APEx: Automated Inference of Error Specifications for C APIs

Yuan Kang
Columbia University, USA
yuanjk@cs.columbia.edu

Baishakhi Ray
University of Virginia, USA
rayb@virginia.edu

Suman Jana
Columbia University, USA
suman@cs.columbia.edu

ABSTRACT

Although correct error handling is crucial to software robustness and security, developers often inadvertently introduce bugs in error handling code. Moreover, such bugs are hard to detect using existing bug-finding tools without correct error specifications. Creating error specifications manually is tedious and error-prone. In this paper, we present a new technique that automatically infers error specifications of API functions based on their usage patterns in C programs. Our key insight is that error-handling code tend to have fewer branching points and program statements than the code implementing regular functionality. Our scheme leverages this property to automatically identify error handling code at API call sites and infer the corresponding error constraints. We then use the error constraints from multiple call sites for robust inference of API error specifications. We evaluated our technique on 217 API functions from 6 different libraries across 28 projects written in C and found that it can identify error-handling paths with an average precision of 94% and recall of 66%. We also found that our technique can infer correct API error specifications with an average precision of 77% and recall of 47%. To further demonstrate the usefulness of the inferred error specifications, we used them to find 118 previously unknown potential bugs (including several security flaws that are currently being fixed by the corresponding developers) in the 28 tested projects.

CCS Concepts

•Software and its engineering → Software libraries and repositories;

Keywords

error handling bugs; specification mining; API errors

1. INTRODUCTION

Reliable software should be able to tolerate a wide variety of failures. Therefore, reliable software must be designed to behave gracefully even when lower-level functions (*e.g.,* API functions) fail [21], otherwise the failure may affect both software availability and security. Typically, when a failure occurs, the failing function notifies its caller about the failure and the caller then decides how to handle such failures in a graceful manner. However, software developers often make mistakes and inadvertently introduce bugs in API error handling code [32, 33]. For example, a significant number of security flaws result from incorrect error handling (*e.g.,* CVE-2014-0092 [2], CVE-2015-0208 [3], CVE-2015-0285 [4], CVE-2015-0288 [5], and CVE-2015-0292 [6]). In fact, incorrect error handling was listed by the Open Web Application Security Project (OWASP) as one of the top ten sources of security vulnerabilities [28].

There are several existing static and dynamic analysis techniques to automatically detect incorrect handling of API function failures. For example, several prior projects used dynamic fault injection to simulate function failures to check whether the test program handles such failures correctly [11, 35, 23]. Another popular approach for detecting error handling bugs is to use static analysis to check whether failures are properly detected and handled [33, 17, 20, 19]. For example, EPEx, a system designed by Jana *et al.* [19], uses under-constrained symbolic execution to explore error paths and leverages a program-independent error oracle to detect error handling bugs. However, all such tools need the error specifications of the API functions as input. The error specification of an API function indicates whether the function can fail or not, and if so, how the function communicates a failure to the caller.

Manually generating error specifications is error-prone and tedious. It is particularly hard for low-level languages such as C that do not provide any specialized exception handling mechanisms and contain only generic channels of communication (*e.g.,* return value, arguments passed by reference) between the caller and callee functions. Therefore, the developers of each API must decide individually which values indicate failure. Consequently, the error specifications are function-specific and may vary widely across functions. While there are values that intuitively indicate errors, such as negative integers or NULL pointers, such conventions do not always hold and thus will result in highly inaccurate specifications. For example, in OpenSSL, for some API functions 0 indicates errors, while for others 0 is used to indicate success.

Moreover, some API functions are infallible, *i.e.* they never return any error value. A correct error specification should only contain fallible API functions but it is hard to manually separate them from the infallible ones. API documentation is one obvious source for gathering such specifications. However, previous research has shown that API documentation is often unavailable or inaccurate [7, 34, 41]. For example, Rubio-González *et al.* found 1,700 undocumented error-code instances in Linux file system API functions [34]. Further, API documentation usually fails to distinguish between benign and critical errors which is essential to minimize spurious bug reports as benign errors can be safely ignored.

In this paper, we present a novel technique to automatically infer the error specifications for C API functions by analyzing their usage across multiple applications. The key insight behind our technique is that when an API function call fails and returns an error, usually its caller does not continue normal execution and returns immediately. Error handling code written in C is known to be significantly more likely to use "goto" statements than regular code for such control transfers [25]. Therefore, error handling code in the caller function is more likely to have less branching points, program statements, and function calls than the code implementing a normal functionality. We leverage such path characteristics to identify error-handling code. In particular, we use under-constrained static symbolic execution at the caller function

to explore all feasible paths and determine whether a path exercises error handling code or not using on the path's features. Next, we extract the constraints on an API function's return value that must be satisfied along the paths exercising error handling for that API function. We call these constraints *potential error specifications*. However, these constraints may not be accurate due to either buggy or over-conservative error handling checks. We minimize these issues by comparing and contrasting the error constraints for the same API function extracted from different call sites across different programs. We pick the most common error constraints as the error specification of the API function. If there is no clear popular error constraint(s) across the call sites (either the constraints are different or there are no constraints), we declare that the API function cannot fail, or the developers do not consider the failure of the function to be critical.

Using our approach, we generated 93 correct error specifications for API functions collected from 6 popular C libraries, with a precision of 77% and a recall of 47%. We also discovered 118 potential error handling bugs that violate the error specifications across 28 different applications. So far we have reported 17 bugs, out of which 4 have been fixed.

In summary, the main contributions of our work are as follows.

- To our knowledge, we are the first to notice that the paths performing error handling in C code have unique features (*e.g.,* branching points, program statements) that distinguish them from non-error paths and empirically confirmed this insight.
- Leveraging characteristics of the program paths performing error handling, we design and implement a novel algorithm, APEx, to automatically infer error specifications of C API functions from their usage.
- We conduct a detailed empirical evaluation of APEx's ability to automatically infer error specifications for 217 C API functions from 6 open-source libraries. We found that APEx has a precision of 77% and a recall of 47%. This improves significantly over trivial specifications like treating a non-negative return value as success for OpenSSL API functions, which only has 12% precision.
- Using these error specifications, we discovered 118 potential error-handling bugs in 28 real-world applications. So far we have reported 17 bugs that we thought to have serious consequences and 4 out of them are already fixed by the developers.

The rest of the paper is organized as follows. Section 2 describes the main insight behind our approach using a running example. Next, in Section 3, we present an empirical study of unique path characteristics of error handling code. Section 4 discusses APEx's methodology for inferring error specifications and Section 6 evaluates it. Section 7 discusses related work. Finally, in Section 8, we examine potential threats that may affect our findings and we conclude in Section 9.

2. MOTIVATING EXAMPLE

In this section, we provide a high-level overview of our approach with a running example adapted from the Curl project (see Figure 1).

In this sample code fragment, the caller function `hugehelp` calls four different API functions: `inflateInit2`, `inflate`, `inflateEnd` from `zlib` and `malloc` from `libc`. `inflateInit2` returns a non-zero integer value on error[1] (line 4). The caller function checks for such cases and immediately returns (line 5) in case of an error. If no such error occurs, the execution continues and calls `malloc` at line 6. As malloc returns a `NULL` pointer in case of a failure, line 7 checks whether malloc was able to allocate memory correctly; if not, it goes directly to line 17 and subsequently returns from the caller function. Otherwise, the regular execution continues. Another similar error-check for `inflate` API call takes place at line 10. Note that `inFlateEnd` can also return an error but the developer chose to ignore the error. All the branches corresponding to error values are shown in red, dotted lines in Figure 1. The error specifications for the API functions `inflateInit2`, `inflate`, and `malloc`, based on their usage, can be expressed as $\neq 0$, `NULL`, and $\neq 0$ respectively. The goal of this work is to automatically infer such API error specifications from these usage patterns.

```
1  /* File:tool_hugehelp.c */
2  void hugehelp(...)
3  {
4    if (inflateInit2(...) != 0)
5      return;
6    buf = malloc(...);
7    if (buf != NULL) {
8      while(1) {
9        status = inflate(...);
10       if (status == 0) {
11         ...
12       }
13       else
14         break;
15     }
16   }
17   inflateEnd(...);
18 }
```

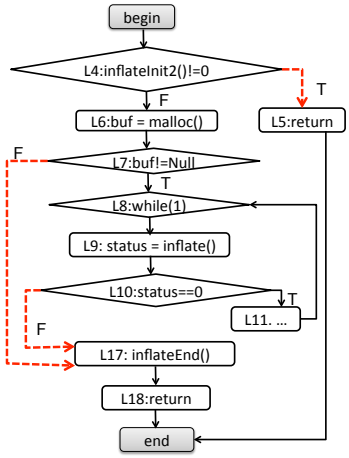


Figure 1: Sample error handling code and the corresponding Control Flow Graph (CFG) containing zlib and libc API functions adapted from Curl. The red, dotted arrows show error branches.

Before we describe our approach, let us introduce some key terms that we will use in the rest of the paper.

DEFINITION 2.1. ***Control Flow Graph (CFG):*** *A CFG of a function in the program is a directed-graph represented by a tuple* $\langle N, E \rangle$. *N is the set of nodes, where each node is labeled with a unique program statement. The edges,* $E \subseteq N \times N$, *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. [29]*

DEFINITION 2.2. ***Path :*** *A path P is a sequence of nodes* $\langle n_0, n_1,....n_j \rangle$ *in a CFG, such that there exists an edge between* n_k *and* n_{k+1}, i.e. $(n_k,n_{k+1}) \in E$, *for k = 0,...,j-1 [26].*

Figure 1 shows the CFG corresponding to the motivating example. The begin and end nodes indicate the start and return/exit from the function. Every other node corresponds to one program statement as marked with the respective line number. A path exists between two nodes, if they are connected by edges. For example, as shown in the Figure, three paths exist between node L7 and L17. Among all the possible paths that exist in a CFG, we define error and non-error paths corresponding to an API function as follows:

[1]zlib error specification:http://lxr.free-electrons.com/source/include/linux/zlib.h

DEFINITION 2.3. ***Error Path:*** *For an API function f, an error path $P_{err}(f)$ is a path in the CFG of the caller function that starts from a node containing a function call to f, follows a branch along which f's error conditions are satisfied, and ends at n_{end}.*

DEFINITION 2.4. ***Non-Error Path:*** *For an API function f, a non-error path $P_{nerr}(f)$ is a path in the CFG of the caller function that starts from a node containing a function call to f, follows a branch along which f's error conditions are not satisfied, and ends at n_{end}.*

For example, in Figure 1, ⟨L4, L5, end⟩, ⟨L6, L7, L17, L18, end⟩, and ⟨L9, L10, L17, L18, end⟩ are error paths for API functions `inflateInit2`, `malloc`, and `inflate` respectively. A CFG can have multiple error and non-error paths for an API function depending on how many error/non-error values the function can return and other conditions that are checked along those paths.

DEFINITION 2.5. ***Fallible/Infallible API Functions:*** *A fallible API function is a function that can fail due to either internal or external reasons and, in case of such a failure, notifies its caller function by setting the API function's return value to certain error value(s). The API functions that cannot fail are called infallible API functions.*

Notice that the numbers of statements, function calls and branching points along the error paths of a fallible API function are lower than their corresponding non-error paths. For example, the error path for `inflateInit2` (line 4) contains only 2 statements, 1 function call, and 1 branching points. In contrast, a non-error path ⟨L4, L6, L7, L8, L9, L10, L17, L18, end⟩ has 8 statements, 4 function calls, and 4 branching points (one while and three if statements). We leverage this insight to design, implement, and evaluate an algorithm to automatically infer error specifications of the API functions.

For an API function f, our algorithm works in three steps as described below.

Step I: Collecting path information from individual call sites. From the CFG of a caller function f, we collect all the paths that start from the node corresponding to f's call statement. We also collect the constraints on the return value of f and calculate the number of statements along each path. Since more paths can only be generated by more branches, we additionally count the number of paths that run the call site for each constraint.

Step II: Combining path information across call sites. We follow the above step for each caller of f across multiple call sites and combine the collected path information (*i.e.* constraints on the return value of f and other path features) based on the different unique constraints.

Step III: Inferring error specifications using voting. We mark the paths as potential error paths that have a lower number of paths satisfying the same return value constraint and statement counts across all the call sites (see Section 3 for a detailed analysis of the unique features of error paths). Thus, for API function `inflateInit2`, path ⟨ L4, L5, end ⟩ will be marked as an error path since it contains significantly less statements and branching points (and therefore, less paths with matching constraints) compared to the non-error paths (see Figure 1). The corresponding constraints will be inferred as error specifications. For instance, the error condition $\neq 0$ for API function `inflateInit2` will be inferred as its error specification.

3. EMPIRICAL STUDY OF ERROR PATHS

We conduct an empirical study of error paths to check whether they have any distinguishing features over non-error paths. Our intuition behind this comes from the observation that developers often check whether an error has occurred during the execution of an API function call before using any of its results; in most error cases, the caller function short-circuits the execution flow and bypasses the main logic. Thus, in the caller functions, the error paths should be simpler, with less logic involved compared to the corresponding error-free paths. We measure such path characteristics in the caller functions by three properties: (i) number of program statements; (ii) number of function calls contained in the path; and (iii) number of paths satisfying the same constraint on the return value of the called function, which approximates the relative number of branch points. We hypothesize that all these three measures should be lower along error paths than the corresponding error-free paths.

3.1 Study Method

Table 1: Study subjects for understanding error path characteristics

Library	#Project	#API functions	#call sites	#error paths	#non-error paths
GnuTLS	6	47	702	532	3840
zlib	9	15	134	249	2943

To understand the characteristics of error paths, first we need to identify the error and non-error paths at each API call site. To do that, we manually generated error specifications of 47 GnuTLS functions and 15 zlib functions that are called from 702 and 134 call sites respectively (see Table 1). Next, we performed under-constrained symbolic execution at the caller functions of each API function to explore the feasible paths. For the paths that invoke calls to the API functions, we checked the constraints on the return values of the functions against their corresponding error specifications. The paths along which only error values can be returned are marked as error paths, while the paths along which only non-error values can be returned are marked as non-error paths. We drop all other cases where we cannot definitely conclude whether they are error or non-error paths. Thus, in total we compared 532 and 249 error paths in GnuTLS and zlib and 3840 and 2943 non-error paths in them.

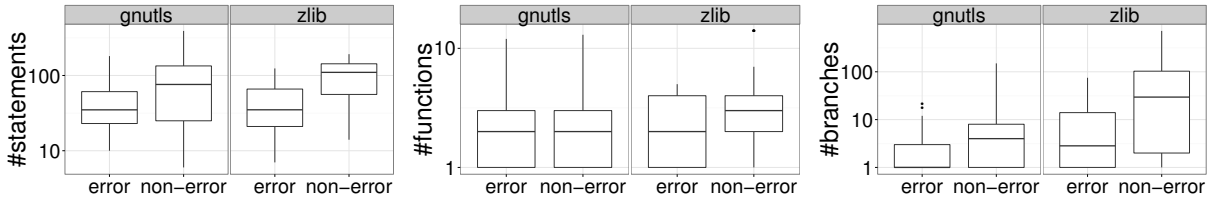
For example, in Figure 1, the error specification of the zlib API function `inflateInit2` is $\neq 0$. Our analysis will conclude that path ⟨ L4, L5, end ⟩ is an error path as the error condition is always satisfied along that path.

3.2 Characteristics of Error Paths

With the set of identified error and non-error paths, we check whether error paths have any distinguishing features over non-error paths. We characterize a program path based on three features: (i) number of statements executed in the path, (ii) number of functions called along it, and (iii) number of paths that satisfy the same constraint at the call site. Figure 2 summarizes the result.

Number of program statements. The box-plots in Figure 2a show that in both GnuTLS and zlib, the number of statements executed along error paths is less than non-error paths. A Wilcoxon non-parametric test shows that these differences are statistically significant (see Table 2). Also, Cohen's D effect sizes between the two are 1.121 and 0.697 in zlib and GnuTLS respectively, which are considered to be large and medium.

Number of function calls. Figure 2b shows that for zlib APIs, the number of function calls in error paths is slightly lower than that of non-error paths. A Wilcoxon non-parametric test in Table 2 also



(a) number of program statements executed **(b) number of functions called** **(c) number of paths**

Figure 2: Characteristics of error vs. non-error paths

Table 2: Comparison between error vs. non-error paths

features	Library	wilcox.p.value	cohendD
#statements	zlib	8.83E-61	1.121 (large)
	GnuTLS	1.87E-11	0.697 (medium)
#functions	zlib	1.91E-06	0.326 (small)
	GnuTLS	0.128760453	0.039 (negligible)
#paths	zlib	0.005256049	0.568 (medium)
	GnuTLS	8.33E-05	0.418 (small)

confirms this observation with statistical significance. However, Cohen's D effect size shows that the difference is small (0.326). In contrast, for GnuTLS APIs, we do not see any differences between the two sets. Thus, overall we may not be able to characterize error paths based on the number of functions.

Number of paths. According to the box-plots in Figure 2c, in both GnuTLS and zlib, the number of error paths are less than non-error paths with statistical significance (see Table 2). Cohen's D effect sizes between the two are 0.568 (medium) and 0.418 (small) for zlib and GnuTLS respectively. This shows that the number of paths can be a good feature for distinguishing error constraints.

Therefore, we conclude that the number of program statements are significantly lower in error paths than non-error paths, and that there are significantly less error paths.

4. METHODOLOGY

This section presents APEx, a tool that automatically infers error specifications for C APIs. For an API function f and its caller c, APEx uses under-constrained symbolic execution [30] to explore all the feasible paths in c that contain a call to f. For each path, APEx collects the constraints on the return value of f and other path specific statistics, *i.e.* number of paths satisfying the same constraint and program statements. APEx aggregates such information over multiple call sites of f. The paths containing fewer paths satisfying the same constraint and program statements than average are identified as potential error paths and corresponding constraints are marked as the potential error specification of f. Finally, the error specification of f is decided by a voting algorithm that picks the most popular constraints from potential error specifications across different programs. Figure 3 shows APEx's workflow. The algorithm works in three steps:

Step I. Collecting path information from individual call site

For an API function f and a given program p, first APEx identifies f's caller functions from p's call graph. Next, for a caller c, APEx uses under-constrained symbolic execution [30, 16] to explore all the feasible paths in c that contain a call to f and collects information about path features (PF) such as statements and number of paths. Note that APEx does not gather information about the number of function calls per explored path because function calls may not be a distinguishing feature for error paths, as shown in Section 3.

For path exploration, APEx uses under-constrained symbolic execution starting from c as opposed to whole-program symbolic execution, thus ignoring the costly path prefix from the main function to c [30]. This helps APEx remain scalable while analyzing large target programs at the cost of introducing some spurious paths. However, since we will combine data from multiple call sites in Step II, effects of such spurious paths will be minimized.

In particular, while exploring paths from caller c, APEx only includes paths that start at the API call, end when the caller function terminates and records statistics about program statements and path counts that directly appear inside the caller. This allows APEx to focus on the relevant part of the caller body where error paths differ from non-error paths. APEx further gathers the constraint (C) on the return value of f. By limiting the paths within caller functions, APEx also limits the constraints that APEx gathers for the return values of the API functions. To maximize the accuracy, APEx delays the evaluation of any unknown, non-constant symbols, until the symbol is no longer used, or the path has reached the end of the caller. Delaying evaluation allows APEx to consider all the places within the caller function that add constraints on the return value of the API function. The final output from this step represents a set of paths characterized by the constraints and path-features: $\{(C, PF)\}$.

Also, note that while our discussion and current implementation only supports constraints on return values, our technique can easily be extended to infer error specifications of an API function that transmits error values by modifying an argument passed by reference.

Step II. Combining path information across call sites

In this step, we aim to combine the path information (*i.e.* constraint and path feature) collected from multiple call sites by comparing constraints on the return values of an API function.

This is challenging because these constraints are often not identical. Some constraints are stronger than others. For example, consider the examples shown in Table 3 for the `libc` API function `stat`. The function returns 0 on success and -1 on error [2]. However, while checking for failure, call site A uses the constraint < 0, while call site B uses the constraint $\neq 0$. While both are correct, < 0 is a stronger constraint than $\neq 0$. While inferring the error specifications of an API call, APEx picks the one that is used often in the program, but the stronger constraint, *i.e.* < 0 in this case, will also include the counts of the more inclusive constraint, so that it

[2] http://linux.die.net/man/2/stat

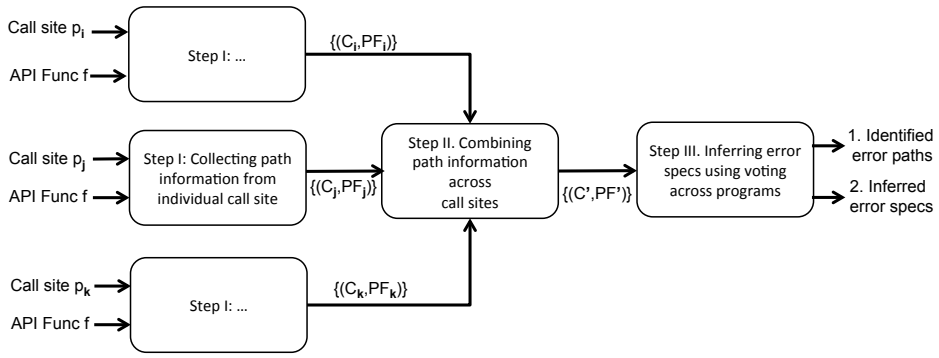


Figure 3: APEx's Workflow

Table 3: Sample code showing error checking of libc API function *stat*

Program A:

```
/* File: mutt/mbox.c */
void mbox_reset_atime (CONTEXT *ctx, struct stat *st)
{
  ...
  if (stat (..., ...) < 0)
    return;
  ...
}
```

Program B:

```
/* File: tar/gnu/chown.c */
int
rpl_chown (const char *file, uid_t uid, gid_t gid)
{
  ...
  if (stat (..., ...) != 0)
    return -1;
  ...
}
```

is more likely to be chosen. This is critical for the accuracy of our inferred error specifications. Notice that, such issues only appear in the case of integers, given that APEx currently supports three types of return values (bool, pointer, and integer). Booleans and pointers cannot have overlapping constraints as APEx currently does not distinguish between different values of non-NULL pointers.

While combining the path information across call sites of an API function, APEx splits any overlapping range into multiple, non-overlapping ranges. For example, a constraint $\neq 0$ will be represented by the inclusive ranges [MIN_INT, -1], [1, MAX_INT]. Let us assume that the return value for a function has the constraint $\neq 0$ (*i.e.* [MIN_INT, -1] and [1, MAX_INT]) at one call site and < 0 (*i.e.* [MIN_INT, -1]) at another call site along two paths; the first path has s program statements and the second path has s' program statements. To compute the aggregate path features across different constraints, APEx splits the first constraint into two constraints, each having 1 path and s program statements. The combined features of common constraint [MIN_INT, -1] from the two call sites will be a path count of 2 and a statement count list of (s, s') respectively.

The final output of this step will be a combined set of path information characterized by constraints and path-features: $\{(C', PF')\}$.

Step III. Inferring error specifications using voting

In this step, APEx aims to identify potential error constraints for a library function based on the path features. The main decisions are made in two stages. First, APEx identifies an error constraint from each project (see Algorithm 1). Then the individual programs vote for the global decision (see Algorithm 2).

Per-project error constraint inference. In this step, APEx tries to identify potential error constraints of an API function f for each project individually, based on the aggregated path features as discussed in Step-II (see Algorithm 1). In particular, among all the non-overlapping constraints over all the paths for f, APEx first tries to identify the constraints for which the total number of paths is at least one standard deviation lower than the average (line 9-12) of all the paths. If it fails to identify any such constraint, APEx then tries to find the constraints whose median statement counts over all paths in the project is at least one standard deviation lower than the average (line 13-16). If this too fails, the corresponding project refrains from global voting.

Global error constraint inference. To infer global error specification of an API function f, APEx seeks votes from all the projects that call the function (line 7 of Algorithm 2). However, if some callers have too few paths, APEx does not trust them. To ensure this, APEx considers votes from the projects where the number of paths starting with a call to f is at least above or equal to the lower quartile of all such paths across all the projects (line 6). Finally, APEx calculates the votes for potential error constraints from each project and picks the constraints which have at least one standard deviation more votes than the average (line 16). However, if the number of such error constraints exceeds MaxChoices, APEx declares the API function as infallible (line 13 to 15). Otherwise, APEx outputs the selected error constraints as the final error specification. The paths where the error specification are satisfied are marked as potential error paths.

5. IMPLEMENTATION

We gathered information about the path features using the Clang static analysis framework [1] (part of the LLVM project) and its underlying symbolic execution engine. We implemented the parsing and processing logic for error specification inference (*i.e.* Algorithms 2 and 1) from the path features using Python. Our addition to Clang consists of $1,180$ lines of C++ code, while the steps written in Python consist of $7,063$ lines, as measured using SLOCCount [40]. We give an overview of our modifications to Clang below.

The Clang analyzer symbolically tries to explore all feasible paths along the control flow graph of an input program. It provides a platform for custom, third-party *checkers* to monitor different paths and inspect the corresponding path constraints. A typical checker often looks for violations of different invariants along a path (*e.g.,* division by zero). In case of a violation, the checker reports bugs. We implement the path gathering process of APEx as a checker inside the Clang analyzer.

We extended Clang's symbolic execution engine to perform under-constrained symbolic execution in each caller function of the API

```
castVote
Input  : Library function: f, Project that uses library function f:
         Proj
Output: Candidate error specification for function f: C*
countMap ← ∅
lengthMap ← ∅
for each constraint c on the return value of f ∈ Proj do
    countMap[c] ← nBranch_{Proj,f,c}
    lengthMap[c] ← MedianStmtCnt(Proj, f, c)
end
C* ← {c : countMap[c] < Avg(countMap \ {c}) − stdev}
if C* ≠ ∅ then
    return C*
end
C* ← {c : lengthMap[c] < Avg(lengthMap \ {c}) − stdev}
if C* ≠ ∅ then
    return C*
end
return ∅
```

Algorithm 1: Algorithm for extracting potential error constraints from each project

```
InferSpec
Input  : Library function: f, Projects that use library function f:
         PROJS
Output: Inferred error specification for function f: C_f
voteMin ← lowerQuartile({nPathCnt : Proj ∈ PROJS})
voteMap ← ∅
for each Proj ∈ PROJS do
    if nPathCnt_{Prog,f} ≥ voteMin then
        C* ← castVote(f, Proj)
        for each c ∈ C* do
            voteMap[c] ← voteMap[c] + 1
        end
    end
end
if nChoices(voteMap) > MaxChoices then
    return infallible
end
return {c : voteMap[c] > Avg(voteMap \ {c}) + stdev}
```

Algorithm 2: Algorithm for error specification mining

functions. Moreover, in its current form, Clang does not support extracting the path constraints for each path. We modified Clang's constraint manager class to print out the path constraints in text format.

For each path through the caller function explored by the symbolic execution engine, we collected the following information: (i) The name of the caller function and the name of the API function(s) called, and the constraints on API function's return values. (ii) the number of program statements, and function calls along the path. (iii) the number of paths satisfying the constraints on the return values. Our current prototype implementation supports three types of constraints for three different data types: (i) a pointer can either be `NULL` or not `NULL`, (ii) constraints on an integer are represented by a sorted list of non-overlapping ranges, and (iii) a C++ `bool` type can be either `true` or `false`.

Table 4 shows the performance of APEx for the 5 largest programs in our data set when run on a machine with 2 Intel Xeon 2.67GHz processors with 4 cores each and 24 GB of memory. For each program-library pair, we counted the time to gather the paths, generate specifications, and find bugs. For each program, we show the average times over all the libraries.

Table 4: APEx analysis times for the 5 largest programs in our test set (averaged over all library APIs) when run on a a machine with 2 Intel Xeon 2.67 GHz processors with 4 cores each and 24 GB of memory

Project	Lines of Code	Average analysis time (s)
ClamAV	575104	1h 3m 39s
Pidgin	343416	1h 37m 50s
Grep	285705	6m 59s
GnuTLS	179322	59m 18s
Coreutils	175353	26m 08s

6. EXPERIMENTAL RESULTS

In this section, we present an empirical evaluation of the ability of APEx to correctly infer error specifications of C API functions. We also evaluate how effective the inferred error specifications are for finding new bugs.

6.1 Study Subjects

We studied 6 C libraries (GnuTLS, Libgcrypt, GTK+, libc, OpenSSL, and zlib) across 28 different projects, as shown in Table 5. Overall, we analyzed 67 combinations of libraries and software projects. In general, for a given library, we analyzed the entire project source code. However, in certain projects, only a part (*e.g.,* particular files or modules) uses the relevant library functions. In such cases, we analyzed only the relevant parts of the project, if we could isolate them. For example, we analyzed only the files that use OpenSSL and Libgcrypt for Curl, the modules that use OpenSSL and zlib for Httpd, and the applications that are included with OpenSSL using OpenSSL API functions. In total, we studied 217 API functions from 16788 distinct call sites.

6.2 Study Methodology

We measure APEx's capability of inferring error specifications in terms of precision and recall. Suppose, we analyzed the usage of E API functions, using APEx. A perfect tool could infer correct error specifications for all the cases in E. However, in practice, APEx will only infer error specifications for APIs A and among them only A' will be correct. In this case, we define the precision and recall of APEx as follows: **Precision.** The percentage of error specifications correctly inferred by APEx over the total number of inferred specifications, *i.e.* $\frac{|A'|}{|A|}$. **Recall.** The percentage of error specifications correctly inferred by APEx over expected number of the specifications, *i.e.* $\frac{|A'|}{|E|}$.

To evaluate the accuracy of APEx's classification of error paths, we calculate the precision and recall using the same formula. However, in this case, E, A, and A' would be the set of all error paths, detected error paths, and correct error paths respectively.

6.3 Study Results

We start with investigating how well APEx can identify error paths since the accuracy of inferring error specifications depends on this step. Thus, we begin with a straightforward question, namely:

RQ1. How accurately can APEx identify error paths?

Table 6 shows that APEx can identify error paths with 95% precision and 66% recall. For the GnuTLS, GTK+, and libc libraries, APEx's precision is more than 90%. APEx's precision is over 80% for the Libgcrypt and OpenSSL libraries. However, APEx does not perform so well for library zlib, with a precision of only 50%. Out of the 4 mistakes in zlib APIs, 2 were cleanup func-

Table 5: Details of study subjects

Project	Libgcrypt	GnuTLS	GTK+	libc	OpenSSL	zlib
ClamAV	-	-	-	20 (1406)	7 (18)	16 (83)
Collectd	10 (33)	-	-	20 (764)	-	-
Coreutils	-	-	-	20 (585)	-	-
Cryptmount	13 (35)	-	-	17 (340)	-	-
cUrl	-	4 (4)	-	18 (396)	30 (43)	3 (9)
Diff	-	-	-	16 (204)	-	-
Evince	-	-	45 (403)	10 (91)	-	-
Gedit	-	-	44 (521)	2 (27)	-	-
GnuPG	1 (1)	-	-	19 (676)	-	-
GnuTLS	-	47 (680)	-	19 (598)	-	4 (4)
Grep	-	-	-	12 (123)	-	-
Httpd	-	-	-	3 (124)	26 (87)	5 (21)
Lighttpd	-	-	-	19 (630)	12 (21)	2 (6)
Lynx	-	2 (2)	-	18 (1029)	8 (13)	7 (9)
Mutt	-	2 (6)	-	21 (502)	15 (19)	-
Nginx	-	-	-	13 (186)	-	3 (7)
OpenSSH	-	-	-	20 (752)	16 (173)	4 (6)
OpenSSL	-	-	-	11 (111)	34 (236)	-
Pidgin	-	4 (8)	39 (861)	20 (1464)	-	-
Remmina	8 (14)	-	42 (461)	7 (22)	-	-
RSync	-	-	-	20 (331)	-	-
Tar	-	-	-	18 (315)	-	-
Teleport	10 (27)	-	2 (3)	6 (15)	-	-
Tor	-	-	-	18 (666)	29 (74)	4 (11)
Totem	-	-	40 (356)	6 (68)	-	-
Uget	1 (1)	-	39 (589)	9 (51)	-	-
VPNC	18 (54)	-	-	16 (51)	-	-
Wget	-	2 (2)	-	21 (340)	18 (20)	-
Total	26 (165)	47 (702)	50 (3194)	23 (11867)	52 (704)	19 (156)

Note: Each entry contains studied number of functions per library per project and (number of unique call sites).

Table 6: Accuracy of predicting error paths

Library	Precision	Recall
Libgcrypt	0.80	0.82
GnuTLS	0.91	0.26
GTK+	0.99	0.91
libc	0.96	0.73
OpenSSL	0.86	0.71
zlib	0.50	0.15
All	0.95	0.66

tions: `deflateEnd` and `gzclose`, for which APEx incorrectly picked the value representing success as the error specification. Our heuristic that error paths are simpler does not tend to work very well for detecting error specifications of cleanup functions as they are usually called at the end of main computation, just before returning from the caller function. Therefore, neither the error paths nor the non-error paths for these cleanup functions performs any significant amount of computation.

We found that APEx's recall is more than 70% for libraries Libgcrypt, GTK+, libc, and OpenSSL. However, for libraries zlib and GnuTLS, APEx's recall is significantly lower—15% and 26%, respectively. As shown in Table 5, we found that the distribution of the zlib and GnuTLS API functions are not distributed evenly across our test projects. While several unique GnuTLS and zlib functions appear many times in the GnuTLS and ClamAV projects respectively, not many functions appear elsewhere. Therefore, APEx cannot infer error specifications for functions that do not have enough diverse samples.

Result 1: *APEx can detect error paths with 95% precision and 66% recall.*

RQ2. How accurately can APEx identify API error specifications?

Table 7 shows that APEx can infer error specifications with 77% precision and 47% recall on average for all the projects. For all five libraries except zlib the precision varies from 74% to 84%. However, for zlib, APEx's precision drops to 50%. Note that, this is not particularly surprising given that APEx did not perform well in detecting error paths for zlib as well (see Table 6) mostly due to the presence of several cleanup functions.

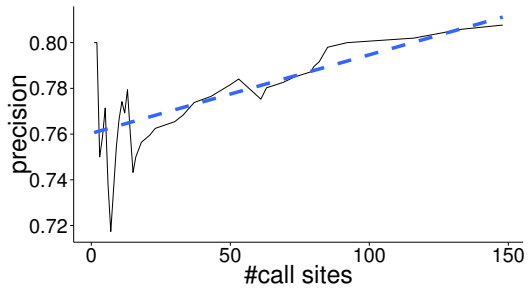
Table 7: Accuracy of inferring error specifications

Library	Precision	Recall
Libgcrypt	0.82	0.64
GnuTLS	0.74	0.34
GTK+	0.84	0.36
libc	0.76	0.65
OpenSSL	0.78	0.62
zlib	0.50	0.27
All	0.77	0.47

APEx's recall for inferring error specifications is above 62% for libraries Libgcrypt, libc, and OpenSSL. The recall is around 30% for the other three libraries. Note that these results closely resemble the recall for error path detection. As mentioned earlier, the drop in recall is primarily caused by the low number of call sites across programs for zlib and GnuTLS APIs.

Since APEx infers error specifications by learning from multiple call sites, next we check how APEx's performance varies with the number of call sites, *i.e.* whether APEx performs better for the functions with a larger number of call sites.

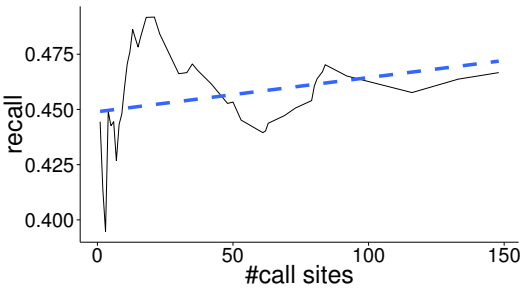
Figure 4a shows the variation of precision with the number of distinct call sites of the tested API functions. Overall, as expected, we see a positive trend (indicated by the blue, dashed line), *i.e.*



#call sites		Spearman Correlation	
quantile	count	estimate	p-val
less than 33%	less than 8	-0.86	2.61E-08 ***
33% to 66%	8 to 50	0.29	0.15602604
greater than 66%	greater than 50	0.98	6.08E-07 ***
all	1 to 1282	0.55	2.71E-07 ***

Note: The Table shows that overall precision increases as the number of call sites increases with statistical significance (marked with ***); however, this trend reverses at lower percentile of the call sites.

(a) Precision



#call sites		Spearman Correlation	
percentile	count	estimate	p-val
less than 33%	8	0.03	0.889
33% to 66%	8 to 50	0.29	0.171
greater than 66%	50	0.85	0.000 ***
all	1 to 1282	0.55	0.000 ***

Note: The Table shows that, overall recall increases with statistical significance (marked with ***) as the number of call sites increases; however, this trend is not significant below the 66 percentile in call site counts.

(b) Recall

Figure 4: The accuracy of inferring error specifications varies significantly with the number of distinct call sites of the API functions.

precision increases with the increase of call sites. Also, the Spearman correlation is positive (0.55) with statistical significance. We further inspect this trend at three different ranges of call sites: (i) at lower range ($< 33\%$), with functions having less than 8 distinct call sites, we see that the precision decreases as the number of call-sites increases; (ii) at medium range, with functions having 8 to 50 distinct call-sites, we see a positive trend, although it is not statistically significant.; and (iii) finally, at higher range ($> 66\%$), with call sites more than 50, we see a distinct positive trend (correlation=0.98) with statistical significance.

A similar trend is observed for recall as well, as shown in Figure 4b. Overall, recall increases with the number of call sites with statistical significance (Spearman correlation 0.55). However, it loses the statistical significance in the lower range.

We further compare APEx's performance with baseline, trivial error specifications: `< 0` for integers and `== NULL` for pointers. The following table shows the results. For integer error return values, APEx's precision is 68%, which is 18 percentage points better than the baseline case. Also, for pointer return values, APEx reports 95% precision, *i.e.* 5 percentage points better than the baseline. Overall, APEx performs 13 percentage points better than the baseline.

Table 8: Comparison of APEx performance with baseline performance with trivial error specifications: `< 0` for integers and `== NULL` for pointers

	Baseline	APEx
Integer	0.50	0.68
Pointer	0.90	0.95
Overall	0.64	0.77

In particular, APEx is more effective for integer error types, since they can have more diverse error values as compared to pointer error types with only `NULL` and non-`NULL` options. For library functions like OpenSSL, where error specifications are more diverse, APEx becomes more effective. For example, in OpenSSL, APEx could detect integer error specifications with 62% precision, as opposed to the baseline performance of 12%, or an overall 50 percentage point gain in the precision.

Result 2: *APEx can infer error specifications with 77% precision and 47% recall. Overall accuracy increases as the number of call sites increases.*

RQ3. How effectively can APEx detect missing error checks?

Table 9: Precision of detecting potential error handling bugs

Library	Unfiltered Bugs	Filtered Bugs	Real Bugs	Precision
Libgcrypt	59	7	2	0.286
GnuTLS	52	32	19	0.594
GTK+	825	-	-	
libc	2243	93	66	0.710
OpenSSL	51	38	26	0.684
zlib	26	5	5	1.000
Total	3256	175	118	0.674

In order to measure the effectiveness of the inferred specifications in finding error handling bugs, we build a simple bug detection scheme using the inferred specifications. Given a fallible API function and its inferred error specification, we investigate whether a caller of the API function checks the errors returned by the API function correctly; If not, we report it as a potential *error-handling bug*. In Table 9, column "Unfiltered Bugs" shows the counts. In total, we found 3,256 cases of missing error checks. However, not all missing checks lead to potential bugs. We randomly selected 50 such cases, and manually checked the call site to determine whether the error could occur, given the information we can observe about the caller function up to the call. If so, we classified the missing check as a potential bug. In practice, the bug might still not occur because of checks further up the call trace, or configurations that eliminate potential failures. We concluded that only 36 of them can be potential bugs. Most of the errors are caused due to either incorrectly inferred error specifications (*e.g.,* libc function `strerror`

Table 10: Samples of confirmed error-handling bugs reported by APEx.

Example 1
Library : **OpenSSL**
Projects : Lynx, Mutt
API function: **SSL_CTX_new**
Status: Acknowledged and Fixed

```
static int ssl_socket_open (CONNECTION * conn)
{
...
  data->ctx = SSL_CTX_new (SSLv23_client_method ());
  /* disable SSL protocols as needed */
  if (!option(OPTTLSV1))
  {
    SSL_CTX_set_options(data->ctx, SSL_OP_NO_TLSv1);
  }
...
}
```

Example 2
Library : **GnuTLS**
Projects : Pidgin
API function: **gnutls_x509_crt_init**
Status: Acknowledged, Fixed, and CVE being requested

```
static PurpleCertificate *
x509_import_from_datum(const gnutls_datum_t dt, gnutls_x509_crt_fmt_t
      mode)
{
...
    gnutls_x509_crt_init(&(certdat->crt));
...
}
```

Example 3
Library : **Libgcrypt**
Projects : Collectd, Remmina, VPNC
API function: **gcrypt_control**
Status: Acknowledged and Fixed in Remmina

```
gcry_control (GCRYCTL_SET_THREAD_CBS, ...);
gcry_control (GCRYCTL_INIT_SECMEM, ..., ...);
```

is actually infallible but was incorrectly classified by APEx to have an error specification of =NULL) or for functions which only fail if they are called with incorrect input values (*e.g.,* Libgcrypt function `gcry_cipher_setiv`).

In order to detect error handling bugs with more serious consequences, we filter out functions whose return values are unchecked for a significant majority of the call sites in a function. By significant, we mean, as in our error specification voting scheme, that the number of call sites that never check the return value is at least one standard deviation higher than that of the remaining sites. We further excluded the library GTK+ in this step, as most of the GTK+ API functions only return error when invoked with invalid inputs and most of the input values are already checked before calling the API functions. The third column of Table 9 (see the column "Filtered Bugs") shows the reported bugs. In total, we reported 175 missing error checks, and a manual investigation reveals 118 of them as potential error-handling bugs. Thus overall we can detect potential error-handling bugs with 67.4% precision. Among them we performed best for zlib with a precision of 100%. We performed poorly for Libgcrypt functions with only 28.6% precision due to the prevalence of formatting functions such as `gcry_mpi_scan` that never fail when invoked with a valid format string.

We are now in the process of reporting these bugs to the developers. So far, we have reported 17 bugs. Among them, 4 have already been patched by the developers. We provide the details of three example bugs that developers confirmed in Table 10.

Example 1 of Table 10 shows that the return value for OpenSSL API function `SSL_CTX_new` was stored in `data->ctx` without any checking. It subsequently was passed as an argument into the API function `SSL_CTX_set_options`. The latter function would crash when a `NULL` pointer is returned by a failed call to `SSL_CTX_new`. APEx detected 3 instances of this bug in projects Lynx and Mutt. All the instances are acknowledged by the corresponding developers. The Mutt developers have already fixed the code while the Lynx developers are in the process of fixing it. [3]

Example 2 of Table 10 shows a bug found in project Pidgin, where a check was missing for the GnuTLS API function `gnutls_x509_crt_init`. In case of a failure, the function returns an error and a missing check renders the certificate `certdat->crt` invalid. The developers acknowledged and fixed this bug. Given the security sensitive nature of this bug, the developers have also reserved a CVE-ID (Common Vulnerabilities and Exposures Identifier [4]), CVE-2016-1000030, for it.

Finally, for Libgcrypt (example 3), APEx found 6 instances of missing checks for API function `gcry_control` in projects Collectd, Remmina, and VPNC. `gcry_control` is a variadic function that takes at least one argument, and is fallible for arguments `GCRYCTL_SET_THREAD_CPS` and `GCRYCTL_INIT_SECMEM`. The latter is of particular importance not only for enabling secure memory but also dropping program privileges [5]. Currently, we have one acknowledgment for this kind of bug from the developers at Remmina, who used the `GCRYCTL_SET_THREAD_CBS` command, and have fixed the bug. [6]

Result 3: *Using APEx's error specifications, 118 new error-handling bugs were detected.*

We found that incorrect error handling is pervasive for certain API functions. In fact, errors in such API functions are routinely ignored in the majority of their call sites. Table 11 shows some examples from OpenSSL and GnuTLS libraries where more than 50% of their call sites fail to perform correct error handling.

Table 11: Sample API functions for which $\leq$ 50% of the call sites performed correct error checking

Library	Function	call sites with correct error checking (%)
GnuTLS	gnutls_x509_crt_get_dn_by_oid	21%
OpenSSL	X509_NAME_get_entry	25%
	SSL_shutdown	29%
	SSL_write	47%
	SSL_do_handshake	50%
	SSL_get_privatekey	50%

7. RELATED WORK

Static detection of error handling bugs. Static code checkers take source code and invariants as input and determine whether the specifications are violated [18, 13]. There is a long line of work using static analysis techniques for detecting different types of bugs including security bugs [10, 9, 14]. In this paper, we primarily focus on techniques whose primary goal is to find error handling bugs.

There are several prior projects that designed specialized static bug finding tools for finding error handling bugs. For example, Rubio-González *et al.* [33] and Gunawi *et al.* [17] created static

[3]https://dev.mutt.org/hg/mutt/rev/00c0c155d992
[4]https://cve.mitre.org/
[5]https://gnupg.org/documentation/manuals/gcrypt-devel/Controlling-the-library.html
[6]https://github.com/FreeRDP/Remmina/issues/830#issuecomment-208383995

bug finders for detecting error handling bugs in Linux file system code. Lawall *et al.* [20] built and evaluated another static bug finding tool for finding error handling bugs in Secure Sockets Layer (SSL) implementations. Weimer *et al.* [37, 39] have developed bug finding tools for finding exception handling bugs in Java programs. Robillard *et al.* [31, 32] have built tools for simplification and visualization of exception handling flow that can help developers minimize mistakes in exception handling code.

The error specifications inferred by APEx can be used with existing bug finding techniques like the ones described above for finding error handling bugs.

Dynamic fault injection. As error conditions rarely appear during regular operation, finding error handling bugs is hard using regular testing methods. To avoid such issues, researchers have used fault injection to dynamically exercise error handling code in a program by injecting synthetic failures. Marinescu *et al.* have developed a general-purpose fault injection infrastructure for running tests with injected faults [22, 23, 24]. Broadwell *et al.* applied fault injection to test the recovery mechanisms of live systems [11], while Süßkraut *et al.* used fault injection for determining where patches are required to insert proper error handling code [35]. However, all these fault injection techniques require fault profiles for deciding which functions can fail and what values will be returned in case of failures. APEx can make dynamic fault injection techniques completely automated as the fault profiles can be auto-generated from the error specifications.

Specification mining. The closest work to ours for automatically mining error specifications is by Acharya *et al.* [7]. However, they assumed that error handling code must be completely contained inside a branch statement conditional on the return value of an API function and must have an explicit return/exit statement. In such cases, they identify the corresponding branch condition as the API's error specifications. However, unlike APEx, this heuristic does not work for functions that can return multiple error or non-error values. Also, unlike us, they used a data-flow insensitive code analysis technique based on [13] that limits their accuracy.

Rubio-González *et al.* [34] and Marinescu *et al.* [23, 24] have used program-specific heuristics (*e.g.,* a fixed range of error values, treating all compiler-generated constants as error codes) to infer API error specifications in their respective settings that do not work across different libraries/programs. Unlike these techniques, APEx focuses on inferring the API error specifications (*i.e.* the range of possible error values) from a large number of programs in an automated manner that works for a diverse set of APIs/projects.

Several prior research projects have mined different types of specifications from source code to help software developers. For example, Buse *et al.* try to aid developers in properly handling errors by inferring the exact causes of the exceptions [12]. Acharya *et al.* [8] attempted to learn the proper order for calling API functions, while Nguyen *et al.* [27] focused on automatically discovering API preconditions. More generally, for inferring proper programming practices, Engler *et al.* [15] sought to infer assumptions that the programmer makes and check if any code contradicts such beliefs. Weimer *et al.* assumed that the programmer's desired specifications can be inferred specifically from normal, presumably non-buggy code and then compared against potentially-buggy exceptional cases [38]. Our work complements these projects by focusing on automated inference of error specifications that can be used to find bugs in error handling code.

Another line of research tries to automatically infer API specifications from documentation (*e.g.,* user manuals, comments, *etc.*) using natural language processing [42, 36]. However, unlike mining specifications from API usage, these approaches are susceptible to errors/omissions in documentation.

8. THREATS TO VALIDITY

External validity. External validity concerns the generalizability of our result. The effectiveness of our technique may be limited by the representativeness of our data set. To minimize this threat we tested our technique on 217 functions from 6 popular open source APIs across 28 programs. However, other APIs that are closed source or more specialized might have their own requirements for handling errors that contradict our observations.

Construct validity. APEx works on the assumption that error paths have distinct features from non-error paths. However, this may not be true for all functions. In fact, we already observed two types of functions that are different: cleanup functions and infallible functions returning invalid-looking values. As we observed before, half of the false error specifications in zlib are cleanup functions and other functions that come at the end of the caller. While we still expect the error handling code to be minimal, the normal path is likely going to be even simpler for the cleanup routines, because the caller intends to use them at the end of a task. Infallible functions could return values that appear invalid and need to be checked, but not due to failures of the function. For example, boolean return values from infallible functions are almost always used in conditional statements. If one value leads to more complex branches, the function may be classified as fallible, as was the case of `gtk_widget_get_visible` in GTK+. Likewise, `memset` and `memcpy` from libc return the first parameter, which could be `NULL`. If the caller decides to check it after the call, the `NULL` value would be attributed to a failure of the API function. To minimize such threats, we perform an empirical study in Section 3 and show that, for the majority of the API functions, error paths indeed show distinctive traits.

Internal validity. The precision of our bug finding results is based on the authors' judgment. Although multiple authors verified the bugs, we are planning to report these bugs to the corresponding developers and already have started the process. As mentioned earlier, several of them have already been acknowledged and fixed by the developers.

9. CONCLUSIONS

In this paper, we introduced APEx, a tool for automatically inferring the error specifications of API functions based on the insight that error paths are often simpler than regular paths (*i.e.* they have lower numbers of branches, statements, and functions). We evaluated our technique over 28 projects using 6 popular libraries and demonstrated that our technique can accurately infer error specifications for different API functions. We also used the inferred specifications to find 118 previously unknown potential error handling bugs in the 28 tested projects.

10. ACKNOWLEDGEMENTS

This work is sponsored in part by the National Science Foundation (NSF) grants CNS-16-17670, CNS-16-18771, and Air Force Office of Scientific Research (AFOSR) grant FA9550-12-1-0162. 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 or AFOSR.

11. REFERENCES

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

[2] CVE-2014-0092. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092, 2014.

[3] CVE-2015-0208. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0208, 2015.

[4] CVE-2015-0285. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0285, 2015.

[5] CVE-2015-0288. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0288, 2015.

[6] CVE-2015-0292. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0292, 2015.

[7] M. Acharya and T. Xie. Mining API Error-Handling Specifications from Source Code. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.

[8] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *ACM SIGSOFT symposium on The foundations of software engineering (FSE)*, 2007.

[9] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[11] 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.

[12] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[13] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)*, 2002.

[14] B. Chess and J. West. *Secure programming with static analysis*. Pearson Education, 2007.

[15] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *International symposium on Software testing and analysis (ISSTA)*, pages 1–4. ACM, 2007.

[17] H. Gunawi, C. Rubio-González, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[18] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[19] S. Jana, Y. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs using Error Specifications. In *USENIX Security Symposium (USENIX Security)*, August 2016.

[20] J. Lawall, B. Laurie, R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference (EDCC)*, 2010.

[21] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, (6):546–558, 1979.

[22] P. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conference*, 2010.

[23] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), 2011.

[24] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 379–388. IEEE, 2009.

[25] M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan. An empirical study of goto in C code from GitHub repositories. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 404–414. ACM, 2015.

[26] B. A. Nejmeh. NPATH: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[27] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 166–177. ACM, 2014.

[28] OWASP top 10. https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf.

[29] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, volume 46, pages 504–515. ACM, 2011.

[30] D. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *USENIX Security Symposium*, 2015.

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

[32] M. P. Robillard and G. C. Murphy. Designing robust Java programs with exceptions. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 2–10. ACM, 2000.

[33] C. Rubio-González, H. Gunawi, B. Liblit, R. Arpaci-Dusseau, and A. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009.

[34] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *PASTE*, 2010.

[35] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. In *Sixth European Dependable Computing Conference (EDCC)*, pages 13–22. IEEE, 2006.

[36] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* icomment: Bugs or bad comments?*/. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158, 2007.

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

[38] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

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

[40] D. A. Wheeler. Sloccount. Available at http://www.dwheeler.com/sloccount/, 2015.

[41] H. Zhong and Z. Su. Detecting API documentation errors. In *ACM SIGPLAN Notices*, volume 48, pages 803–816. ACM, 2013.

[42] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *International Conference on Automated Software Engineering (ASE)*, pages 307–318, 2009.