showing erroneous behavior as error conditions rarely occur in a regular program execution. Also, the generated patches produced by the previous tools, even though functionally correct, often do not blend in with the surrounding context. ErrDoc addresses both of these issues for error handling bugs by leveraging heuristics specific to C error handling code.

(iii) Specification mining. There is a large volume of generic techniques to automatically detect bugs based on the specifications. Kang *et al.* [16] proposed APEx that automatically inferred error specification of C APIs and detected error handling bugs with 67.4% precision overall. Engler *et al.* [12] and Li *et al.* [20] used different static analysis techniques to extract rules from source code and then cross-check for contradictions. Wu *et al.* [34] proposed RRFinder to automatically mine resource-releasing specifications for Java API libraries. Our work complements these generic techniques and particularly focuses on error handling behavior of C code.

8 CONCLUSION

Error handling bugs are common and can have a significant effect on software security and reliability. In this paper, we first present a comprehensive study of real world error handling bugs and create a taxonomy of them. Next, we design, implement and evaluate ErrDoc that automatically detects, diagnoses, and fixes error handling bugs. Evaluating ErrDoc on five open-source code bases reveals that ErrDoc can detect error handling bugs with high accuracy—it outperforms the precision of existing tool EPEx up to 5 percentage points and recall up to 44 percentage points. ErrDoc can further categorize and fix the bugs with high precision and recall.

Not all the error handling bugs have same consequences. In future, we plan to examine the severity of these bugs and investigate heuristics to automatically rank these bugs based on their potential severity. We will also extend this approach to analyze error and exception handling behavior for programs written in other programming languages and framework.

ACKNOWLEDGEMENTS

This work is sponsored by the National Science Foundation (NSF) grant grant CNS-16-18771 and CNS-16-17670. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

REFERENCES

[1] 2011. Checker developer manual. http://clang-analyzer.llvm.org/checker_dev_manual.html. (2011).

[2] 2012. Clang LibTooling. http://clang.llvm.org/docs/LibTooling.html. (2012).

[3] 2013. Clang LibASTMatchers. http://clang.llvm.org/docs/LibASTMatchersReference.html. (2013).

[4] 2014. CVE-2014-0092. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092. (2014).

[5] 2015. CVE-2015-0208. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0208. (2015).

[6] 2015. CVE-2015-0285. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0285. (2015).

[7] 2015. CVE-2015-0288. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0288. (2015).

[8] 2015. CVE-2015-0292. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0292. (2015).

[9] 2017. CVE-2017-3318. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3318. (2017).

[10] 2017. CVE-2017-5350. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5350. (2017).

[11] Miltos Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE'14)*.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 57–72.

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

[14] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs using Error Specifications. In *USENIX Security Symposium (USENIX Security)*.

[15] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 266–276.

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

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.

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

[19] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 3–13.

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

[21] Peng Liu, Omer Tripp, and Xiangyu Zhang. 2014. Flint: fixing linearizability violations. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 543–560.

[22] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 133–146.

[23] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Sydit: creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 440–443.

[24] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices* 46, 6 (2011), 329–342.

[25] Brian A Nejmeh. 1988. NPATH: a measure of execution path complexity and its applications. *Commun. ACM* 31, 2 (1988), 188–200.

[26] owasp 2007. OWASP TOP 10. https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf. (2007).

[27] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 504–515.

[28] M.P. Robillard and G.C. Murphy. 1999. Analyzing exception flow in Java programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.

[29] Martin P Robillard and Gail C Murphy. 2000. Designing robust Java programs with exceptions. In *ACM SIGSOFT Software Engineering Notes*, Vol. 25. ACM, 2–10.

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

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

[32] W. Weimer and G.C. Necula. 2004. Finding and preventing run-time error handling mistakes. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[33] W. Weimer and G.C. Necula. 2008. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2008).

[34] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2011. Iterative mining of resource-releasing specifications. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 233–242.

[35] Anna Zaks and Jordan Rose. 2012. How to Write a Checker in 24 Hours. (2012). http://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf