Generating Precise Error Specifications for C: A Zero Shot Learning Approach

BAIJUN WU, UL Lafayette, USA
JOHN PETER CAMPORA III, UL Lafayette, USA
YI HE, UL Lafayette, USA
ALEXANDER SCHLECHT, UL Lafayette, USA
SHENG CHEN, UL Lafayette, USA

In C programs, error specifications, which specify the value range that each function returns to indicate failures, are widely used to check and propagate errors for the sake of reliability and security. Various kinds of C analyzers employ error specifications for different purposes, e.g., to detect error handling bugs, yet a general approach for generating precise specifications is still missing. This limits the applicability of those tools.

In this paper, we solve this problem by developing a machine learning-based approach named MLPEx. It generates error specifications by analyzing only the source code, and is thus general. We propose a novel machine learning paradigm based on transfer learning, enabling MLPEx to require only one-time minimal data labeling from us (as the tool developers) and zero manual labeling efforts from users. To improve the accuracy of generated error specifications, MLPEx extracts and exploits project-specific information. We evaluate MLPEx on 10 projects, including 6 libraries and 4 applications. An investigation of 3,443 functions and 17,750 paths reveals that MLPEx generates error specifications with a precision of 91% and a recall of 94%, significantly higher than those of state-of-the-art approaches. To further demonstrate the usefulness of the generated error specifications, we use them to detect 57 bugs in 5 tested projects.

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

Additional Key Words and Phrases: Error specification generation, machine learning, project-specific features

ACM Reference Format:
Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating Precise Error Specifications for C: A Zero Shot Learning Approach. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 160 (October 2019), 30 pages. https://doi.org/10.1145/3360586

1 INTRODUCTION

Reliable and secure software should be fail-safe or fail-secure when functions fail [Cristian 1982]. When a failure occurs, such as memory allocation failure or permission denial, the function that detects the failure should return an error to its caller, which checks the error to handle the failure gracefully. Since the C language lacks exception handling mechanisms [Goodenough 1975],

Authors' addresses: Baijun Wu, School of Scomputing and Informatics, UL Lafayette, 301 E Lewis St. Lafayette, LA, 70503, USA, bj.wu@louisiana.edu; John Peter Campora III, School of Scomputing and Informatics, UL Lafayette, 301 E Lewis St. Lafayette, LA, 70503, USA, campora@louisiana.edu; Yi He, School of Scomputing and Informatics, UL Lafayette, 301 E Lewis St. Lafayette, LA, 70503, USA, yi.he1@louisiana.edu; Alexander Schlecht, School of Scomputing and Informatics, UL Lafayette, 301 E Lewis St. Lafayette, LA, 70503, USA, ads1937@louisiana.edu; Sheng Chen, School of Scomputing and Informatics, UL Lafayette, 301 E Lewis St. Lafayette, LA, 70503, USA, chen@louisiana.edu.

2475-1421/2019/10-ART160
https://doi.org/10.1145/3360586

programmers use certain values that differ from the return values of normal executions to check and propagate errors. We refer to such sets of "certain values" as *error specifications*.

An example error specification is given in Figure 1. The function `deflatePrime()` from zlib tries to insert bits in a given stream `strm`, but may fail in two cases. When failures happen, it will return `Z_STREAM_ERROR` to indicate an inconsistent stream state or `Z_BUF_ERROR` to indicate that the stream does not have enough buffer to insert the bits. Since zlib sets the values of the two macros `Z_STREAM_ERROR` and `Z_BUF_ERROR` to -2 and -5, respectively, the error specification for `deflatePrime()` is {-2, -5}.

It is possible that the error specification of a function is empty. This is because some functions such as `strcmp()` are related to the core algorithmic logic and do not return errors.

```
1  /* zlib-1.2.11/deflate.c */
2  int ZEXPORT deflatePrime (
3    z_streamp strm, int bits, int value)
4  {
5    deflate_state *s;
6    int put;
7
8    if (deflateStateCheck(strm))
9      return Z_STREAM_ERROR;
10
11   s = strm->state;
12   if ((Bytef *)(s->d_buf) < ...)
13     return Z_BUF_ERROR;
14   ...
15   return Z_OK;
16 }
```

Fig. 1. An example error specification. The function `deflatePrime()` returns `Z_STREAM_ERROR` and `Z_BUF_ERROR` on error (marked in italic-red) while returns `Z_OK` on success (marked in bold-green).

1.1 Usefulness of Error Specifications and Challenges in Generating Them

Error specifications are very useful for C analyzers. Many static and dynamic analysis methods [Gunawi et al. 2008; Jana et al. 2016; Marinescu and Candea 2011; Rubio-González et al. 2009; Susskraut and Fetzer 2006; Tian and Ray 2017] employ error specifications to detect error handling bugs, which are caused by missing or incomplete error handling and are prevalent in real-world software projects [OWA 2007].

To illustrate, consider the code snippet in Figure 2, which shows how a bug detection tool, such as EPEx [Jana et al. 2016], may exploit error specifications to detect a bug in the Linux kernel (versions before 4.20.15) that causes a security vulnerability (CVE-2019-12818 [CVE 2019]). The bug is related to Logical Link Control Protocol (LLCP) and is in the NFC subsystem. Note that, as line 17 shows, `nfc_llcp_build_tlv()` returns `NULL` when a memory allocation fails (line 16) and no memory address is returned. As a result, the error specification for `nfc_llcp_build_tlv()` is {`NULL`}. The function `nfc_llcp_build_gb()` calls `nfc_llcp_build_tlv()` to allocate a block of memory for an NFC data exchange message (line 5). It directly uses the returned memory on line 7, ignoring the fact that the memory address could be `NULL`. This may trigger a NULL pointer dereference, causing a denial of service failure. To determine if the call to `nfc_llcp_build_tlv()` (line 5) leads to error handling bugs, EPEx examines whether the return value (`lto_tlv`) is properly checked against the corresponding error specification ({`NULL`}). Since such a check is missing, EPEx may report an error handling bug.

Besides detecting error handling bugs, error specifications can be also used in many other application scenarios [Cheon 2007; Dillig et al. 2007; Myers and Stylos 2016; Rubio-González and Liblit 2010; Weimer and Necula 2005], for example for verifying the consistency between API documentations and system implementations.

Automatically generating precise error specifications is non-trivial in practice for several reasons. First, while some conventions, such as NULL pointers and negative values, are used in projects to denote errors, they do not always hold. This is because error values indicating failure are function-specific and may overlap with non-error values from other functions within the same project. For example, in zlib, the function `gzgets()` uses `NULL` to denote that reading a compressed file fails

```
1  /* Linux-4.19/llcp_core.c */
2  static int nfc_llcp_build_gb (...)
3  {
4    ...
5    lto_tlv = nfc_llcp_build_tlv(...);
6    ...
7    memcpy(gb_cur, lto_tlv, lto_length);
8    gb_cur += lto_length;
9    ...
10 }
```

```
11 /* Linux-4.19/llcp_commands.c */
12 u8 *nfc_llcp_build_tlv (...)
13 {
14   ...
15   tlv = kzalloc(...);
16   if (tlv == NULL)
17     return tlv;
18   ...
19   return tlv;
20 }
```

Fig. 2. An example of an error handling bug. Function `nfc_llcp_build_tlv()` returns NULL on error (marked in italic-red) and the start address of the allocated memory (marked in bold-green) on success.

whereas `gzerror()` returns `NULL` to indicate a compressed file stream is correct. According to Kang et al. [2016], the precision of error specifications collected following such conventions is only about 50%. Second, in addition to such conventions, projects usually introduce their own idioms for representing errors. For example, while OpenSSL uses `SSL_R_UNSUPPORTED_OPTION` to indicate errors, Linux never uses this name. As a result, identifying errors by pattern matching over a set of idioms does not work across projects.

1.2 Limitations of Existing Approaches

Several approaches have been developed to help generate error specifications [Acharya and Xie 2009; Kang et al. 2016; Marinescu and Candea 2011]. However, while they work well under certain conditions, they are generally limited in two aspects. First, they have a limited generalizability. For example, LFI [Marinescu and Candea 2011] gets error specifications for only exported library functions, through a certain register where all application binary interfaces (ABIs) place their return values. As another example, APEx [Kang et al. 2016] extracts error specifications for APIs that are frequently used by other projects only. To make the generated error specification precise, in general, APEx needs to collect information from more than 100 call sites for each API. This limitation prevents us from inferring error specifications for internal functions in libraries like OpenSSL and system software like Linux.

Second, the current approaches to generating error specifications have a limited accuracy[1]. While LFI assumes only constant values indicate error situations, constant values can indicate non-error situations and non-constant values can indicate error situations. In general, as noted by Kang et al. [2016], this approach yields quite poor precision across different libraries. The accuracy of APEx is dramatically affected by the number of call sites using APIs for which error specifications are to be collected. The overall precision and recall of APEx are 77% and 47%, respectively.

These limitations restrict the applicability of the tools relying on error specifications. For instance, the state-of-the-art work to detect error handling bugs [Jana et al. 2016] still requires manually supplying error specifications, since automatically generated error specifications are not yet precise enough, or even missing for internal functions.

1.3 Our Solution

In this paper, we generate error specifications by first identifying error paths and then collecting return values for these paths. A path is an *error path* if its execution ends in error situations, for example, failing to open files, obtain resources, and allocate memory. For instance, in the function

[1] We measure the error specification accuracy by using the metrics precision, recall, and F-measure. We give the formal definitions of these metrics in Section 6.4. The F-measure is a measure of the overall accuracy by computing the harmonic mean of precision and recall.

`nfc_llcp_build_tlv` (Figure 2), the path that failed to allocate memory (ends in line 17) is an error path and the path that obtained memory (ends in line 19) is a non-error path.

The key observation in our solution is that we can characterize paths through a set of features, such as path lengths (the distances between the exit points and the corresponding entry points), categories of return expressions (constants, variables, macros, etc.), and the number of function calls involved along a path. These features are related to the behaviors of programs, and are thus useful to identify *error paths*. For example, since error values in C are often returned by using "goto" statements [Nagappan et al. 2015], error paths are likely to have fewer number of function calls and/or branching points than non-error paths.

Based on path features, one straightforward way to identify error paths is to employ some heuristics. In APEx [Kang et al. 2016], the paths with fewer branching points and program statements than average are identified as error paths. However, the accuracy of error path identification in heuristic-based approaches is low. For example, the F-measure[1] of APEx is only about 77%. The main reason is that the heuristic used in APEx only leverages two features, which is not comprehensive enough to cover different kinds of error paths. To improve the accuracy, one can design a more sophisticated heuristic considering more features. Ideally, a heuristic that appropriately, consistently, and coherently makes full use of all possible features could identify error paths with very high accuracy. On the other hand, manually designing such heuristics is not feasible in practice when too many features are involved. Moreover, the identified error paths based on heuristics lack statistical guarantees, which means that the proposed heuristic may not generalize across different projects. For example, APEx achieves 94% F-measure for GTK+ while only 23% F-measure for zlib.

To address these issues, we propose a machine learning-based method, named MLPEx, which effectively establishes a statistical relationship between the features and error paths. One inherent problem of traditional machine learning methods is that considerable manual data labeling efforts are required for training a model. We build upon a simple insight that enables MLPEx to move beyond such a supervised paradigm: although the paths in different projects exhibit distinctive properties due to coding style, functionality, etc., they share commonalities across projects. For example, one common feature to characterize paths could be path length, even though the functions invoked by paths vary from one project to another. It is thus possible that the knowledge, in terms of the common characteristics of error and non-error paths, could be transferred from a pre-labeled project to another project that has no labeled paths. Based on the transferred knowledge, we learn a "tailored" model for the unlabeled project, eliminating the needs for arduous labeling overhead.

We formalize this intuition by proposing a novel two-phase learning approach and present its workflow in Figure 3. In the first phase, transfer learning technique [Pan and Yang 2010] is introduced to help transfer the knowledge from a pre-labeled project to a new project without its label information, yielding a core model. The core model samples a set of the most likely error and non-error paths. They are then used as the training data set (since the set is the output of the first learning phase, we refer to it as the *learned training set* hereafter) in the second phase, where a project-specific model is trained to classify all paths as error or non-error in the new project.

On average, MLPEx learns the exact error specifications for more than 90% of functions, a significant improvement over the state-of-the-art approach APEx [Kang et al. 2016], whose F-measure is about 58%. In addition, while APEx infers error specifications for APIs only, MLPEx learns error specifications for APIs as well as internal functions, as long as the source code is available. In summary, the main contributions of this paper are as follows:

(1) We develop a "zero-shot" learning approach, requiring no manual labeling effort from users to train a model for error path prediction. Theoretical analysis and evaluation results verify that our approach can generalize to different C projects.

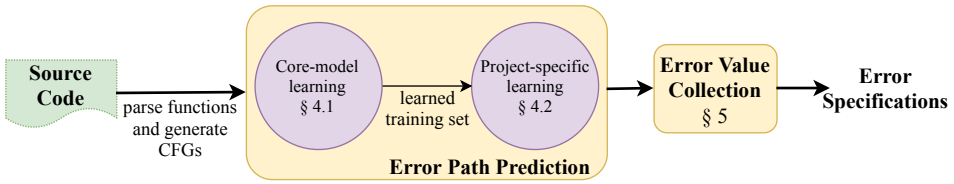


Fig. 3. The workflow of MLPEx. Given a new project, MLPEx first samples its most likely error and non-error paths in source code as the training data (Section 4.2), from which a project-specific model is learned for classifying all paths in the new project (Section 4.3). MLPEx generates the error specification for a function by collecting return values of all its error paths (Section 5).

(2) We infer error specifications based on a set of universal and project-specific features, which are all related to the behaviors of C programs. We statistically validate the usefulness of the features, and show that project-specific features could improve the accuracy of generated error specifications.
(3) We have implemented MLPEx and evaluated it on 10 real-world projects, including Linux, OpenSSL, and httpd. We investigate the result for about 17,750 paths in 3,443 functions. The results show that MLPEx generates correct error specifications for 92% of all functions. Our implementation is available at https://bitbucket.org/plcacs/errorspec/src/master/.
(4) Based on MLPEx, we developed EAB-Miner, a tool for detecting error handling bugs. We detected 57 error handling bugs in 5 real-world projects. The overall precision of detecting bugs is 79%. So far we have reported 8 previously unknown bugs to the development communities.

The rest of the paper is organized as follows. Section 2 shows the overview of MLPEx. Section 3 discusses the features selected to characterize error paths. Sections 4 presents the two learning phases for path classification. Section 5 describes the implementation of MLPEx. Section 6 presents the evaluation results of our approach. To illustrate the usefulness of error specifications, we present an application of detecting error handling bugs in Section 7. We discuss related work in Section 8, talk about threats to validity in Section 9, and conclude in Section 10.

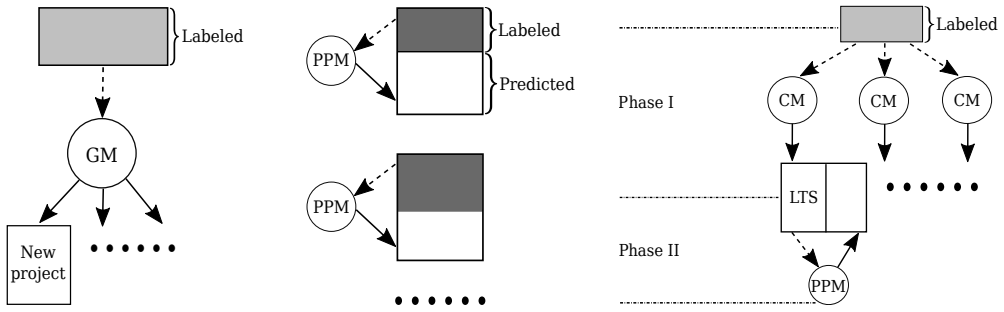
2 OVERVIEW

Before presenting our approach design, we first define paths, which, in turn, relies on the notion of Control Flow Graphs. Both definitions are conventional in the literature.

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}. All the nodes in the CFG are reachable from the n_{begin} and the n_{end} is reachable from all nodes in the CFG [Person et al. 2011].

Definition 2.2. **Path:** A path is a sequence of nodes $\langle n_0, n_1, ...n_j \rangle$ in a CFG, such that there exists an edge $e_{k,k+1} \in E$ between n_k and n_{k+1}, for $k = 0, ..., j-1$ [Nejmeh 1988].

In this paper, a path in a given function `f()` starts from the entry n_{begin} of `f()` and ends at the exit n_{end} of `f()`. Language constructs, such as `for`, `while`, and `goto`, may introduce cycles in a CFG, yielding an infinite number of paths. To avoid this issue, we require that each single path does not contain duplicate nodes. Other work [Jana et al. 2016; Kang et al. 2016; Tian and Ray 2017] faced the same issue and used a similar solution.



(a) Global paradigm (b) Per-project paradigm (c) Two-phase paradigm

Fig. 4. Different learning paradigms. Obtaining the light-grey and dark-grey objects need manual labeling efforts from the researcher and the user, respectively. Each dotted arrow denotes the direction of learning a model from labeled paths, and each solid arrow indicates the direction of predicting the "target" with a model. We use the acronyms "GM", "PPM", "CM" and "LTS" to represent global model, per-project model, core model and learned training set, respectively.

2.1 Approach Design

Since collecting error specifications is easy once we have identified error paths, we focus on error path identification until Section 5, where we discuss error specification collection.

Many recent approaches [Balog et al. 2017; L. Seidel et al. 2017; Pradel and Sen 2018; Wu et al. 2017] have employed supervised learning to solve some programming language problems. To achieve a good learning performance, there are two basic paradigms: learning a global model for different projects or learning a per-project model for individual projects. Each paradigm suffers from some shortcomings. We propose a new paradigm that uses two-phase learning to avert the shortcomings with these basic paradigms.

Learning a global model. As shown in Figure 4a, a global model is trained from a large set of training data, such that it can generalize to new data. We may think of following this idea to develop a model for predicting whether a path is error or non-error. This involves the following steps. (1) We design a set of universal features that can characterize paths across projects. (2) We label each path as error or non-error from a considerable number of projects. (3) We train a model from the labeled paths to relate feature values with labels by using some supervised machine learning method [Hastie et al. 2009]. After that, users could directly apply the trained model to new projects for predicting error paths.

The global model is essentially a data hungry model, and its main drawback is that the size of data needed to train an accurate model is usually up to hundreds of mega bytes, or even giga bytes [Russakovsky et al. 2015]. Obviously, in practice it is usually infeasible to obtain such a large amount of labeled data for training a global model. In the process of solution searching, we trained a global model from 300 labeled paths and observed that its error path prediction accuracy is about 65% only.

Learning a per-project model. One reason that the global model trained from a small set of labeled paths does not perform well is underfitting [Alpaydin 2009]. For example, as shown in Figure 5, the error path in `vlc_sem_post()` is longer than the non-error path, contradicting the intuition that in C the lengths of error paths are usually smaller than those of non-error paths. If the amount of training data is small, such cases are not seen often enough by the global model and thus cannot be handled well. To address this, we can either provide more training data including sufficient "corner" cases, ending up in a global model, or incorporate more features that bring extra

information to learn a more representative model [Hua et al. 2004; Zhang and Ling 2018], leading to this second learning paradigm.

The main idea here is to leverage domain knowledge to extract project-specific features and combine them with universal features to learn a per-project model, as shown in Figure 4b. Project-specific features are closely related to the learning task and are helpful to improve the prediction performance of the trained model [Kopanas et al. 2002]. Take the function `vlc_sem_post()` as an example. It is quite easy for human beings to classify the path returning 0 as a non-error path and the one returning `EINVAL` as an error path. The reason is that the former path ends in the true branch of an if-statement whose condition compares with `KERN_SUCCESS`, a value that usually indicates correct program executions. Similarly, the latter path returns `EINVAL`, a value usually returned by functions to indicate the occurrence of errors. As a result, using that if-statement and the return expression `EINVAL` as two project-specific features can help better identify error and non-error paths in project VLC. For instance, if the return expression of a new path is `EINVAL`, then the new path is likely to be an error path in VLC.

```
1  /* vlc-2.1.4/src/posix/thread.c */
2  int vlc_sem_post(vlc_sem_t *sem)
3  {
4    int val;
5    if (likely(semaphore_signal(*sem) == KERN_SUCCESS))
6      return 0;
7    ...
8    if (unlikely(val != EOVERFLOW))
9      VLC_THREAD_ASSERT("unlocking semaphore");
10   return EINVAL;
11 }
```

Fig. 5. An example of error and non-error paths in VLC. The path returning 0 (marked in bold-green) is a non-error path, while that returning `EINVAL` (marked in italic-red) is an error path.

The shortcoming of a per-project model is that the user needs to manually provide domain knowledge for training a model, and the process has to be repeated for each project because the domain knowledge is project-specific. For example, zlib uses `Z_OK` to indicate non-error situations while VLC never uses such conventions.

Two-phase learning paradigm. The paradigms for training a global model (Figure 4a) and a per-project model (Figure 4b) have their respective advantages, and the advantage of one paradigm is the disadvantage of the other. For example, the advantage of the global paradigm is that the user does not have to label any paths and the disadvantage is that it requires a very large training data set to generate a representative model. In contrast, the per-project paradigm utilizes project-specific features extracted from domain knowledge to train a model from a relatively small training data set, but it requires users to provide the domain knowledge for each new project, which is tedious and hinders its applicability.

Surprisingly, we can integrate the global paradigm and per-project paradigm cogently with the following key observations. (1) Give a small training data set, while the global model fails to accurately classify all paths in a new project, it can do so for a certain portion of paths. We refer to that portion of paths as the learned training set (LTS). LTS contains both error and non-error paths and should be highly accurate. (2) Project-specific domain knowledge can be obtained by extracting relevant information from LTS. For example, if a path in LTS returning `EINVAL` is labeled as an error path, then it is sensible to infer that `EINVAL` represents error paths in the project.

This integration leads to the two-phase paradigm shown in Figure 4c. In phase I, we first learn a core model for each new project based on the knowledge transferred from a set of per-labeled paths. Then, we construct LTS by getting the paths for which the core model is highly confident in its classification. In phase II, we learn a per-project model from LTS and use it to identify error paths in the corresponding new project. The proposed two-phase paradigm provides a nice property: it

ID	Summary	Feature Values
1	Path length	Float (0.0 ~ 1.0]
2	# of expression statements	Float [0.0 ~ 1.0]
3	# of function calls	Float [0.0 ~ 1.0]
4	# of if-conditions	Float [0.0 ~ 1.0]
5	Spanning distance of return expression	Float (0.0 ~ 1.0]
6	Kind of return expression	Nominal [0 ~ 4]
7	Kind of return value	Nominal [0 ~ 3]
8 / 9	Presence of the return expression in set Err_{RetVar} / $NErr_{RetVar}$	Nominal [0 ~ 1]
10 / 11	Presence of the if-conditions in set $Err_{LastCond}$ / $NErr_{LastCond}$	Nominal [0 ~ 1]
12 / 13	Presence of the called functions in set $Err_{LastFunc}$ / $NErr_{LastFunc}$	Nominal [0 ~ 1]

Fig. 6. An overview of path features. The first seven features are universal features, and the rest are project-specific features. We normalize the values of the first five features. A "Nominal" feature takes any integral value specified in the corresponding range.

requires one-time minimal labeling efforts from us to initialize MLPEx and zero manual work from users to predict error paths in different projects. In this paper, we use this paradigm to classify paths into error or non-error. We formalize this paradigm in Section 4.

3 PATH FEATURES

This section presents the features used to classify the paths in C programs. The features are directly extracted from source code and are related to program behaviors. A path can be represented by a path feature vector that stores the feature values. We give an overview of the features and their values in Figure 6. We discuss universal features in Section 3.1 and project-specific features in Section 3.2. We perform statistical analysis to validate the used features in Section 3.3.

3.1 Universal Features

Usually, a function in C checks against different errors before returning the final result of normal execution, and it returns the corresponding error immediately when failing on any such check. To capture this characteristic, we consider the distance between the entry point and the corresponding exit point (feature 1), the number of expression statements (feature 2), the number of function calls (feature 3), the number of if-conditions (feature 4), and the spanning distance with respect to lines of code from the entry point to the corresponding exit point (feature 5).

We normalize the values of the first five features to make them meaningful across different functions and projects. Specifically, for each feature, we divide every non-normalized value by the largest non-normalized value of all paths from the same function. For example, if a path has length 3 and the longest path in the corresponding function is 10, then the normalized value of feature 1 for that path is 0.3.

Moreover, C functions return different kinds of expressions, such as macro and constant, and different kinds of values, such as positive and negative values, to represent the execution results in error or non-error situations. The return expression kind and return value kind could thus indicate if a path is error or non-error. For example, functions tend to use macros with negative values as error returns in C projects. We use feature 6 and 7 to describe the two return kinds.

To compute the values for features 6 and 7, we first perform aggressive substitutions to make the result more accurate. Specifically, we substitute variables with expressions based on the corresponding data flows. For example, in Figure 2, the return variable `tlv` in line 19 is substituted

with `kzalloc(...)`. Based on the substituted return expression, for feature 6, we assign 0 if it is a macro, 1 if it is a constant, 2 if it is a function call, 3 if it is an expression (such as `a+b` where `a` and `b` are two expressions), and 4 if it is a variable. For example, if two paths share the same return statement `return ret` but the first path has the assignment `ret=false` before return and the other has `ret=f(...)`, then the values for feature 6 are 0 and 2 for the two paths, respectively.

For feature 7, we assign 0, 1, and 2 if the substituted return expression can be evaluated to 0, a positive value, and a negative value, respectively, and 3 if the value cannot be statically known. We do not directly encode the return expressions and return values as features. The reason is that, by doing this, the features would have too many possible values, which will hurt machine learning performance according to the bias-variance tradeoff [Geman et al. 1992].

3.2 Project-specific Features

Using project-specific features can better characterize paths since they provide more relevant information about path erroneousness/non-erroneousness. In each path, project-specific information usually appears in three places. The first is the return expression. Error paths within the same project use a set of predefined idioms to represent error code. For example, the functions in httpd return the macros `AP_NOBODY_READ` and `AP_FILTER_ERROR` to indicate errors. We use Err_{RetVar} and $NErr_{RetVar}$ to represent the sets of error and non-error return expressions in a project, respectively.

The second place is the latest if-condition before the path returns since programs usually test with certain conditions to decide whether the current status is still normal or not. For example, in httpd, several functions perform the following test `(VAR_NAME->VAR_NAME)==HTTP_BAD_GATEWAY` (`VAR_NAME` represents any variable name) to indicate that a server received an invalid response and take corresponding actions. We use $Err_{LastCond}$ and $NErr_{LastCond}$ to represent the sets of the latest if-conditions that appear in error and non-error paths, respectively.

The third place is the functions called within the latest conditional before the return statement. The rationale is that functions usually perform similar clean-ups (e.g. release a lock or free some dynamically allocated memory) or loggings before returning from error/non-error paths. This is particularly true for error paths. For example, in httpd, functions `ap_log_rerror` and `ap_abort_on_oom` are used when related errors occur. We use $Err_{LastFunc}$ and $NErr_{LastFunc}$ to represent the sets of such functions in error and non-error paths, respectively.

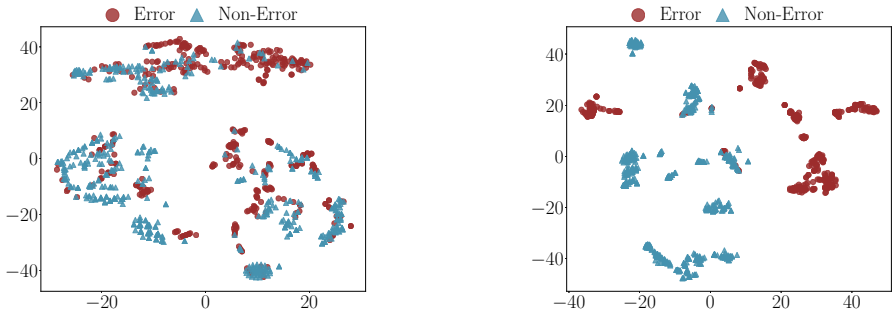
The six sets could be distilled from training data, i.e., a set of labeled paths. From the labeled error paths, extracting Err_{RetVar} and $Err_{LastFunc}$ is straightforward but extracting $Err_{LastCond}$ has some subtlety. In particular, conditions may involve variables, such as `rv == AP_FILTER_ERROR`. Recording the whole condition in the set maybe not very helpful since other places could use different variables to compare with `AP_FILTER_ERROR`. We address this issue by replacing all variables in conditions with `VAR_NAME`. For example, if the training data contains the two error paths in Figure 1, then we add "`Z_STREAM_ERROR`" and "`Z_BUF_ERROR`" to Err_{RetVar}, and add "`deflateStateCheck(VAR_NAME) ==` True" and "`VAR_NAME < ... ==` True" to $Err_{LastCond}$. Similarly, we extract $NErr_{RetVar}$, $NErr_{LastCond}$, and $NErr_{LastFunc}$ from the labeled non-error paths.

It is possible that Err_{RetVar} and $NErr_{RetVar}$ are overlapping, which means that either there exist incorrectly labeled paths in the training data, or some return expressions appear in both labeled error and non-error paths, for example `0` can be used to indicate both error and non-error in Linux kernel. Such return expressions are not associated with error or non-error situations. We thus remove the overlapping elements from both Err_{RetVar} and $NErr_{RetVar}$. Similarly, we do this for $Err_{LastCond}$ and $NErr_{LastCond}$ and for $Err_{LastFunc}$ and $NErr_{LastFunc}$ when overlapping happens.

The project-specific features are then extracted in terms of the presences of the six sets in each path. Specifically, for feature 8, we assign 1 if the return expression of the path appears in Err_{RetVar}, and 0 otherwise. We compute the values for the other project-specific features in the same way.

	ID	p-value			$\mathcal{X}^2$ score		
		libc	httpd	Linux	libc	httpd	Linux
Universal features	1	2.1e-4	1.4e-4	4.6e-2	13.8	14.4	3.2
	2	2.3e-3	2.4e-3	3.1e-2	9.3	9.2	4.7
	3	2.1e-9	2.8e-7	2.8e-3	35.9	26.3	8.9
	4	1.4e-7	4.2e-4	6.2e-4	27.7	12.5	11.7
	5	7.9e-6	1.6e-5	3.1e-2	20.0	18.6	4.7
	6	4.4e-5	2.7e-5	9.5e-4	14.7	15.0	5.2
	7	1.4e-7	4.0e-6	8.7e-5	27.6	36.6	15.8
Project-specific features	8	2.7e-16	5.8e-25	1.1e-18	67.0	106.4	77.85
	9	3.2e-11	4.0e-14	5.0e-6	34.5	57.2	20.8
	10	6.7e-7	2.2e-17	2.5e-5	24.7	71.9	17.7
	11	8.2e-32	7.1e-33	2.0e-15	137.8	142.6	63.1
	12	8.7e-85	2.5e-109	5.7e-79	380.7	493.5	354.3
	13	7.8e-56	1.4e-66	1.5e-49	247.8	297.1	219.0

Fig. 7. Results of Chi-square analysis. For all three projects, the p-value of each feature is smaller than 0.05. Moreover, the $\mathcal{X}^2$ scores of project-specific features are in general higher than those of universal features.



(a) With only universal features.

(b) With universal and project-specific features.

Fig. 8. t-SNE visualization of path distributions in httpd with and without project-specific features. The error and non-error paths are easily separated after considering project-specific features.

3.3 Feature Validation

We perform Chi-square analysis [Liu and Setiono 1995] to investigate the usefulness of path features, which is a standard method of feature validation [Li et al. 2018]. Specifically, we randomly selected 300 functions from libc, Linux and httpd, which are different from the evaluated functions in Section 6. We manually labeled all 1604 paths generated from these functions. For each project, 100 randomly selected paths were used to extract the project-specific information (i.e., the six sets such as $Err_{LastFunc}$ and $NErr_{LastFunc}$), based on which we extracted project-specific features. The results of Chi-square analysis are presented in Figure 7, which contains two parts: p-value and $\mathcal{X}^2$ score. The smaller the p-value, the higher the significance that the feature is associated with path label. In addition, The higher the $\mathcal{X}^2$ score is, the more important the feature is for path classification [Li et al. 2018]. Based on the results, we aim to answer the following two questions:

Q1. Are all features relevant to the path classification?

From Figure 7, we find that, for libc, the p-values of all features are smaller than 0.05, showing high-level significance when associated with path label. We observe similar results for httpd and Linux. Thus, the features used in our work are statistically correlated with error/non-error paths.

Q2. Can project-specific features help distinguish error paths from non-error paths?

From Figure 7, we also find that the $\mathcal{X}^2$ scores of project-specific features are higher than those of universal features across all projects. For example, the average scores of universal and project-specific features in httpd are 18.9 and 194.8, respectively. Therefore, considering project-specific features can effectively enhance the performance of error path prediction in machine learning. To illustrate this, we visualize the spatial distribution of paths in httpd (using t-SNE [Maaten and Hinton 2008]), and then compare the path distributions with and without project-specific features in Figure 8. It is clear to see that error and non-error paths become much more separable after considering project-specific features. We make the same observations on libc and Linux. The results suggest that a better path classifier could be learned from the feature space containing both universal and project-specific features.

4 TWO-PHASE LEARNING

In general, the goal of supervised learning is to learn a model from a given set of labeled instances (training set) and then use the learned model to make predictions for new instances. As discussed in Section 2.1, there are two issues when applying traditional learning methods on path prediction. First, to make sure that the learned model generalizes well to new projects, onerous manual efforts are required to label a large size of paths such that the training set is representative enough. Second, comparing to universal features, project-specific features can improve the prediction accuracy. However, the extracted project-specific features vary across projects and thus can not be used as universal features, i.e., we can not directly apply the project-specific features of one project to another. The dilemma between choosing universal and project-specific features limits the applicability of machine learning. To address these issues, we proposed a novel two-phase learning paradigm in Section 2.1.

The workflow of two-phase learning is as follows. In phase I, we learn a core model with universal features and use it to automatically generate a learned training set for a new project. In phase II, we extract project-specific features using the learned training set from phase I, learn a per-project model, and apply it to the new project to perform path predictions. Each learning phase itself is a complete learning process, and each phase consists of two steps: training and predicting. The structures of Sections 4.2 and 4.3 that present the two learning phases reflect this fact, after we present the preliminaries of machine learning in Section 4.1. To improve the readability of this section, we summarize the notations used in this section in Figure 9.

4.1 Preliminaries of Machine Learning

As mentioned earlier, a supervised learning process consists of both training and predicting. The goal of the training step is to learn a probability description that can best approximate the distribution of the given training dataset. Concretely, let $(\mathbf{X}, \mathbf{Y}) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ denote a set of labeled paths $\mathcal{D}$, where $x_i \in \mathbb{R}^d$ is a vector containing d features as the representation of path i, and the class label $y_i \in \{-1, +1\}$ indicates if path i is error or non-error, respectively. To learn the probability distribution of $\mathcal{D}$, a common way is to use Maximum Likelihood Estimation (MLE) [Vapnik 2013], expressed as follows:

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{(x,y)\in\mathcal{D}}[\ln \mathcal{L}(x, y, \theta)], \tag{1}$$

Notations	Interpretations
$\mathbb{R}^d$	Real coordinate space of d dimensions
x	The feature vector of a path
y	The label of a path
(x, y)	A path whose feature vector is x and label is y
$\mathrm{P}(y\|x, \theta)$	The probability that the label of a path is y when the feature vector is x, and θ is the parameter of the probability function P.
$\mathbf{X}$	A set of path feature vectors
$\mathbf{Y}$	A set of path labels
$\mathcal{D}_{\text{preL}}$ / $\mathcal{D}_{\text{new}}$	The set of paths in the pre-labeled / unlabeled project
$\mathrm{P}_{\text{preL}}(\mathbf{X}, \mathbf{Y})$ / $\mathrm{P}_{\text{new}}(\mathbf{X}, \mathbf{Y})$	The distribution of the paths in $\mathcal{D}_{\text{preL}}$ / $\mathcal{D}_{\text{new}}$
$\mathcal{D}_{\text{LTS}}$	The learned training set
θ^*_{preL} / θ^*_{core} / θ^*_{perP}	The model learned from $\mathcal{D}_{\text{preL}}$ / $\mathcal{D}_{\text{preL}}$ and $\mathcal{D}_{\text{new}}$ / $\mathcal{D}_{\text{LTS}}$
γ	The prediction confidence threshold

Fig. 9. Summary of the notations used in Section 4.

where $\mathcal{L}(x, y, \theta)$ is the likelihood function in MLE that depends on its parameter $\theta \in \mathbb{R}^d$. In machine learning, the conventional choice of a likelihood function could be an entropy function [Nigam et al. 1999], the gini index [Breiman 2017], or some form of regression function. Equation (1) aims to find an assignment to the parameter θ that maximizes (expressed by arg max) the expected likelihood of the paths in $\mathcal{D}$ (expressed by $\mathbb{E}$). The best assignment to θ is denoted by θ^*.

In this paper, we let $\mathcal{L}(x, y, \theta)$ be the standard logistic regression [Kleinbaum et al. 2002]:

$$\mathcal{L}(x, y, \theta) = \frac{1}{1 + \exp(-y\theta^\top x)} = \mathrm{P}(y|x, \theta), \tag{2}$$

which computes the probability that a path x has the label y.

Plugging equation (2) into equation (1), the task of the training process is to determine the optimal parameter θ^* by solving equation (1) with some gradient based method (more in Section 5). We refer to the obtained θ^* as the learned model, under which the given paths in $\mathcal{D}$ is most probable. With the model θ^*, we can predict if a new path $\hat{x}$ is error or non-error by:

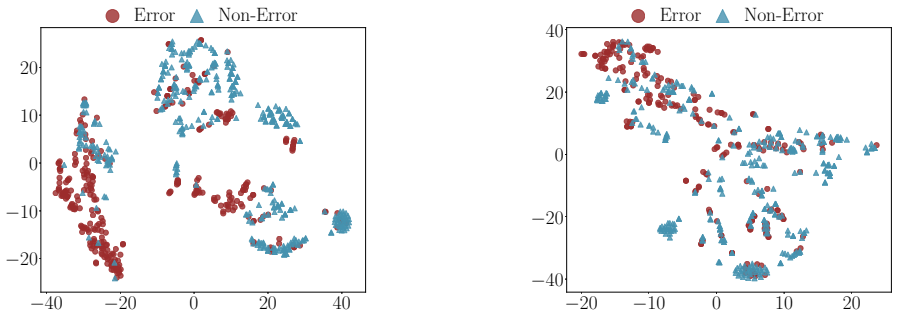
$$\hat{y} = \underset{y \in \{-1, +1\}}{\arg\max}\ \mathrm{P}(y|\hat{x}, \theta^*), \tag{3}$$

where we use the definition in equation (2) for $\mathrm{P}(y|x, \theta)$ in this equation (and all other equations in the rest of this section).

We can follow the spirit of the paradigm in Figure 4a or 4b to train a model for path prediction. However, both solutions require expensive manual labeling efforts, rendering them infeasible. In this work, we exploit the recent advances in transfer learning [Pan and Yang 2010], which help us utilize the knowledge of a pre-labeled project to automatically identify a set of error and non-error paths as the training data for a new project.

4.2 Core-model Learning (Phase I)

The goal of phase I learning is to construct the learned training set (LTS) for a new project without manually labeling paths. To this end, we need to learn a model for the new project and use it to select paths that are most likely to be error or non-error. The naive way to obtain such a model is that we train a model from a pre-labeled project $\mathcal{D}_{\text{preL}}$. We use $\mathrm{P}_{\text{preL}}(\mathbf{X}, \mathbf{Y})$ to denote the path



(a) libc (b) Linux

Fig. 10. t-SNE visualization of path distributions based on universal features in libc and Linux. The path distribution of libc is quite different from that of Linux.

```
/* libc-2.26/malloc/arena.c */
static int shrink_heap (...)
{
  long new_size;

  new_size = (long) h->size - diff;
  if (new_size < (long) sizeof(*h))
    return -1;
  ...
}
```

```
/* Linux-4.4.0/open.c */
static long do_sys_truncate (...)
{
  unsigned int lookup_flags = LOOKUP_FOLLOW;
  struct path path;
  int error;

  if (length < 0)
    return -EINVAL;
  ...
}
```

Fig. 11. An example that paths from two different projects share similar characteristics, such as the path length, the number of if-conditions, and the kind of return value. Functions `shrink_heap()` and `do_sys_truncate()` return -1 and `-EINVAL` to represent errors, respectively.

distribution of $\mathcal{D}_{\text{preL}}$. In Phase I, only universal features are used, so $x \in \mathbb{R}^{d_u}$ where d_u is the number of universal features. The model learned from $\mathcal{D}_{\text{preL}}$ then is:

$$\begin{aligned}\theta^*_{\text{preL}} &= \arg\max_{\theta} \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{preL}}}[\ln \mathrm{P}(y|x,\theta)] \\ &= \arg\max_{\theta} \sum_{x,y} \mathrm{P}_{\text{preL}}(x,y) \ln \mathrm{P}(y|x,\theta). \end{aligned} \tag{4}$$

Unfortunately, the obtained θ^*_{preL} may not generalize to the new project, and thus can not be used to construct LTS. For example, we trained θ^*_{preL} from 473 error and non-error paths in libc and used it (replacing θ^* in equation (3) with θ^*_{preL}) to classify paths in Linux. The observed accuracy is only 38.9%, indicating we can not use θ^*_{preL} to determine if a path is error or non-error. The reason is that the path distributions of libc and Linux are quite different, as shown in Figure 10. As a matter of fact, θ^*_{preL} can generalize only to projects that have similar underlying path distributions, which is the assumption held by most machine learning approaches [Murphy 2012].

4.2.1 Learn a Core Model. Although directly applying θ^*_{preL} to perform path predictions in a new project does not work, it is possible that we can utilize some knowledge from the pre-labeled project to help label the paths in the new project. To illustrate, consider the functions `shrink_heap()` and `do_sys_truncate()` in Figure 11. The two functions are syntactically quite different. For example,

they differ in variable names, called functions, and if-conditions. However, some of their paths are quite similar in terms of universal features. For instance, the universal feature vectors (Section 3.1) for the error paths in `shrink_heap()` and `do_sys_truncate()` are $[0.44, 0.25, 0, 0.33, 0.28, 1, 2]$ and $[0.50, 0.25, 0, 0.33, 0.33, 0, 2]$, respectively. We observe that for each single feature, the feature values in two feature vectors are very close. This indicates that the two paths have a similar structure, such as the path lengths are relatively small, the numbers of statements before `return` are also small, and the return expressions are both negative constants.

The above observation suggests that there exist some paths that have similar structures in both the pre-labeled and new project, i.e., the path distributions of the two projects are overlapped. If we learn a model based on the overlapped path distribution, then the learned model can be generalized to the new project. This idea is known as reweighting in transfer learning, which means that we assign higher weights to the overlapped paths in the pre-labeled project for training a model that can be used in the new project. In particular, we reweight the likelihood of each path in $\mathcal{D}_{\text{preL}}$ by:

$$\theta^*_{\text{core}} = \arg\max_{\theta} \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{preL}}}[\beta(x,y)\ln \text{P}(y|x,\theta)], \tag{5}$$

where $\beta(x,y) = \text{P}_{\text{new}}(x,y)/\text{P}_{\text{preL}}(x,y)$, and $\text{P}_{\text{new}}(x,y)$ denotes the path distribution of the new project. We will discuss how to determine the reweighting factor $\beta(x,y)$ later.

We call θ^*_{core} a core model and have the following property of the obtained core model.

Theorem 4.1. *The core model θ^*_{core} learned from the reweighted $\mathcal{D}_{preL}$ can be generalized to the new project.*

Due to the space restriction, we provide the proof of the theorem in the appendix of the longer version of this paper [Wu et al. 2019a]. Theorem 4.1 shows that if we change the path distribution of $\mathcal{D}_{\text{preL}}$ from $\text{P}_{\text{preL}}(\mathbf{X},\mathbf{Y})$ to $\beta(\mathbf{X},\mathbf{Y})\text{P}_{\text{preL}}(\mathbf{X},\mathbf{Y})$, we will obtain the core model with good generalizability.

The question is how to compute $\beta(\mathbf{X},\mathbf{Y})$? Note that our application scenario is similar to sample selection bias [Zadrozny 2004] and covariate shift [Shimodaira 2000], which tackle the problem of training and testing data that are drawn from different distributions. For example, in our case, we can consider the paths in libc as the training data and those in Linux as the testing data. Therefore, we follow the same way used in sample selection bias and covariate shift to approximate $\beta(\mathbf{X},\mathbf{Y})$ via $\text{P}_{\text{new}}(\mathbf{X})/\text{P}_{\text{preL}}(\mathbf{X})$, where $\text{P}_{\text{new}}(\mathbf{X})$ and $\text{P}_{\text{preL}}(\mathbf{X})$ are the marginal distributions of paths in the new project and the pre-labeled project, respectively. Accordingly, we obtain θ^*_{core} using $\mathcal{D}_{\text{preL}}$ by:

$$\theta^*_{\text{core}} = \arg\max_{\theta} \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{preL}}}[\frac{\text{P}_{\text{new}}(x)}{\text{P}_{\text{preL}}(x)}\ln \text{P}(y|x,\theta)]. \tag{6}$$

The intuition behind equation (6) is that, given a path x from $\mathcal{D}_{\text{preL}}$, a higher weight is assigned to x if it is over-represented in $\mathcal{D}_{\text{new}}$, whereas a lower weight is assigned if x is under-represented in $\mathcal{D}_{\text{new}}$. For example, if a path in libc has a similar structure as some path in Linux, then we should focus more on this path with a large weight when training θ^*_{core}. On the other hand, if the structure of a path in libc is unlikely to exist in Linux, then we should ignore this path during training. There are various ways to estimate $\text{P}_{\text{new}}(x)/\text{P}_{\text{preL}}(x)$ [Huang et al. 2007; Sugiyama et al. 2008; Zadrozny 2004]. In the implementation, we apply the method proposed by Huang et al. [2007] since it can work well even when the sizes of $\mathcal{D}_{\text{preL}}$ and $\mathcal{D}_{\text{new}}$ are small.

We reuse the data in Section 3.3 to verify the effectiveness of learning θ^*_{core}. Specifically, we consider the labeled paths of libc as $\mathcal{D}_{\text{preL}}$ based on which the core model θ^*_{core} for Linux could be obtained. We also train θ^*_{preL} from libc and compare it with θ^*_{core}. We find that θ^*_{core} yields better prediction accuracy than θ^*_{preL} does on Linux. For example, the accuracy is 38.9% when directly

applying θ^*_{preL} to Linux, while it increases to 73.6% when using θ^*_{core}. Thus, with the knowledge transferred from $\mathcal{D}_{\text{preL}}$, we can effectively learn θ^*_{core} on different projects.

4.2.2 Construct LTS. The obtained θ^*_{core} generalizes to the new project, but it may not yet achieve very high prediction accuracy. For example, the accuracy on Linux is 73.6% only. This is because either the estimation of $\beta(x, y)$ is not 100% correct or $\mathcal{D}_{\text{preL}}$ is not representative enough, or both. More concretely, since the structures of some error/non-error paths in the new project could be unseen in $\mathcal{D}_{\text{preL}}$, θ^*_{core} may make wrong predictions on those paths. Nevertheless, we can use θ^*_{core} to sample a set of error and non-error paths with high confidence as the learned training set $\mathcal{D}_{\text{LTS}}$.

Let $\mathcal{D}_{\text{new}}$ denote the paths in the new project. The core model θ^*_{core} computes the probability that $x \in \mathcal{D}_{\text{new}}$ is an error path by $\text{P}(y = -1|x, \theta^*_{\text{core}})$, or a non-error path by $\text{P}(y = 1|x, \theta^*_{\text{core}})$. The larger the probability, the more accurate the prediction made by θ^*_{core}. Therefore, we choose paths for which θ^*_{core} predicts with confidence higher than a threshold γ as $\mathcal{D}_{\text{LTS}}$. In the implementation we set γ to 0.85, a relatively high value, such that $\mathcal{D}_{\text{LTS}}$ is accurate[2].

$$\mathcal{D}_{\text{LTS}} = \{(x, y)|x \in \mathcal{D}_{\text{new}} \wedge \exists y \in \{-1, 1\} : \text{P}(y|x, \theta^*_{\text{core}}) \geq \gamma\}. \tag{7}$$

We want to emphasize that, at the first glance, one may think of using θ^*_{preL} instead of θ^*_{core} to learn the training set $\mathcal{D}_{\text{LTS}}$, but it will not work in practice. This is because correctly predicting a path in $\mathcal{D}_{\text{new}}$ as error or non-error is related to the generalizability of the model θ^*. If the relationship between paths $\mathbf{X}$ and labels $\mathbf{Y}$ in $\mathcal{D}_{\text{new}}$ is inappropriately captured by θ^*, then the computed probability could not represent the confidence in terms of the ground truth of the predicted path. For example, we applied θ^*_{preL}, which is learned from libc, and θ^*_{core}, which is learned from the knowledge transferred from libc, on Linux to construct different $\mathcal{D}_{\text{LTS}}$ and compared the training set accuracy in Figure 12. We observe that, as the value of γ increases from 0.5 to 0.95, the accuracy yielded by θ^*_{core} monotonically increases from 0.71 to 0.96, while the accuracy yielded by θ^*_{preL} changes arbitrarily. This means that, to learn an accurate training set, we need to use the core model θ^*_{core} that generalizes to $\mathcal{D}_{\text{new}}$.

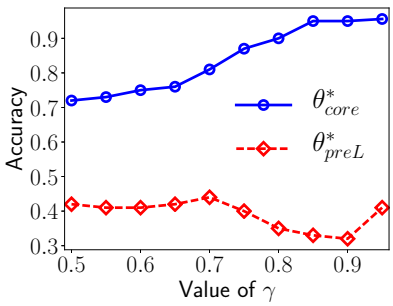


Fig. 12. LTS accuracy comparison.

4.3 Project-specific Learning (Phase II)

Once the learned training set is obtained, we can extract d_p project-specific features of the new project, as described in Section 3.2. These features are domain-specific knowledge, and thus they can be used to represent the structures of error/non-error paths in a finer granularity. Such finer representations are helpful to distinguish error paths from non-error paths. Therefore, in this phase, we use universal and project-specific features together to learn a project-specific model. Then, we apply the project-specific model on the new project to classify paths as error or non-error.

4.3.1 Learn a Per-Project Model. In Phase II, we use project-specific features together with universal features, and thus $x \in \mathbb{R}^{d_u+d_p}$ for any path x in $\mathcal{D}_{\text{new}}$. Similar to the way of training a model in supervised learning, the per-project model θ^*_{perP} trained from $\mathcal{D}_{\text{LTS}}$ is:

$$\theta^*_{\text{perP}} = \arg\max_{\theta} \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{LTS}}}[\ln \text{P}(y|x, \theta)]. \tag{8}$$

[2] We define the accuracy of the training set as the ratio between correctly sampled paths and all sampled paths.

Algorithm 1: Two-phase Learning

Initialize: Pre-labeled project $\mathcal{D}_{\text{preL}}$
Input: New project $\mathcal{D}_{\text{new}}$
Output: Path prediction $\hat{\mathbf{Y}}$ for $\mathcal{D}_{\text{new}}$

1 $\hat{\mathbf{Y}} \leftarrow \emptyset$;
// Phase I
2 Estimate $\frac{\mathrm{P}_{\text{new}}(x)}{\mathrm{P}_{\text{preL}}(x)}$ for each path x in $\mathcal{D}_{\text{preL}} \cup \mathcal{D}_{\text{new}}$;
3 Train the core model θ^*_{core} using equation (6);
4 Construct the learned training set $\mathcal{D}_{\text{LTS}}$ using equation(7);
// Phase II
5 Extract project-specific features from the paths in $\mathcal{D}_{\text{LTS}}$;
6 Train the per-project model θ^*_{perP} using equation(8);
7 **foreach** $x \in \mathcal{D}_{new}$ **do**
8 $\quad \hat{\mathbf{Y}} \leftarrow \hat{\mathbf{Y}} \cup \{(x, \arg\max_{y \in \{-1,+1\}} \mathrm{P}(y|x, \theta^*_{\text{perP}}))\}$;
9 **return** $\hat{\mathbf{Y}}$;

Alternatively, we could use other traditional machine learning methods like random forests [Breiman 2001] to learn θ^*_{perP}, which might further improve the classification accuracy. We do not consider, however, deep learning algorithms using neural networks here for two reasons. First, they require a large amount of training data while the size of $\mathcal{D}_{\text{LTS}}$ could be relatively small. Second, the training process of neural networks usually needs to manually tune the hyper-parameters, hindering us from automating the error specification generation. To keep our two-phase learning algorithm simple and efficient, we obtain θ^*_{perP} by solving equation (8).

4.3.2 Classify Paths. Having obtained the per-project model θ^*_{perP}, we use it to classify the paths in $\mathcal{D}_{\text{new}}$ into error or non-error according to equation (3). It is possible that in the classification result all paths in one function belong to the same class. This does not necessarily mean that the path prediction is wrong since in practice some functions are infallible. For example, the function `shmem_getattr()` in Linux memory management module will never fail to get the shared memory state. In our evaluation, we find that about 8% of the functions are infallible.

Time complexity of two-phase learning. We present the details of two-phase learning in Algorithm 1. The time complexity of each step of the algorithm is summarized as follows. Step 2 uses all the paths in $\mathcal{D}_{\text{preL}}$ and $\mathcal{D}_{\text{new}}$, yielding a time complexity of $O(|\mathcal{D}_{\text{preL}} \cup \mathcal{D}_{\text{new}}|^2)$. Equations (6) and (8) in steps 3 and 6 can be solved by using SGD [Robbins and Monro 1951], which take $O(|\mathcal{D}_{\text{preL}}|)$ and $O(|\mathcal{D}_{\text{LTS}}|)$, respectively. Steps 4, 5 and 7 have the same time complexity, and all can be finished in $O(|\mathcal{D}_{\text{new}}|)$. The overall time complexity is thus $O(|\mathcal{D}_{\text{preL}} \cup \mathcal{D}_{\text{new}}|^2 + |\mathcal{D}_{\text{preL}}| + |\mathcal{D}_{\text{LTS}}| + 3|\mathcal{D}_{\text{new}}|) = O(|\mathcal{D}_{\text{preL}} \cup \mathcal{D}_{\text{new}}|^2)$.

5 IMPLEMENTATION

Our implementation mainly consists of extracting paths from source code, implementing the two-phase learning (Section 4), labeling initial paths, and collecting error specifications. Overall, MLPEx is implemented in 2.2K lines of Python code.

Extracting paths itself includes several steps. First, we apply the default project configuration to preprocess the code base of each project with GCC using options "-E -fdirectives-only". At this

step, GCC will handle the directives such as `#ifdef` and `#define` by including appropriate code regions (enclosed by, for example, `#ifdef` and `#endif`) into the preprocessed code. It does not expand macros, allowing us to obtain information about macros during feature extraction. Next, we apply Joern [Yamaguchi et al. 2014] to build the control-flow and data-dependency graphs of functions, and store them in the graph database Neo4j [neo 2019]. After that, we query the Neo4j database to extract paths without duplicate nodes. Finally, we generate path features and store them in MongoDB [mon 2019].

To solve the optimization problems in two-phase learning, we adopt SGD solver from the scikit-learn Python package [Pedregosa et al. 2011]. We initialize MLPEx by randomly selecting 100 functions from GNU Tar project (as the pre-labeled project $\mathcal{D}_{\text{preL}}$). There are in total 513 paths of these functions. We manually label all the paths, and store the information including path features and the corresponding labels in MongoDB.

Given the error paths and non-error paths for a function, its error specification consists of two parts: the set of return values of error paths and the union of error specifications of functions returned in non-error paths. For example, if function `f()` returns `g()` in a non-error path, then the error specification of `g()` becomes part of that of `f()`. For each function, we first construct a transitive closure to capture all its dependencies by using static analysis of the given code base and system configuration information [Arnold 1996]. Then, based on the data dependence graph generated by Joern, we collect the error specification by implementing a method similar to constant folding [Muchnick 1997] for evaluating the error return values of each path. Note that, in principle, statically determining function return values is undecidable. However, the error values are very often literals, and thus our approach can effectively collect error specifications for most realistic programs.

When the return expression contains function calls, error specification collection does not work well in two cases. (1) The source code of the called function is unavailable, such as a third-party API that exposes only its type. We could address this case by obtaining the error specifications for these functions from other sources, such as API documentations. (2) In each particular context, it might be possible that only a subset of the error specification of the callee will become that of the caller while our implementation includes the full set as the error values are collected from all error paths. Nevertheless, we feel this treatment is acceptable for two reasons. First, our result may be over approximating but never misses error codes. Second, even over-approximated, the error specifications from MLPEx are small. For example, we have measured their sizes (the number of values in a specification), and we find that they are single values in 66% of cases and have more than two values in only 10% of cases.

While error values are often returned directly from functions in C code, they can be returned in more complex ways, such as assigned to some struct field or a parameter pointer. Our approach, as well as other work on error specification generation such as APEx [Kang et al. 2016], currently do not handle such cases. To address this problem, we can reuse our work on classifying paths, and extend the step of collecting error specifications to identify the assignments to fields or parameter pointers that store error values. However, there is no clear solution to identify such assignments. One possible idea is still using machine learning based on features such as how often or when the fields or parameter pointers are assigned. We leave this investigation to future work.

6 EVALUATION

We use 10 projects (6 libraries and 4 applications) to evaluate MLPEx. We provide the details of these study subjects in Section 6.1. In Section 6.2, we show how the ground truths of error specifications are obtained. In Section 6.3, we present the results of automatically learned training sets for the evaluated projects. In Section 6.4, we show the results of error path prediction of MLPEx.

		Funcs	Paths	Path len*	Func calls*	Err range	NErr range
Libs	zlib	159	1001	30.40 (***36.84***)	2.27 (***2.53***)	$e \leq 0$	$e \geq 0$
	Libgcrypt	500	1740	21.37 (***13.73***)	2.68 (***2.28***)	e = -1 or $e \geq 0$	$e \geq 0$
	OpenSSL	129	824	25.70 (***15.37***)	6.97 (***4.74***)	$e \leq 0$	$e > 0$
	GTK+	141	429	15.49 (***16.83***)	3.72 (***3.19***)	$e < 0$	$e \geq 0$
	libc	500	2432	45.82 (***30.25***)	9.14 (***5.11***)	e = -1 or $e > 0$	$e \geq 0$
	GnuTLS	260	1352	14.43 (***9.59***)	3.96 (***3.36***)	$e \leq 0$	$e \geq 0$
Apps	Linux	1000	6245	30.16 (***20.15***)	6.89 (***7.52***)	$e \leq 0$	$e \geq 0$
	httpd	500	2421	36.04 (***31.56***)	7.12 (***9.41***)	e = -1 or $e > 0$	$e \geq 0$
	VLC	102	572	25.96 (***16.98***)	7.51 (***5.13***)	$e \leq 0$	$e \geq 0$
	wget	154	734	22.54 (***12.36***)	4.09 (***3.47***)	$e < 0$	$e \geq 0$
Overall		3,445	17,750	29.20 (***25.73***)	6.16 (***5.94***)	All values	$e \geq 0$

Fig. 13. Study subjects. We randomly selected functions from each study subject. For each function, we evaluated all its paths. The values in "Funcs" and "Paths" columns represent the number of evaluated functions and paths in each project, respectively. The columns marked with an asterisk include average value and the corresponding standard deviation (in bold italic). We use the short names Err and NErr for Error and Non-Error, respectively.

In Section 6.5, we compare the error specification results of MLPEx with APEx, a state-of-the-art error specification generation approach. In Section 6.6, we discuss the usefulness of project-specific features for error path prediction. In Section 6.7, we evaluate the runtime aspect of MLPEx.

6.1 Study Subjects

We give the details of the 10 study subjects in Figure 13. For each project, we randomly selected functions and evaluated all the paths of each function. We manually investigated the evaluation results to determine the correctness of path prediction and error specification.

Libraries include zlib (v1.2.11), Libgcrypt (v1.8.1), OpenSSL (v1.1.1), GTK+ (v3.22), libc (v2.26), and GnuTLS (v3.5). These libraries expose popular functionalities that are commonly used by other projects. As discussed in Section 1.2, existing approaches like APEx can generate error specifications for APIs, but fail to work for internal functions that implement underlying facilities since internal functions have a limited number of call sites. MLPEx does not suffer from this issue, and our evaluation considers both APIs and internal functions.

Applications include Linux (v4.4.0), httpd (v2.4.29), VLC (v2.1.4), and wget (v1.19). They are all widely used in reality. Although the functions in applications are usually internal, MLPEx is able to infer error specifications for them based on their source code.

Our study subjects are representative since the average path length and the number of function calls in each path vary significantly across projects. For example, on average, the path length in GnuTLS is 14.43, while that in libc is 45.82. This indicates that projects have very different code structures. Moreover, within a single project, the standard deviations of path lengths and function calls are large, meaning that paths in individual project have diverse structures.

The last two columns of Figure 13 list the error range and non-error range for each project. From them, we make the following observations. First, error ranges vary across projects. For example, while zlib uses ≤ 0 to represent error ranges, httpd mostly uses values > 0 to denote errors.

Second, in three projects, the error range and non-error range are overlapping. For example, in httpd, the function `ap_parse_from_data` returns `HTTP_BAD_REQUEST`, which is defined to be 400, as an error value. In contrast, the function `ftp_set_TYPE` returns `HTTP_OK`, a positive value indicating

the ftp command is correct, as a non-error value. As a result, positive values can represent both errors and non-errors in this application. In addition, in five projects, such as zlib, 0 is used to indicate both error and non-error. Such counter intuitions suggest that generating precise error specifications for our study subjects is nontrivial.

6.2 Obtaining Ground Truths

In our evaluation, we need the ground truths for the predicted paths and the generated error specifications. The most important information for obtaining the ground truths is function error codes. Once we have the complete error codes, it is straightforward to determine the error paths and the error specification for each function based on the return values. For example, recalling the function `deflatePrime()` given in Figure 1, there are three paths with three corresponding return values. Assume that we already know that zlib uses `Z_STREAM_ERROR` and `Z_BUF_ERROR` to represent errors and uses `Z_OK` to represent success. The paths returning `Z_STREAM_ERROR` and `Z_BUF_ERROR` are thus the error paths. Given that the values of `Z_STREAM_ERROR` and `Z_BUF_ERROR` are -2 and -5, respectively, the error specification for `deflatePrime()` is $\{-2, -5\}$.

We present our methods for finding the error codes of a project as follows.

From official documentation and online materials. For most large C projects like Linux and httpd, they have well-documented materials for users to read and understand the code. For example, httpd has an official website[3] providing the description of each HTTP status code. Based on such descriptions, it is very easy to determine which status codes represent error situations in httpd. However, there are two shortcomings if we only rely on those documentations. First, since most documentations only provide information for APIs, the error codes for internal functions cannot be obtained. Second, the error codes for APIs may be incomplete if the documentations is out-dated.

From header files. Another place that we can directly get the error codes is header files. In C projects, the error codes are usually defined as macros and put together in header files for the purpose of software maintenance. For example, in Linux, most error codes are defined in errno-base.h and errno.h. As a result, we can easily obtain all error codes except 0 used in Linux. However, for some projects like VLC, they use different error codes for different modules. This may cause the header files containing error codes scatter into the subfolder of each modules, making locating such header files become difficult.

Based on naming convention. Individual C projects prefer their own coding style. As suggested by the Linux kernel development guide[4], the errors returned by functions should be either 0 or -Exxx. For example, -EIO represents I/O errors. Based on such naming convention, we can infer error codes if the return values are written in -Exxx. The disadvantage of this method is that it requires coding knowledge of the project and the naming convention varies from different projects.

```
1  /* Linux-4.7/amd64-agp.c */
2  int __init apg_amd64_init(void)
3  {
4    ...
5    if (!pci_dev_present(amd_nb_misc_ids)) {
6      pci_unregister_driver(&agp_amd64_pci_driver);
7      return -ENODEV;
8    }
9    ...
10 }
```

Fig. 14. An example of using call site to determine error code. Function `apg_amd64_init()`, the call site of `pci_dev_present()`, returns `-ENODEV` on error (marked in italic-red).

[3]https://ci.apache.org/projects/httpd/trunk/doxygen/group__HTTP__Status.html
[4]https://www.kernel.org/doc/html/v4.10/process/coding-style.html#function-return-values-and-names

	zlib	httpd	VLC	SSL	wget	GTK+	libc	Linux	TLS	gcrypt
Precision	0.957	0.920	0.965	0.955	0.952	0.953	0.950	0.944	0.942	0.958

Fig. 15. Accuracy of the learned training set for each project. The overall precision is 94.6%. To save space, we use the short names SSL, TLS, and gcrypt for OpenSSL, GnuTLS, and Libgcrypt, respectively.

From call sites. Although the first three methods combined look effective to find most error codes, we may still fail on some corner cases. For instance, Linux uses 0 to indicate both error and non-error, and it is thus not clear if 0 should be included in the error specification for a function. In such cases, we inspect the call sites of the function. This is because the call site needs to check against the error returns of the callee and performs corresponding error handling. Consider the example shown in Figure 14. The caller `apg_amd64_init()` returns `-ENODEV` to indicate error when the return value of callee `pci_dev_present()` is 0. Therefore, we know that 0 is one error code in `pci_dev_present()`.

In our experience, the above four methods help determine the error codes for more than 90% of functions, but they may fail to cover some internal functions with few call sites. For those cases, we read the relevant source code to determine the error codes. We discuss the threats to validity in Section 9.

6.3 Learned Training Set Accuracy (First Learning Phase)

An obstacle of applying supervised machine learning is that manually building a training set is time-consuming and error-prone [Sheng et al. 2008]. To address this issue, MLPEx automatically learns the training set by sampling a set of error and non-error paths. To measure the accuracy of the learned training set, we use the sampling precision as the metric, which is a division of the number of correctly sampled error and non-error paths over all sampled paths.

As Figure 15 shows, the precision of the learned training set for each project is higher than 90%. In addition, the average precision of the paths sampled by MLPEx is high, about 94%, meaning that the quality of the learned training set is good. Although there are about 6% of paths in the learned training set are labeled incorrectly, they can be considered as noise, which could be absorbed in phase II when learning the per-project model for error path prediction.

To check if initializing MLPEx with different pre-labeled paths affects the precision of learned training sets, we have repeated the evaluation process by labeling paths from other projects. We observe that the results are stable across different sets of pre-labeled paths. For example, the average precision is always higher than 90%. This supports Theorem 4.1 that the core model obtained in phase I can generalize to different projects, for the purpose of constructing the learned training set.

6.4 Error Path Prediction Accuracy (Second Learning Phase)

MLPEx predicts error paths in all analyzed paths based on the classification result of project-specific learning. In this section, we evaluate the error path prediction of MLPEx. Given a set of functions, assume there are N true error paths, MLPEx reports N' paths as error paths, and among which N'_t are true error paths. We measure the prediction results by using standard metrics precision (P), recall (*re*), and F-measure (F_1), defined as follows.

$$pr = \frac{N'_t}{N'} \qquad re = \frac{N'_t}{N} \qquad F_1 = 2 \times \frac{pr \times re}{pr + re}$$

The precision is the ratio of correctly predicted error paths, the recall is the ratio of covered true error paths, and the F_1 is the measure of error path prediction accuracy.

		With project-specific features			Without project-specific features		
		Precision	Recall	F_1	Precision	Recall	F_1
Libs	zlib	0.947	0.974	0.960	0.792	0.974	0.873
	Libgcrypt	0.916	0.947	0.931	0.883	0.929	0.905
	OpenSSL	0.983	0.961	0.971	0.915	0.961	0.937
	GTK+	0.818	0.991	0.896	0.581	0.991	0.732
	libc	0.821	0.962	0.886	0.644	0.950	0.768
	GnuTLS	0.923	0.814	0.865	0.940	0.809	0.869
Apps	Linux	0.952	0.773	0.853	0.814	0.791	0.802
	httpd	0.859	0.954	0.904	0.689	0.966	0.804
	VLC	0.908	0.984	0.944	0.872	0.984	0.924
	wget	0.931	0.954	0.942	0.898	0.944	0.920
Overall		0.912	0.891	0.901	0.790	0.903	0.843

Fig. 17. Results of error path prediction with/without project-specific features. The results for "with project-specific features" are described in Section 6.4 and their difference with those for "without project-specific features" are described in Section 6.6.

Figure 17 (the columns under "With project-specific features") shows that, on average, the prediction precision of MLPEx for all 10 projects is 91.2% and the recall is 89.1%. Note that the F-measures for GnuTLS and Linux are 86.5% and 85.3%, respectively, which are about 5% lower than the average. The main reason is that the paths in the learned training set of each of the two projects have similar code structures. Therefore, the training data fed to project-specific learning is not representative enough, affecting the accuracy of the trained model. Figure 17 also shows that the accuracy of error path prediction across projects is stable, since the standard deviation of F-measure is 0.04.

Given the number and sizes of projects used for evaluation, it is infeasible to evaluate all functions in each project. A question is, how do we make sure that the result is representative? To address this, we have chosen 4 projects and for each we gradually investigated more functions and paths and assess if the F-measure is stable. For example, for Linux, we first randomly selected 200 functions, evaluate them, and recorded the result. We then chose another 200 functions and repeated the process. We continued until all 1,000 functions were evaluated. We also applied the same process to httpd, Libgcrypt, and libc, but with an interval of 100 functions. We plot the result in Figure 16. From the figure, we observe that the evaluation result is stable within each project. Thus, we are confident that our results are representative.

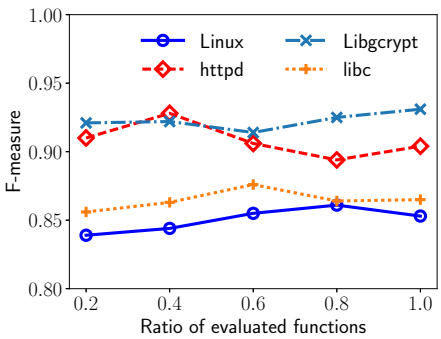


Fig. 16. Stability of error path prediction.

6.5 Error Specification Accuracy

In this section, we measure the inferred error specifications of MLPEx. We reuse the three metrics defined in Section 6.4 here but reinterpret N as the number of functions that we evaluate about

		Precision		Recall		
		MLPEx	APEx	MLPEx	MLPEx*	APEx
Libs	zlib	0.90	0.50	0.93	0.98	0.27
	Libgcrypt	0.80	0.82	0.63	0.89	0.64
	OpenSSL	1.0	0.78	0.87	0.96	0.62
	GTK+	0.85	0.84	0.93	1.0	0.36
	libc	0.86	0.76	0.82	0.87	0.65
	GnuTLS	0.94	0.74	0.65	0.92	0.34
Apps	Linux	0.97	N/A	0.80	0.98	N/A
	httpd	0.81	N/A	0.71	0.95	N/A
	VLC	1.0	N/A	0.91	1.0	N/A
	wget	0.82	N/A	0.86	0.96	N/A
Overall		0.91	0.77	0.78	0.94	0.47

Fig. 18. Results of generated error specifications. The recall values in column "MLPEx*" are calculated by filtering out the cases that lack necessary source code. The overall F-measure of MLPEx and APEx is 0.92 and 0.58, respectively.

their specifications, N' as the number of functions for which MLPEx generates specifications, and N'_t as the number of functions for which MLPEx generates correct specifications.

In Section 5, we discussed that MLPEx is unable to collect the full error specification for function `f` if it uses some function call `g` as the return expression in a path and the source code of the callee `g` is unavailable to MLPEx. Such limitation is not fundamental to MLPEx. Our evaluation considers two Ns, one with functions of the form `f` above and one without. Accordingly, we have two different recalls. The precision keeps the same no matter which N we use.

We present the precision and both recalls in Figure 18. We observe that the recall values (under "MLPEx" in the table) for Libgcrypt, GnuTLS, and httpd are much lower than those for other projects. The reason is that these projects are heavily dependent on third-party libraries that are not contained in our analyzed source code. For example, in libgcrypt that requires the third-party library libgpg-error, the error path MLPEx predicted in function `make_space` returns `GPG_ERR_TOO_LARGE`. However, since the code of libgpg-error was not included, we missed the error value of `GPG_ERR_TOO_LARGE` in the error specification.

In Figure 18, we also show the error specification results from APEx [Kang et al. 2016] for a comparison. Since most functions in applications are internal and APEx does not work for them, we put "N/A" for the entries where APEx is not applicable. We observe that the precision of MLPEx is much higher than that of APEx for all projects, except for Libgcrypt, for which APEx is 2% more precise than MLPEx. For recall, MLPEx is about 30% better than APEx for all projects, except for Libgcrypt, for which MLPEx and APEx perform similarly. If we ignore the functions that lack necessary source code, then the recall of MLPEx (under "MLPEx*" in Figure 18) is significantly higher than that of APEx. For zlib, GTK+, and GnuTLS, the recall of MLPEx is triple that of APEx, and for other three projects, MLPEx is 30% better. Overall, MLPEx outperforms APEx significantly in both precision and recall.

6.6 Usefulness of Project-specific Features

To improve the error path prediction precision, a key idea introduced in MLPEx is to use project-specific features to better differentiate error paths from non-error paths. In this section, we investigate the usefulness of project-specific features for error path prediction.

We repeated the process of evaluating the accuracy of path prediction without considering the project-specific features. We compare the results with those with project-specific features and present the comparison result in Figure 17. We find out that project-specific features improve the overall precision by about 12%, and the precision for GTK+ by 23%, libc and httpd by 17%, and Linux and zlib by 15%. Meanwhile, the overall recall with project-specific features is 89.1%, which is almost the same as that without project-specific features. The results mean that, with project-specific features, MLPEx discovers most error paths with fewer false positives (a false positive predicts a non-error path as an error path). Therefore, project-specific features are helpful in predicting error paths.

```
1  /* Linux-4.4.0/aio.c */
2  static struct kioctx *ioctx_alloc(unsinged nr_events)
3  {
4    ...
5    err = ioctx_add_table(ctx, mm)
6    if (err)
7      goto err_cleanup;
8    ...
9    return ctx;
10
11 err_cleanup:
12   ...
13   aio_free_ring(ctx);
14   ...
15   return ERR_PTR(err);
16 }
```

Fig. 19. An example of project-specific features helps predict error paths. The line marked red represents an error return, while the line marked green represents a correct return.

To illustrate how project-specific features help differentiate error paths from non-error paths, we use a concrete example shown in Figure 19. In this code excerpt, the error path p_{err} returning `ERR_PTR(err)` is predicted as a non-error path with universal features only. The reason is that the universal feature values of this path are quite similar to those of non-error paths in the training data. However, the return expression `ERR_PTR(err)` and the function call `aio_free_ring()` (at line 13) within the latest conditional before return statement match the patterns in the extracted project-specific information from the learned training set for Linux. MLPEx thus successfully classifies p_{err} as an error path with project-specific features.

6.7 MLPEx Performance

For each project, we measured the time for parsing, core-model learning, project-specific learning, and error specification generation. We present the detailed result in Figure 20. Since we use the third-party tool Joern to parse source code, we focus our discussion on the time consumed by other parts of MLPEx. The average total time to learn error specifications over the 10 projects is 614.8 seconds, without the code parsing time. Compared to APEx, our tool is much faster. For example, for GnuTLS, MLPEx took in total 15 mins to generate error specifications for 260 functions, while APEx took about 50 mins to generate error specifications for only 47 functions.

The number of paths may grow exponentially as the number of if-statements increases. For example, a code block with 20 sequential if-statements will lead to about 10^7 paths, requiring more time to analyze such a large number of paths. This is, however, an inherent problem to all path-sensitive analyses [Dillig et al. 2008]. Fortunately, functions with such large numbers of sequential if-statements are uncommon in practice.

To evaluate the scalability of MLPEx, we applied it on the memory management module of Linux, which contains 3,629 functions and 25,617 paths. The whole analysis process was finished in 8 hours, of which 58 mins was used by MLPEx for learning error specifications and the rest was used by Joern for parsing the source code. The result shows that MLPEx could scale to large projects.

	Linux	libc	httpd	GnuTLS	zlib
Code parsing	4,112.56	1,129.08	1,432.90	642.13	326.46
Learning phase I	456.95	182.44	175.83	63.73	18.94
Learning phase II	1,903.96	430.18	421.94	195.24	134.56
Error spec generation	3.15	2.39	2.30	1.43	1.73
Total	6,476.62	1,744.09	2,032.97	904.05	481.69

Fig. 20. Performance of MLPEx in seconds. The runtimes for Linux and zlib are the largest and smallest among all the evaluated projects, respectively, while those for libc, httpd and GnuTLS are in-between. The times are measured on CentOS 6.7 with 1 Intel i7-3940xm processor and 16 GB memory.

7 AN APPLICATION: DETECTING INCORRECT ERROR CODE ASSIGNMENTS

To handle a program failure correctly, a function should return an error code to inform the upper-level functions. In practice, however, developers may forget to propagate error values or the return value with a correct error code gets overwritten by a non-failure value due to complicated function logic, causing error assignment bugs [Gunawi et al. 2008; Liang et al. 2016; Tian and Ray 2017]. For example, in Figure 21, negative values should be propagated upstream when the call to `blk_mq_init_queue()` at line 11 fails. However, the developer forgot to set the appropriate value of `err` for this failure, and a non-failure value `0` is returned. This misleads the system into believing that the network block device is initialized successfully, even when unexpected events have occurred during the initialization `nbd_init()`.

```
1  /* Linux-4.9-rc6/nbd.c */
2  static int __init nbd_init(void)
3  {
4    ...
5    err = blk_mq_alloc_tag_set(&nbd_dev[i].tag_set);
6    if (err) {
7      put_disk(disk);
8      goto out;
9    }
10
11   disk->queue = blk_mq_init_queue(&nbd_dev[i].tag_set);
12   if (!disk->queue) {
13     blk_mq_free_tag_set(&nbd_dev[i].tag_set);
14     put_disk(disk);
15     goto out; //forgot to assign variable err  with correct error value.
16   }
17   ...
18   return 0;
19
20 out:
21   ...
22   return err;
23 }
```

Fig. 21. An example of an incorrect error code assignment. Function `nbd_init()` returns `0` and negative values to represent success and error, respectively.

We propose a simple approach EAB-Miner to detect error assignment bugs based on the generated error specifications and path prediction results from MLPEx. We illustrate how EAB-Miner works with the example in Figure 21. We represent a path by a sequence of program line numbers $\langle l_i, l_{i+1}, ...l_j \rangle$. By applying MLPEx to the source code containing `nbd_init()`, the path p_m $\langle 2, ...17, 18 \rangle$ is predicted as a non-error path, and the path p_n $\langle 2, ...12, 13, ...21, 22 \rangle$ is predicted as an error path. Intuitively, p_m and p_n should return different values since one is an error path while the other is not. However, the return values of p_m and p_n both evaluate to 0. EAB-Miner considers such conflicting situations as incorrect error code assignments. In general, EAB-Miner reports a potential error assignment bug if two paths from a single function are classified into different classes by MLPEx but return the same value.

We evaluated the effectiveness of detecting error assignment bugs of EAB-Miner on five projects, listed in Figure 22. For Linux, we applied our tool to its memory management module and 49 functions that have already been reported to have error assignment bugs in kernel Bugzilla. We did not apply EAB-Miner to the whole Linux because it takes Joern too much time to parse the full source code. For the rest of the four projects, we applied EAB-Miner to their full code base. The program versions used in this testing are the same as those in Section 6.1, which are the latest release versions, except for Linux and VLC.

Figure 22 presents the bug detection results of EAB-Miner. In total, EAB-Miner reports 72 functions that may contain error assignment bugs. We manually inspected all of them and found that 57 functions are very likely to have real bugs[5]. Thus, the overall precision of detecting bugs is 79%. Among the 57 potential bugs, 48 bugs are previously known, including 45 bugs from Linux (MLPEx detects 45 out of 49 included for testing) and 3 from VLC, which have been fixed in a later VLC release. The rest of the 9 bugs are previously unknown, including 1 from httpd, 4 from VLC, 2 from zlib, and 2 from OpenSSL. We have submitted these bugs to the development communities and are awaiting confirmations.

	# of Reported	# of Known	# of Unknown	Precision
Linux	56	45	0	0.80
httpd	2	0	1	0.50
VLC	9	3	4	0.77
zlib	2	0	2	1.0
OpenSSL	3	0	2	0.66
Total	72	48	9	0.79

Fig. 22. Results of detecting potential incorrect error code assignments. The "Known" bugs are those that have already been fixed by developers. The "Unknown" bugs are those still appear in the tested versions.

EAB-Miner reports a potential bug when MLPEx classifies two paths into different classes (error and non-error) but their returned expressions evaluate to a same value. The false positive rate of EAB-Miner is 21% as shown in Figure 22, which happens for two reasons: (1) a path is misclassified and (2) a return expression is evaluated incorrectly. Case (1) happens because the precision of error path prediction of MLPEx is not 100% (Figure 17). Case (2) happens because our current value evaluation of return expression is based on the idea of constant folding (Section 5), which is not very accurate and could be improved by, for example, using LLVM. Since our main goal in this paper is to learn precise error specifications and that in the section is to demonstrate the usefulness of error path prediction and error specifications, we leave the improvement of (2) as future work. In addition to detecting error assignment bugs, error specifications can be used to detect other error handling bugs. According to Tian and Ray [2017], 34% of all error handling bugs are caused by forgetting to check return values against errors. Such bugs can be detected if function return values are not checked against all values in the function's error specification. We plan to develop a tool for detecting missing error checks in the future.

8 RELATED WORK

Mining error specifications. Identifying error return values of functions is not easy in practice, since error range varies across projects and the error and non-error ranges may overlap within a single project. APEx [Kang et al. 2016] infers error specifications for APIs by analyzing their uses. Specifically, for each API, APEx considers its call sites in other projects, identifies the error

[5]EAB-Miner does not find an excessive number of error assignment bugs, which coincides with an empirical study performed by Tian and Ray [2017] that error assignment bugs account for about only 7% of all error handling bugs.

handling code at each call site and infers the corresponding error constraints, and uses the majority error constraints from multiple call sites to extract error specifications. Similarly, Acharya and Xie [2009] developed an error specification mining approach based on API usages in call sites. The limitation in these approaches is that, for each function, they require many distinct call sites for generating the error specification. MLPEx infers error specifications from function source code itself, and hence it can work for both APIs and internal functions.

LFI [Marinescu and Candea 2011] identifies errors APIs may return by profiling library binaries. Although LFI can work without source code, it employs a strong assumption that error return values are always constants, leading to low precision of error specifications. MLPEx does not rely on any assumption about error values. It generates error specifications based on the learned relation between a set of features and error paths

Relation with machine learning paradigms. Machine learning techniques have been used to solve problems in program analysis [David et al. 2016; DeFreez et al. 2018; L. Seidel et al. 2017; Long and Rinard 2016; Raychev et al. 2014; Wu et al. 2017; Zhu et al. 2016]. In general, there are three learning paradigms: *supervised learning*, *unsupervised learning*, and *semi-supervised learning*.

Essentially, supervised learning aims at solving classification problems, which requires a set of labeled data as training data to classify the unlabeled data into different groups [Hastie et al. 2009]. The barrier of applying supervised learning is that manual data labeling is inevitable [Alpaydin 2009]. MLPEx can automatically label data precisely, reducing the burden of users for using it to infer error specifications in practice.

The goal of unsupervised learning is to discover the precise data distribution of a given dataset by grouping data into different clusters [Alpaydin 2009]. For example, David et al. [2016] apply unsupervised learning for finding semantically similar pieces of code. However, without pre-labeled data or user feedback, we can determine only the data similarities but not the meanings of the resulting clusters. MLPEx transfers the knowledge about error and non-error paths learned from one project to other new projects, such that the paths in new projects are correctly labeled.

Semi-supervised learning could be an extension of either supervised or unsupervised learning [Zhu and Goldberg 2009]. It uses a few labeled data, together with a large amount of unlabeled data, to train a classifier. Unfortunately, semi-supervised learning would fail to work if all data does not follow the same distribution [Zhu and Goldberg 2009]. MLPEx is different from semi-supervised learning, which combines the advantages from both supervised and unsupervised learning to predict error paths without data assumption.

Zero-shot learning is an emerging concept in machine learning, which aims to classify objects whose instances may not have been seen during training [Wang et al. 2019; Wu et al. 2019b]. In this paper, we achieve it by proposing a new two-phase learning paradigm that makes use of both universal and project-specific features to train a model on an unseen project.

9 THREATS TO VALIDITY

There are several potential threats to the validity of the results reported in this paper. First, due to the scale of the evaluation, it's very time-consuming for each author to perform the whole study individually. However, we have used the following strategies to minimize this threat. (1) All authors cross-checked 1,000 paths to reach an agreement about the result. (2) The author who performed the evaluation read each function before evaluating its path prediction result to reduce mis-classifications. (3) We documented all the intermediate results for a second verification and artifact evaluation. (4) Our assessment in Figure 16 shows that our results are representative.

Second, MLPEx may not generalize to new projects well since the path distributions vary a lot across different projects. We have paid special attention to this potential threat and have chosen a

diverse set of projects to evaluate. As shown in Figure 13, we studied libraries, system software, and applications. The code structures and error ranges vary significantly across the evaluated projects, and non-error range is overlapping with error range in some individual projects. Therefore, our study subjects are representative. The evaluation results show that MLPEx works well across subjects, which suggests that our tool should generalize to any C projects.

10 CONCLUSION AND FUTURE WORK

In this paper, we targeted the problem of mining error specifications in C programs, which are useful for different kinds of C analyzers. We proposed a zero-shot learning approach, MLPEx, to infer precise error specifications directly from program source code. One of the main innovations is that we introduced a two-phase learning paradigm that requires minimal labeling efforts from us to initialize MLPEx, while no such efforts from users to use MLPEx. Theoretical analysis and evaluation results show that this novel learning paradigm enables MLPEx to generalize to different C projects. We believe that the same methodology could be easily applied to other specification mining problems. In addition, MLPEx effectively extracts and makes use of project-specific information for generating accurate error specifications, with precision and recall both being more than 90%. MLPEx helps remove a main obstacle for broader applications of many error-handling bug detection tools, which require precise error specifications as an input. We also used MLPEx to find 57 error handling bugs in 5 real-world projects.

In this paper, both universal and project-specific features extracted by MLPEx are predefined. We have shown that these features are useful to characterize paths in C programs. However, different programming languages may require designing their specific features. Such hand-crafted features thus would limit the applicability of MLPEx. One idea to make our approach more general is to learn these features directly from source code. Program representation learning [Allamanis et al. 2018a; Bengio et al. 2013] is an emerging research problem at the intersection of programming languages, software engineering, and machine learning. There are several approaches [Allamanis et al. 2018b; Alon et al. 2018, 2019; Chae et al. 2017; DeFreez et al. 2018] that try to automatically generate features, but they are designed for certain purposes. For example, to identify functions that are synonyms in a code base, DeFreez et al. [2018] represent code based on static function call traces; Allamanis et al. [2018b] embed a program as a graph by tracking the dependencies of the same variables and functions, such that the learned embedding can be used for code completion; and Alon et al. [2018] learn the program representations using paths in abstract syntax trees, which are useful to predict variable names, method names, and expression types.

Those learned representations in most existing studies are syntax-based, and thus do not fit well in our approach. The reason is that such syntax-based representations may not be able to convey adequate information when transferring the knowledge from the pre-labeled project to new projects. In the future, we plan to learn the representations of program semantics, which can serve as universal features. Moreover, we intend to substitute project-specific features with automatically learned syntax-based features. This will fully automate our two-phase learning paradigm and can extract different features from different projects to best utilize project-specific information.

ACKNOWLEDGMENTS

We thank anonymous OOPSLA reviewers for their constructive feedback, which has improved both the contents and the presentation of this paper. This work is partially supported by the National Science Foundation under the grant CCF-1750886.

REFERENCES

2007. OWASP TOP 10. https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

2019. CVE-2019-12818. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12818
2019. MongoDB. https://www.mongodb.com/
2019. Neo4j. https://neo4j.com/
Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (FASE '09)*. Springer-Verlag, Berlin, Heidelberg, 370–384.
Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018a. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.
Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018b. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations (ICLR '18)*.
Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 404–419. https://doi.org/10.1145/3192366.3192412
Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.
Ethem Alpaydin. 2009. *Introduction to machine learning*. MIT press.
Robert S Arnold. 1996. *Software change impact analysis*. IEEE Computer Society Press.
Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations (ICLR '17)*.
Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.
Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
Leo Breiman. 2017. *Classification and regression trees*. Routledge.
Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics for C-like Languages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 101 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133925
Yoonsik Cheon. 2007. Automated random testing to detect specification-code inconsistencies. *Technical Report UTEP-CS-07-07* (2007).
Flaviu Cristian. 1982. Exception handling and software fault tolerance. *IEEE Trans. Comput.* 6 (1982), 531–540.
Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 266–280. https://doi.org/10.1145/2908080.2908126
Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 423–433.
Isil Dillig, Thomas Dillig, and Alex Aiken. 2007. Static Error Detection Using Semantic Inconsistency Inference. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/1250734.1250784
Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 270–280. https://doi.org/10.1145/1375581.1375615
Stuart Geman, Elie Bienenstock, and René Doursat. 1992. Neural networks and the bias/variance dilemma. *Neural computation* 4, 1 (1992), 1–58.
John B. Goodenough. 1975. Structured Exception Handling. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '75)*. ACM, New York, NY, USA, 204–224. https://doi.org/10.1145/512976.512997
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 *6th USENIX Conference on File and Storage Technologies (FAST '08)*, Vol. 8. 1–16.
Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. Overview of supervised learning. In *The elements of statistical learning*. Springer, 9–41.
Jianping Hua, Zixiang Xiong, James Lowey, Edward Suh, and Edward R Dougherty. 2004. Optimal number of features as a function of sample size for various classification rules. *Bioinformatics* 21, 8 (2004), 1509–1515.
Jiayuan Huang, Arthur Gretton, Karsten Borgwardt, Bernhard Schölkopf, and Alex J Smola. 2007. Correcting sample selection bias by unlabeled data. In *Advances in neural information processing systems*. 601–608.
Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically Detecting Error Handling Bugs Using Error Specifications. In *USENIX Security Symposium*. 345–362.

Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated Inference of Error Specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 472–482. https://doi.org/10.1145/2970276.2970354

David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. 2002. *Logistic regression*. Springer.

Ioannis Kopanas, Nikolaos Avouris, and Sophia Daskalaki. 2002. The role of domain knowledge in a large scale data mining project. *Methods and Applications of Artificial Intelligence* (2002), 746–746.

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '17)*. ACM.

Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. 2018. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 94.

Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 333–344.

Huan Liu and Rudy Setiono. 1995. Chi2: Feature selection and discretization of numeric attributes. In *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 388–391.

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.

Paul D Marinescu and George Candea. 2011. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)* 29, 4 (2011), 11.

Steven S Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. 2015. An Empirical Study of Goto in C Code from GitHub Repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 404–414. https://doi.org/10.1145/2786805.2786834

Brian A. Nejmeh. 1988. NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM* 31, 2 (Feb. 1988), 188–200. https://doi.org/10.1145/42372.42379

Kamal Nigam, John Lafferty, and Andrew McCallum. 1999. Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, Vol. 1. 61–67.

Sinno Jialin Pan and Qiang Yang. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 504–515. https://doi.org/10.1145/1993498.1993558

Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 147.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

Cindy Rubio-González and Ben Liblit. 2010. Expect the unexpected: error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 73–80.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.

Victor S. Sheng, Foster Provost, and Panagiotis G. Ipeirotis. 2008. Get Another Label? Improving Data Quality and Data Mining Using Multiple, Noisy Labelers. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, USA, 614–622. https://doi.org/10.1145/1401890.1401965

Hidetoshi Shimodaira. 2000. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference* 90, 2 (2000), 227–244.

Masashi Sugiyama, Shinichi Nakajima, Hisashi Kashima, Paul V Buenau, and Motoaki Kawanabe. 2008. Direct importance estimation with model selection and its application to covariate shift adaptation. In *Advances in neural information processing systems*. 1433–1440.

Martin Susskraut and Christof Fetzer. 2006. Automatically finding and patching bad error handling. In *Dependable Computing Conference, 2006. EDCC'06. Sixth European*. IEEE, 13–22.

Yuchi Tian and Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 752–762. https://doi.org/10.1145/3106237.3106300

Vladimir Vapnik. 2013. *The nature of statistical learning theory*. Springer science & business media.

Wei Wang, Vincent W Zheng, Han Yu, and Chunyan Miao. 2019. A survey of zero-shot learning: Settings, methods, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 13.

Westley Weimer and George Necula. 2005. Mining temporal specifications for error detection. *Tools and Algorithms for the Construction and Analysis of Systems* (2005), 461–476.

Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019a. Generating Precise Error Specifications for C: A Zero Shot Learning Approach. Available at https://people.cmix.louisiana.edu/schen/ws/techreport/errspec.pdf.

Baijun Wu, John P. Campora III, and Sheng Chen. 2017. Learning User Friendly Type Error Messages. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '17)*. ACM.

Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019b. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 590–604.

Bianca Zadrozny. 2004. Learning and Evaluating Classifiers Under Sample Selection Bias. In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML '04)*. ACM, New York, NY, USA, 114–. https://doi.org/10.1145/1015330.1015425

Ying Zhang and Chen Ling. 2018. A strategy to apply machine learning to small datasets in materials science. *Npj Computational Materials* 4, 1 (2018), 25.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 491–507. https://doi.org/10.1145/2908080.2908125

Xiaojin Zhu and Andrew B Goldberg. 2009. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning* 3, 1 (2009), 1–130.