Information and Software Technology 152 (2022) 107043

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

CREX: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics

Dapeng Yan[a], Kui Liu[b,*], Yuqing Niu[a], Li Li[c], Zhe Liu[a], Zhiming Liu[d], Jacques Klein[e], Tegawendé F. Bissyandé[e]

[a] *Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[b] *Huawei Software Engineering Application Technology Lab, Hangzhou, China*
[c] *Monash University, Melbourne, Australia*
[d] *Northwestern Polytechnical University, Xian, China*
[e] *University of Luxembourg, Luxembourg City, Luxembourg*

ARTICLE INFO

Dataset link: https://github.com/1993ryan/crex

Keywords:
Program repair
Patch correctness
Semantic feature
Transfer learning

ABSTRACT

A significant body of automated program repair literature relies on test suites to assess the validity of generated patches. Because such oracles are weak, state-of-the-art repair tools can validate some patches that overfit the test cases but are actually incorrect. This situation has become a prime concern in APR, hindering its adoption by the industry. This work investigates execution semantic features based on micro-traces, a form of under-constrained dynamic traces. We build on transfer learning to explore function code representations that are amenable to semantic similarity computation and can therefore be leveraged for classifying patch correctness. Our CREX prototype implementation is based on the TREX framework. Experimental results on patches generated by the CoCoNut APR tool on CodeFlaws programs indicate that our approach can yield high accuracy in predicting patch correctness. The learned embeddings were proven to capture semantic similarities between functions, which was instrumental in training a classifier that identifies patch correctness by learning to discriminate between correctly patched code and incorrectly patched code based on their semantic similarity with the buggy function.

1. Introduction

Test-based automated program repair (APR) has witnessed great momentum in recent software engineering research literature [1–3]. Unfortunately, the (low) quality of the automatically-generated program patches affects potential interest from industry practitioners. The state-of-the-art mainly relies on weak test suites as the oracle for validating repair attempts (i.e., the correctness of APR-generated patches). And if a patch generated by an APR tool can make the patched buggy program pass all tests, the patch will be considered as a valid patch. However, such validation always yields patches that mostly overfit [4] the test suites (i.e., they can make the patched buggy programs pass all tests, but they do not really fix the related bugs) and are actually incorrect [5], or even, sometimes, are introducing more bugs [6].

Towards addressing the challenges of reliable patch validation, the community is actively exploring various techniques for identifying patch correctness automatically [7]. While some approaches propose strengthening the validation oracle by augmenting the test suite through automatic test case generation [8] or inspecting the output difference in dynamic execution traces [9], the trend in the last couple of years has been to investigate static features for predicting patch correctness, in order to bypass the oracle problem in test generation as well as to scale to the large number of patches produced by APR tools [5]. In this direction, Csuvik et al. [10] have built up the empirical observation that buggy code and correctly-patched code share some textual and structural similarity. Ye et al. [11] followed up by proposing a supervised learning-based approach with carefully engineered patch features at the syntax level while Tian et al. [12] leveraged deep representation learning of code changes for feeding to a classifier of patch correctness.

While the aforementioned static approaches have achieved promising performance [7] on benchmarks in the lab, they still suffer from one fundamental weakness: the lack of code semantic basis in their decision on patch correctness does not readily allow for practitioner's

* Corresponding author.
E-mail addresses: dapeng.yan@nuaa.edu.cn (D. Yan), brucekuiliu@gmail.com (K. Liu), 977012358@qq.com (Y. Niu), li.li@monash.edu (L. Li), zhe.liu@nuaa.edu.cn (Z. Liu), zliu@nwpu.edu.cn (Z. Liu), jacques.klein@uni.lu (J. Klein), tegawende.bissyande@uni.lu (T.F. Bissyandé).

https://doi.org/10.1016/j.infsof.2022.107043
Received 25 January 2022; Received in revised form 6 August 2022; Accepted 8 August 2022
Available online 12 August 2022

confidence. In contrast, dynamic-based approaches fully exploit semantic information of patch-based behavior collected from execution traces. Unfortunately, this is achieved through expensive execution campaigns of the test suite, which unfavorably affect the efficiency of the patch validation [5]. The test oracle problem also threatens dynamic approaches: given a test case, one may always lack an accurate specification of the output should be [13]. Recent literature has shown that despite the substantial analysis effort and the use of deep representation learning, it remains challenging to extract semantic (i.e., from execution behavior perspective) features from the disparate syntax and structure of programs [14,15]. Wang et al. have recently further shown that static approaches struggle with a generalization problem beyond the subjects they have been trained on [7].

This paper. We aim to find a balance between exploiting static code information and exploring dynamic execution behavior. To that end, we refer to strong results in the recent literature on patch correctness: (1) Xiong et al. [9] observed that the behavior of test case execution for the correct patch is different from the incorrect patches. This finding suggests that patch correctness can be decided by reasoning about *similarity of execution traces*. (2) Tian et al. [12], with deep representation learning, have suggested that *patch code syntax deeply embeds some semantics* that can be leveraged to predict correctness. Inspired by the work of Mckee et al. [14] on binary semantic analysis, we resort to the notion of *micro-execution*, which is the ability to execute any code fragment without a user-provided test driver or input data [16]. Then, we build on transfer learning to learn execution semantics explicitly from the micro-traces (cf. Section 2.1) that can be extracted from micro executions of the buggy and patched code.

We design and implement Crex, a model for identifying patch **c**o**r**rectness based on learned **ex**ecution semantics. Overall, our contributions are as follows:

- We investigate the automation of patch correctness identification in automated program repair of C programs. The state-of-the-art in this research direction has mainly focused on Java programs. Our efforts to run experimental validations have also uncovered various reproduction/replication issues with APR tools from the literature.
- We propose a novel perspective in patch correctness identification based on micro-executions and transfer learning of execution semantics. The learned embeddings are used by Crex to learn to predict correctness based on functional changes.
- We conduct experimental validations based on patches from CoCoNut and find that Crex is able to achieve high Recall (at 100%) and high F1 (at 89.0%) when the classifier is trained on micro-trace based embeddings with a Logistic regression learner.

2. Crex

Fig. 1 overviews the main steps of Crex. In the data preprocessing step, patches are first processed to infer the buggy program code and patched program code, and we collect execution traces from these code samples. Then, in the embedding process, these execution traces are fed to a pre-trained architecture to learn embeddings that characterize semantic executions of the buggy and patched code. With these embeddings, we can readily train classifiers to predict patch correctness in the last prediction step. In the remainder of this section, we first provide background information around the main concepts of micro-traces (cf. Section 2.1) and execution semantics (cf. Section 2.2), before discussing the design and implementation details of Crex (cf. Section 2.3).

2.1. Micro-traces: Representing execution semantics

Execution semantics refer to the behavior of the program when it is executed, in contrast to the static syntax of code. Such semantics are generally inferred by launching a campaign of dynamic tests based on a diverse set of input data covering the program execution paths. In a recent work, Godefroid [16] has presented the concept of micro-execution, which is a form of under-constrained dynamic execution that does not require a user-provided test driver or input data. Instead, any code fragment can be micro-executed by randomly initializing registers and memory, which will yield micro-traces of program micro-executions. A micro-execution indeed will produce five kinds of data: micro-trace code sequence, micro-trace value sequence, instruction position sequence, opcode/operand position sequence, and architecture sequence.

The micro-trace code sequence is an assembly code sequence generated by tokenizing the assembly instructions in the code fragment: these tokens capture the syntax and semantics of the instructions. The micro-trace value sequence is a sequence of the dynamic values of the corresponding assembly instruction code in the micro-trace. The instruction position sequence represents the relative positions between the instructions, and the opcode/operand position sequence represents the relative positions within each instruction. These positions are critical for inferring semantics (at the binary level). The architecture sequence describes the input binary's instruction set architecture (i.e., x86, x64, ARM, or MIPSgn) to distinguish the syntax of different architectures.

Fig. 2 illustrates the inference of micro-traces from a code sample: “Inst POS” and “OP POS” represent the instruction position sequence and opcode/operand position sequence. The micro-trace value sequence consists of four parts (i.e., Byte1, Byte2, Byte3, and Byte4). The concrete dynamic values in the micro-trace value sequence are independent tokens, which leads to prohibitively large vocabulary size. So they are divided into four parts with a hierarchical input encoding scheme [15].

In a recent work for binary analysis, Pei et al. [15] have demonstrated that micro-traces, although they only *approximate program behavior*, can be leveraged as carriers of execution semantics to enable the computation of functional semantic similarity at the binary level. In this work, we follow the same approach and collect micro-traces for patched and buggy programs to reason about patch correctness.

2.2. Transfer learning execution semantics

Deep representation learning approaches have been popular in the recent research literature for extracting features of program code in order to help reason about software semantics (to some extent). For example, NLP-based embedding techniques such as Word2Vec [17], Doc2Vec [17] and BERT [18] have been successfully applied for semantics-related tasks [12,19,20] and achieved promising results. However, code is fundamentally structural, which means that NLP models do not necessarily capture the appropriate signal to infer semantics. Alon et al. [21] took into consideration structural information in code and proposed code2vec model to mine semantic features from AST paths of code functions. Hoang et al. [22] proposed CC2Vec, which specializes in code changes, for program debugging tasks. Nevertheless, all these learning models focus on the code syntax and structure, which does not necessarily relate to program execution semantics [15].

In a recent work, Pei et al. [15] proposed learning execution semantics explicitly from functions' micro-traces using a hierarchical Transformer [23]. Their Trex framework has been shown to outperform state-of-the-art approaches in matching semantically similar functions. Concretely, considering the diversity of micro-traces that can be inferred from different implementations, architectures, and compilers, the five kinds of micro-traces sequences can help carry different contextual information for code functions. Firstly, the four micro-trace value sequences are encoded with a hierarchical input encoding scheme based on a 2-layer bidirectional LSTM model to address the challenge of the large vocabulary size of values. The other four micro-trace sequences are encoded with the one-hot encoded embedding matrix [24]. All of the sequences are encoded with the same embedding dimension

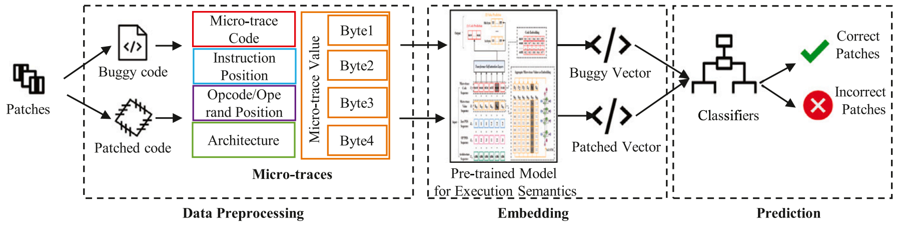
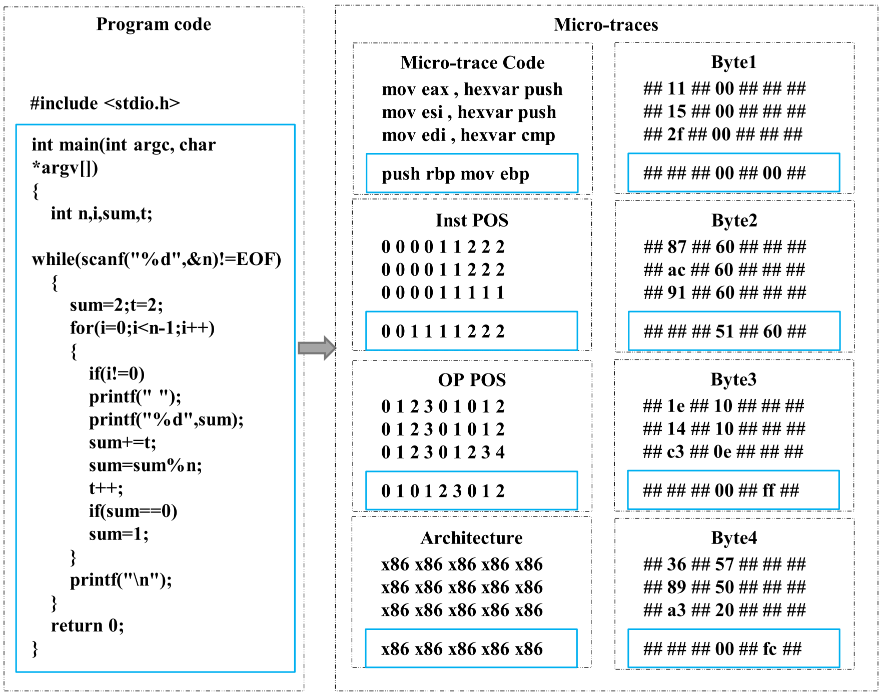


Fig. 1. Overview of Crex.



```
#include <stdio.h>

int main(int argc, char
*argv[])
{
  int n,i,sum,t;

while(scanf("%d",&n)!=EOF)
  {
    sum=2;t=2;
    for(i=0;i<n-1;i++)
    {
      if(i!=0)
      printf(" ");
      printf("%d",sum);
      sum+=t;
      sum=sum%n;
      t++;
      if(sum==0)
      sum=1;
    }
    printf("\n");
  }
  return 0;
}
```

```
Micro-trace Code
mov eax , hexvar push
mov esi , hexvar push
mov edi , hexvar cmp
push rbp mov ebp

Inst POS
0 0 0 0 1 1 2 2 2
0 0 0 0 1 1 2 2 2
0 0 0 0 1 1 1 1 1
0 0 1 1 1 1 2 2 2

OP POS
0 1 2 3 0 1 0 1 2
0 1 2 3 0 1 0 1 2
0 1 2 3 0 1 2 3 4
0 1 0 1 2 3 0 1 2

Architecture
x86 x86 x86 x86 x86
x86 x86 x86 x86 x86
x86 x86 x86 x86 x86
x86 x86 x86 x86 x86

Byte1
## 11 ## 00 ## ## ##
## 15 ## 00 ## ## ##
## 2f ## 00 ## ## ##
## ## ## 00 ## 00 ##

Byte2
## 87 ## 60 ## ## ##
## ac ## 60 ## ## ##
## 91 ## 60 ## ## ##
## ## ## 51 ## 60 ##

Byte3
## 1e ## 10 ## ## ##
## 14 ## 10 ## ## ##
## c3 ## 0e ## ## ##
## ## ## 00 ## ff ##

Byte4
## 36 ## 57 ## ## ##
## 89 ## 50 ## ## ##
## a3 ## 20 ## ## ##
## ## ## 00 ## fc ##
```

Fig. 2. Example of micro-traces converted from the code.

to facilitate integrating these encoded sequences into a single sequence that will be fed to the Transformer. After the sequence encoding, Trex was trained on an unsupervised pre-training task. In this process, given the micro-trace of a function, Trex first randomly masks some parts of the micro-trace and learns a masked language model to predict the masked parts using the non-masked parts without any additional labeling effort. To ensure that Trex will capture the execution semantic similarities beyond the syntactic similarities, the pre-training task of the masked language model is strengthened with several masking strategies (e.g., masking registers, masking opcodes, etc.). In addition, to learn the contextual information, Trex is employed with self-attention layers [23] to endow the context-sensitive meaning of each token to its embedding. Each token is assigned to a fixed embedding regardless of its changed contexts, which is different from the static embeddings with word2vec [25]. After the self-attention layers, the execution semantics of each instruction and the overall function will be encoded into final learned embeddings to represent the execution semantic features.

2.3. Implementation of Crex

Crex is implemented by concerning the three processes of data preprocessing for extracting the micro-traces related to the program before and after patching (cf. Section 2.3.1), learning the execution semantics and producing embeddings (cf. Section 2.3.2) and training a classifier to predict patch correctness (cf. Section 2.3.3).

2.3.1. Data pre-processing

Patches are textual syntax presentations of localized code changes for fixing bugs in programs. Because it is challenging to have a full view of the change impact from the patch, we propose rebuilding the two versions of the programs before and after the patch: the first is the buggy program; the latter is the patched program. To characterize how the changes impact the program behavior, we must retrieve execution semantics from both versions. Code syntax not being sufficient for this task, we infer micro-traces [15], the under-constrained dynamic traces (cf. Section 2.1), which we use to represent the execution semantics of the buggy code and the patch code at the function level. Our prototype implementation leverages a state of the art tool implemented by Godefroid [16] to perform micro-execution of programs. We then adopt Trex as the framework, which supports micro-executions, to collect the micro-traces for the buggy code and the patched code.

Trex produces micro-traces at the program level. Our design also considers the function level as the sweet spot for learning execution differences between buggy and patched code and deciding on correctness. Given a patch, we leverage annotations to identify the changed functions, and then we can circumscribe the relevant part of the micro-traces for the function. Given assembly language definition, we leverage the function call notation “`push rbp`” to pinpoint the starting point of our target function traces. We observe that the other representations of the micro-trace actions for each function have the same position as the micro-trace code sequence in the corresponding sequences (cf. the example in Fig. 2).

2.3.2. Embedding execution semantics

Each sequence of micro-traces represents different contextual information for the related functions. These sequences should therefore be appropriately integrated for feature learning of execution semantics. So far, prior works leveraging representation learning to embed patches for correctness identification have a focus on the static syntax code exploiting signals in the token stream or the AST structure [11,12]. In this work, we propose building representations that embed execution semantics using the micro-traces. To that end, we leverage a pre-trained model released by Pei et al. [15]: The model was trained on 1,472,066 functions collected from 13 popular open-source software projects and fine-tuned with 50,000 random function pairs for each project. This model was already proven effective in matching semantically similar functions by exceeding the prevailing state of the art results. Our implementation of Crex builds on this pre-trained model to produce embeddings of buggy code functions and patched code functions for learning to predict patch correctness.

2.3.3. Prediction of patch correctness

Given the previous code embeddings, which encode the execution semantics of buggy and patched code, we propose training binary classifiers to predict patch correctness. We consider several learning algorithms for the patch correctness prediction task: Logistic Regression, Naïve Bayes, Decision Tree, Random Forest, XGBoost, and Deep Neural Networks. These algorithms have already been employed in prior studies of patch correctness [12], and we expect to be able to compare the performance results on an equal basis.

3. Experimental study design

We enumerate the research questions that are investigated to evaluate our contributions before presenting the bugs and patch datasets. The reason we do it at the function level is so that the code snippets we analyze can both contain the context of the modification and remove redundant parts of the code.

3.1. Research questions

This work aims to investigate the possibility of learning execution semantics from the micro-traces of buggy functions and patched functions of the patches generated by APR tools to predict patch correctness. Thus, our investigation is conducted by answering the following research questions:

- **RQ-1.** *Do the execution semantics learned from the micro-traces (of the buggy and patched functions) align with the empirically validated conclusion that correct patches incur small changes?* Bug fixes always leverage small changes. This is a hypothesis that is widely accepted and built upon in the program repair literature [26–30]. With this RQ, we assess whether embeddings reflecting execution semantics indeed yield close cosine similarity scores for buggy and correctly-patched functions.
- **RQ-2.** *To what extent can dissimilarity measurements based on the execution semantics of buggy and patched code be used to filter out overfitting patches?* In this RQ, we investigate whether a threshold can be inferred to separate correct and incorrect patches based on the similarities of learned embeddings.
- **RQ-3.** *Can the Crex approach yield classifiers that are effective in predicting the correctness of patches generated by APR tools?* We assess the performance of Crex on predicting the correctness of patches produced by the CoCoNut repair tool. We also present comparisons against baseline approaches, which we replicated from prior work targeting patch correctness in Java.

3.2. Dataset

Our work applies to C programs. We consider Codeflaws [31] which is the most suitable benchmark for our work from those available in literature since it includes the buggy programs, which can be compiled, as well as the associated correct patches. Other benchmarks (e.g., DBGBench [32], IntroClass [33], and ManyBugs [33]) do not meet all these requirements at the same time. Codeflaws include C program defects with clear defect categories. This benchmark was widely used in the literature. Because of an implementation constraint in the underlying Trex framework, which must encode 8 micro-trace sequences within 512 characters, our experiments could only focus on 1582 bugs from the Codeflaws dataset.

Different APR tools have been exploited on C programs [34]. Unfortunately, only the authors of CoCoNut [35] publicly released their generated patches for 413 bugs from Codeflaws. Therefore, our study focus on the 212 patches from CoCoNut for which micro-traces could be computed, as the dataset for the experimental validation of the patch correctness predictor. After a careful manual inspection of patches generated for each buggy program, we were able to label, among the 212 patches, 170 patches as correct and 42 as incorrect.

3.3. Experimental setup

Crex has been implemented in 1874 lines of Python code. The latest version of the pre-trained Trex model[1] provided by Pei at al. [15] is used in Crex. All experiments were conducted on a platform with a configuration of 4-core CPU, 16GiB memory, and Ubuntu 16.04 64-bit operating system. The other configuring environment includes Python 3.6, Conda 4.10.1, and the latest PyTorch 1.9 as well as its required packages.

4. Experimental results

4.1. RQ-1: Similarity of execution semantics

We first investigate the distribution of (cosine) similarities between the execution semantics embeddings of the buggy and patched functions to assess the possibility of using the semantic execution embedding to represent the patches. To that end, we aim to answer the following three questions:

- **RQ-1.1:** Do different patches present different similarities between the buggy functions and patched functions based on their execution semantics embeddings?
- **RQ-1.2:** To what extent are the similarities between the buggy functions and the patched functions different from the similarities between non-semantically-similar functions?
- **RQ-1.3:** Will the correct patches present different semantic similarities from the incorrect patches?

RQ-1.1:. According to the AST type and Defect type taxonomies[2] of bugs provided by Codeflaws, we classify the 1582 bugs and their related patches into four AST types and 39 Defect types, respectively. Then, we calculate the cosine similarities between these buggy functions and their patched functions with the execution semantics embeddings.

The similarity distribution of the four AST type bugs is shown in Fig. 3, where "Higher-order", "OperanD", "Operator", and "Statement" represent the AST types that are related to bug locations as well as the corresponding code changes of patches. From the AST type category, the similarities between buggy function and patch function present different distributions. The "OperanD" and "Operator" categories contain

[1] https://drive.google.com/file/d/1xNcW8r01_J2OTZFh1B0eOG5ikj73zhwe/view.

[2] https://codeflaws.github.io.

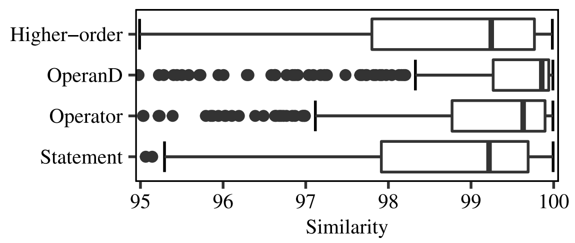


Fig. 3. Distribution of semantic similarities between buggy functions and patched functions with AST-type categories.

the bugs located on operand/operator with simple code changes. Thus the distributions of their semantic similarities tend to span less than the distribution similarities for patches in "Higher-order" and "Statement" categories where bugs are located and fixed at the statement or higher AST node levels.

Fig. 4 further shows the similarity distributions with the 39 Defect categories. Each defect type name is coined by concatenating the first one character (representing its AST type, i.e., D, H, O, and S representing OperanD, Higher-order, Operator, and Statement respectively) with its concrete code change action. For example, HIMS (Insert multiple non-branches statements, Higher-order type), DRAC (Replace constant of array initialization, OperanD type), ORRN (Replace relational operator, Operator type), and SISF (Insert function call, Statement type). For clearer definitions, please reference the aforementioned website. As presented in Fig. 4, we can observe that the 39 categories present different distributions of the similarities between their buggy functions and patched functions. It implies that the extracted execution semantics features of the buggy functions and patch functions can be used to distinguish the different categories of defects.

✍ Answers to RQ-1.1

The embeddings learned from execution semantics of buggy and patched code capture very well the different semantic characteristics of AST contexts per defect type: the yielded embeddings can be used to discriminate defect types.

RQ-1.2:. For this question, we aim to investigate to what extent the embeddings (based on execution semantics mined from micro-traces) help to identify which patched function is associated with which buggy function based on code semantic similarity. To this end, for the 1582 buggy functions we selected, we first randomly selected their both different AST and Defect-class types for each function of them to compose 1582 comparison groups. Therefore, each group of functions we compared comes from different programs and bug types, so that we consider that the 1582 groups we matched are not semantically similar functions and then compute their execution semantics-based embeddings. We then calculate the cosine similarity between the buggy function embedding and its non-semantically-similar function embedding. The distributions of these similarity values are illustrated in Fig. 5.

Only a single outlier of patched functions in Fig. 5 has a much lower similarity than others. Fig. 6 gives an overview of this outlier by presenting the function patch, where only a statement is deleted from the buggy function.

When looking at their micro-traces (shown in Fig. 7), the differences between these functions appear indeed significant, especially w.r.t. the four micro-trace value sequences. Such significant differences in micro-traces led to the low similarity between the buggy function and the patched one. We infer that the big changes in micro-trace values cause the differences (highlighted with yellow background, the same as other similar figures) of execution semantics.

Table 1
Statistics on the similarities for the ground-truth dataset.

Embedding	1st Qu.	Med.	3rd Qu.	Max.	Mean
BERT	96.2%	99.5%	100%	100%	95.1%
Word2Vec	99.8%	100%	100%	100%	99.7%
Crex	98.5%	99.5%	99.9%	100%	96.7%

✍ Answers to RQ-1.2

Embeddings learned from execution semantics are faithful to the semantic similarity between a patched function and its buggy version. When compared to semantically-dissimilar functions, the embeddings present high dissimilarity.

RQ-1.3:. APR tools generate plausible patches without the possibility for the imperfect test oracles to decide which are incorrect. As demonstrated by Xiong et al. [9], incorrect patches present different semantics, in the execution traces, from the correct patches. In this question, we thus assess to what extent learn embeddings capture these differences. To this end, we compute the similarities for the 170 correct and 42 incorrect patches collected from CoCoNut's results.

Fig. 8 shows that for correct patches, the similarities between buggy and patched functions are high, while this is not as often the case for incorrect patches. We further compute the correlation coefficient between the two groups of similarities to investigate the relationship between them. Specifically, we consider the Pearson correlation coefficient. If the Pearson correlation coefficient is closer than zero, the linear correlation between the two groups of tested data is weaker. Furthermore, if the value is less than zero, the linear correlation is negative. The Pearson correlation coefficient value is −0.038874, which is a negative value close to zero. It indicates that, given a buggy function, the correctly patched functions can be differentiated, with significant confidence, from the incorrectly patched functions, based on the embeddings learned from execution semantics inferred from micro-traces.

✍ Answers to RQ-1.3

Using the embeddings learned from execution semantics of functions, similarities between the correctly patched function and the buggy function significantly differ from the similarity between the incorrectly patched functions and the buggy function.

4.2. RQ-2: Identifying the correct patches with execution semantics and straightforward thresholds

Based on the findings in the first research question, we further investigate the possibility of setting a threshold similarity score to decide which APR-generated patches are likely incorrect. Results from this investigation will provide insights into the exploration of execution semantics in automated program repair.

To answer this question, we first resort the distribution of the similarities between the 1582 buggy and patched functions to guide the selection of thresholds. In this experiment, in addition to Crex embedding, we also consider two popular embedding models (BERT and Word2Vec) to extract the representation vectors from the syntax and static structure of buggy and patched functions. Both BERT and Word2Vec embeddings have been demonstrated effective by Tian et al. [12] for predicting patch correctness related to Java bugs in program repair. We consider both embedding models as a baseline for identifying correct patches with simple threshold settings.

Table 1 shows the statistic on the similarities between the 1582 buggy and patched functions. For instance, We can see that with both

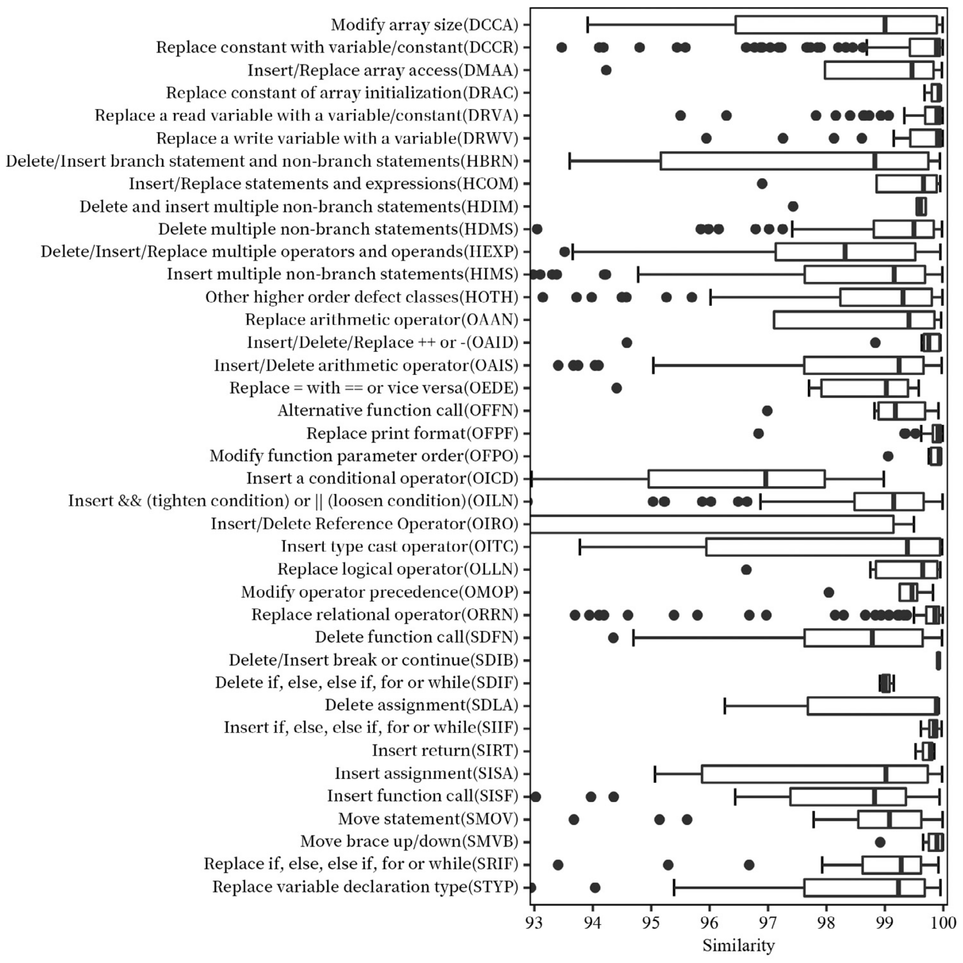


Fig. 4. Distribution on semantic similarities between buggy functions and patch functions with Defect-class categories.

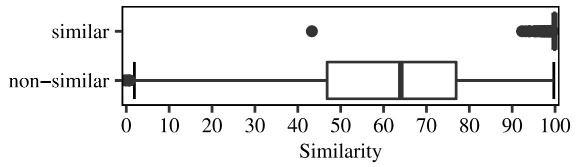


Fig. 5. Distributions of the similarities between the buggy functions and the patched functions against non-semantically-similar functions.

BERT and Crex embeddings, 50% of the buggy and patched functions have a similarity score higher than 99.5% (column Med., i.e., Median in Table 1). We also note that with Word2Vec, the similarity score is extremely high (100%) for most of the cases. More generally, for all three embeddings, the distribution of the similarities between the buggy and patched functions is narrow. This can be explained by the fact that their semantics are highly similar to each other due to simple code changes.

To answer RQ2, we follow a straightforward process: we consider the 1st and 3rd quartiles as well as the median and mean as possible thresholds. We then use these thresholds to predict correct and incorrect patches on the labeled dataset of 212 patches generated by CoCoNut. We note that, among these 212 patches, 170 are correct and 42 are incorrect. Table 2 presents the results. Overall, Crex outperforms BERT and Word2Vec models for the 4 metrics considered (i.e., Accuracy, Precision, Recall, and F1 score), except for the scenario with the 1st quartile, where BERT is slightly better. Such results show that leveraging the execution semantics assesses patch correctness effectively, and this embedding outperforms the deep representations learned from the code syntax and static structure (i.e., with BERT or Word2Vec).

Finally, as shown in Table 2, when setting a higher threshold (i.e., median and 3rd quarter), more incorrect patches are missed. While setting a relatively lower threshold (i.e., 1st quartile and mean) can contribute to catching more correct patches (higher recall). Practitioners could decide such a different setting by recalling more (in)correct patches.

✍ Answers to RQ-2

With the execution semantics learned from the micro-traces, it is possible to identify the correct patches generated by the APR tool by simply setting a threshold.

4.3. Predicting patch correctness with execution semantics

With the promising results presented in previous sections, we propose investigating the feasibility of predicting patch correctness for APR tools with the execution semantics learned from the micro-traces of patched functions. To this end, we compare the performance of Crex using different well-known classifiers: XGBoost, Random Forest, Logistic regression, Decision Tree, Naïve Bayes, and Deep Neural Networks (DNN). We also compare the performance of Crex against the related approach proposed by Tian et al. [12]. The authors present an approach to predict patch correctness with semantic features learned from code syntax and structure in this recent work. The underlying embedding is yielded by either BERT or Word2Vec. In this experiment, we consider

```
Ground-truth patched function:
$ diff 150-B-5820415.c 150-B-5820417.c
int main(int argc, char *argv[]) {
   int n, m, k;
   scanf("%d %d %d", &n, &m, &k);
- printf("%d\n", pow(100000, 2));
   if (k > n || k == 1) printf("%d\n", pow(m, n));
   else if (k == n) printf("%d\n", pow(m, (n + 1)/2));
   else if (k % 2) printf("%d\n", m*m);
   else printf("%d\n", m);
   return 0;
}
```

Fig. 6. Example of a patched function with low execution semantic similarity.

Table 2
Classification with different thresholds.

Model	Metric	Thresholds			
		1st Qu.	Median	3rd Qu.	Mean
BERT	#TP	148	110	6	150
	#TN	11	20	36	11
	#FP	31	22	6	31
	#FN	22	60	164	20
	Accuracy	**75.0%**	61.3%	19.8%	75.9%
	Precision	**82.7%**	83.3%	50.0%	**82.9%**
	Recall	**87.1%**	64.7%	3.5%	88.2%
	F1	**84.8%**	72.8%	6.6%	85.5%
Word2Vec	#TP	133	88	40	110
	#TN	10	19	34	14
	#FP	32	23	8	28
	#FN	37	82	130	60
	Accuracy	67.5%	50.5%	34.9%	58.5%
	Precision	80.6%	79.3%	83.3%	79.7%
	Recall	78.2%	51.8%	23.5%	64.7%
	F1	79.4%	62.6%	36.7%	71.4%
Crex	#TP	145	118	51	161
	#TN	10	23	36	6
	#FP	32	19	6	36
	#FN	25	52	119	9
	Accuracy	73.1%	**66.5%**	**41.0%**	**78.8%**
	Precision	81.9%	**86.1%**	**89.5%**	81.7%
	Recall	85.3%	**69.4%**	**30.0%**	**94.7%**
	F1	83.6%	**76.9%**	**44.9%**	**87.7%**

the 212 patches generated by CoCoNut (corresponding to 212 bugs in Codeflaws). Considering the small size of the dataset, we conduct 10-fold cross-validation in this experiment. We present the results of the experiment in Table 3 with five metrics: accuracy (Acc.), precision (Prec.) recall, F-measure (F1), and AUC (area under the ROC curve, i.e., comprehensive performance of the predictor).

Table 3 presents the performance comparison for predicting patch correctness. Overall, Crex achieves better performance with XGBoost and Logistic regression classifiers than the BERT and Word2Vec models but underperforms the two models with the other four classifiers. However, when Crex performs worse, the performance of Crex is only slightly lower than BERT and Word2Vec. Moreover, among all these metrics, we note that Crex achieves the highest value, i.e., Crex outperforms BERT and Word2Vec, for AUC metric at 61.6% (with Random Forest), for F1 at 89.0%, Accuracy at 80.2% and Recall at 100% (with Logistic Regression). Such promising results indicate that Crex can effectively predict patch correctness by leveraging the execution semantics learned from the micro-traces of patched functions. We also note again that the execution semantics based embedding of Crex outperforms the approaches of learning semantic features from the code syntax and structure.

When looking at the different classifiers, we note that the DNN classifier performs worse than the other classifiers for all three learning methods. It is reasonable considering the very small dataset size, as the DNN is a deep-learning classifier that should rely on a large and balanced dataset (as in [12]). When looking at the five other machine learning classifiers, the values of the related metrics for the BERT and word2Vec models vary largely. In contrast, Crex yields relatively stable values. This result suggests that the execution semantics learned

Table 3
Performance comparison for predicting patch correctness.

Classifier	Embedding	Accuracy	Precision	Recall	F1	AUC
XGBoost	BERT	74.1%	79.2%	91.8%	85.0%	46.4%
	Word2Vec	76.0%	79.6%	94.1%	86.2%	53.2%
	Crex	**78.3%**	**80.8%**	**95.9%**	**87.6%**	**55.9%**
Logistic Regression	BERT	78.3%	80.1%	97.1%	87.7%	**55.1%**
	Word2Vec	76.9%	79.9%	95.3%	86.9%	51.1%
	Crex	**80.2%**	**80.2%**	**100.0%**	**89.0%**	51.4%
Decision Tree	BERT	67.9%	78.5%	77.1%	77.7%	48.8%
	Word2Vec	68.4%	79.2%	**84.7%**	**79.7%**	**49.2%**
	Crex	**68.4%**	**81.7%**	77.1%	77.9%	47.2%
Random Forest	BERT	**76.9%**	**80.4%**	**94.1%**	**86.7%**	50.0%
	Word2Vec	74.5%	79.3%	92.4%	85.3%	58.7%
	Crex	75.0%	79.1%	93.5%	85.7%	**61.6%**
Naïve Bayes	BERT	58.6%	77.9%	65.9%	70.7%	47.7%
	Word2Vec	**70.0%**	**82.8%**	**77.6%**	**79.7%**	**60.0%**
	Crex	66.0%	82.1%	74.1%	77.5%	54.5%
DNN	BERT	56.1%	**52.9%**	51.0%	**47.1%**	51.0%
	Word2Vec	**59.5%**	36.0%	**52.1%**	39.6%	**52.1%**
	Crex	56.9%	28.5%	50.0%	33.7%	50.0%

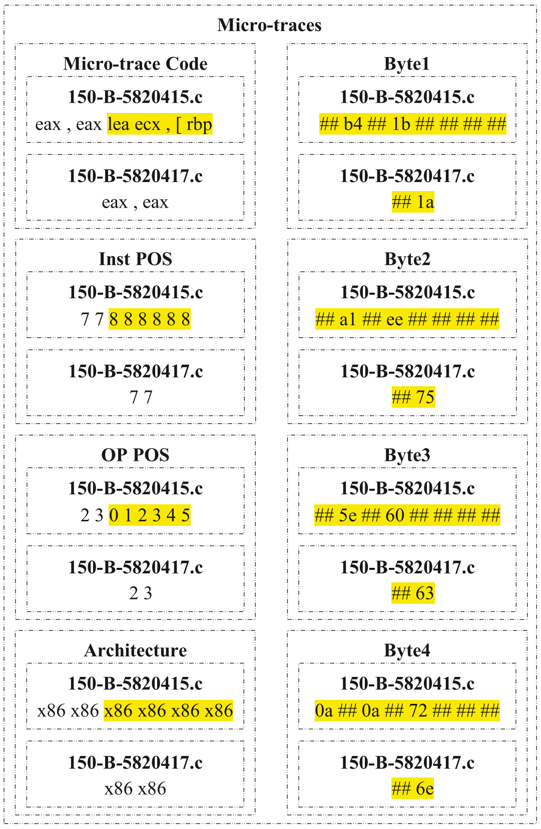


Fig. 7. Excerpted differences on micro-traces between the buggy function and its patch in Fig. 6 (differences are highlighted in yellow). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

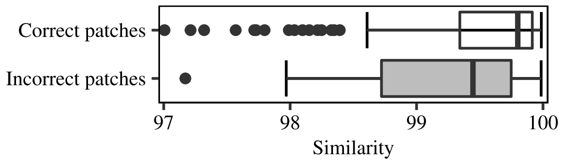


Fig. 8. Distributions of the similarities between the (in)correct patched functions and the buggy functions.

from micro-traces are less sensitive to the classification algorithm for predicting patch correctness.

Correct Patch Prediction Overlap: To further investigate the differences among the three embedding approaches, we study the classification of the correct patches by focusing on a single classifier. To that end, we take the XGBoost classifier as an example and put the analysis result in Fig. 9. Overall, the three models have a very high degree of overlap for identifying correct patches. Compared to the BERT and Word2Vec models, Crex is more efficient. It can indeed independently identify more correct patches than the other two models. Meanwhile, Crex predicted two correct patches that cannot be predicted by either BERT or Word2Vec. Only five correct patches predicted by both BERT and Word2Vec could not be predicted by Crex.

Why does Crex fail (#1): As stated, five patches are correctly classified by both BERT and Word2Vec, but not by Crex. Let us consider one of these patches to understand better why Crex can fail. This patch, generated by CoCoNut, is presented in Fig. 10.

From the aspect of code syntax, the patch removed only two tokens from six tokens with small changes. When looking at their micro-traces of the buggy function and the patched function, their micro-trace code, Inst POS, OP POS, and architecture sequences are the same, but

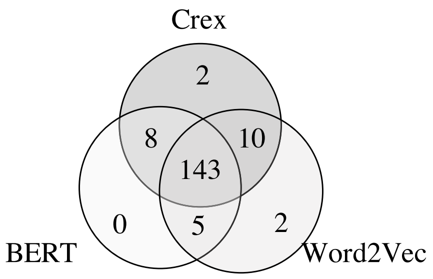


Fig. 9. # identified correct patches.

their micro-trace value sequences present great differences, as shown in Fig. 11, which leads to the low similarity of execution semantics.

Crex performance by type of patches: Based on the above experiments, we find Crex can identify more correct patches than the other two models. We further investigate the performance of Crex by different AST types on correct patches. We also take the XGBoost classifier as an example. The classification of bugs is based on their corresponding AST types as described in Section 4.1.

Fig. 12 presents, for each AST type, the number of correct patches predicted by Crex to be correct or incorrect. Overall, the accuracy of Crex in identifying correct patches is high for each AST type bug (ratio score always higher than 90%). Crex fails in identifying more correct patches for OperandD and Operator bugs than for the other two AST categories.

Why does Crex fail (#2): We focus on OperandD correct patches that have been misclassified by Crex. In particular, we select one misclassified patch and show the detailed diff information in Fig. 13. Then we investigate the changes of its Micro-traces in Fig. 14. The modification of the APR patch has not changed the micro-trace code, but micro-trace values have been impacted greatly.

In Fig. 14, we present the excerpted micro-trace value sequences, where their differences are highlighted in yellow. The small change of specific values in code leads to the great change of micro-trace value sequence, which further impacts the execution semantic learning and eventually results in the incorrect prediction of correct patches. As illustrated by the previous cases, Crex does not perform well on capturing the value changes in bug fixes, which should be carefully solved in the future work.

✍ Answers to RQ-3

❶ *With the execution semantics learned from the micro-traces,* Crex *presents a promising performance in predicting patch correctness with the recall and F1 metrics at 100% and 89.0%, respectively.* Crex *outperforms the state-of-the-art patch correctness prediction approaches relying on semantic features learned from code syntax and structure.*
❷ *The learned execution semantics are not sensitive to classification algorithms in contrast to the syntax and structure-based learning approaches.*
❸ Crex *might be further improved by reducing the weights of micro-trace value sequences in the transfer-learning process of execution semantics considering the value sequences always contain too specific values.*

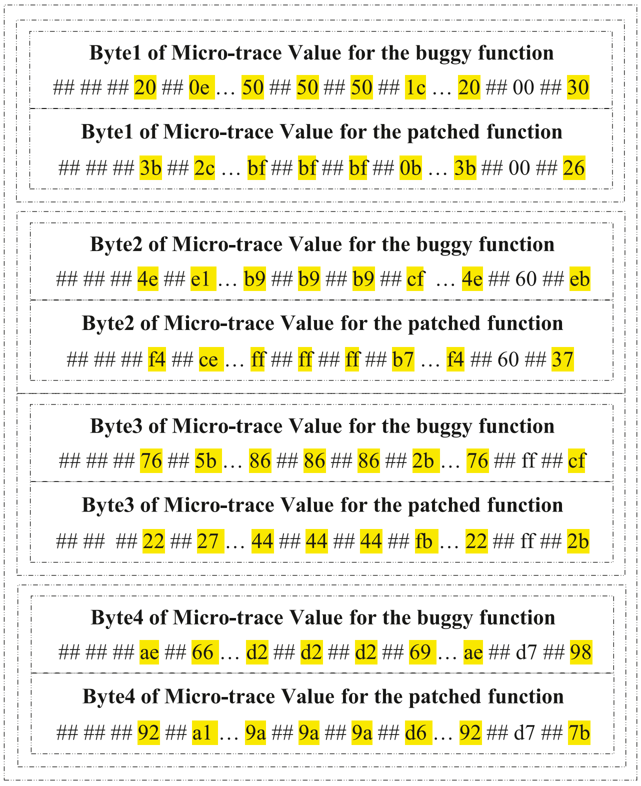
5. Threats to validity

Threats to External Validity: A threat to the validity of our study is the dataset used in our experiments. There are abundant C program bugs and patches that can be collected from open-source projects or datasets used in the literature. However, Crex requires the complete C program as its input. Such datasets released in the community are rare, and collecting such datasets from open-source projects needs heavy manual efforts to avoid bug-irrelated commits. So we resort

```
Ground-truth patch:
$ diff 197-A-4539529.c 197-A-4539541.c
- if(y%2==0)
+ if(y==0)

CoCoNut-generated patch:
$ diff 197-A-4539529.c 197-A-4539529-CoCoNut.c
- if(y%2==0)
+ if( y ==0)
```

Fig. 10. The diff information for CoCoNut patched program of bug "197-A-bug-4539529-4539541".



Byte1 of Micro-trace Value for the buggy function
20 ## 0e … 50 ## 50 ## 50 ## 1c … 20 ## 00 ## 30

Byte1 of Micro-trace Value for the patched function
3b ## 2c … bf ## bf ## bf ## 0b … 3b ## 00 ## 26

Byte2 of Micro-trace Value for the buggy function
4e ## e1 … b9 ## b9 ## b9 ## cf … 4e ## 60 ## eb

Byte2 of Micro-trace Value for the patched function
f4 ## ce … ff ## ff ## ff ## b7 … f4 ## 60 ## 37

Byte3 of Micro-trace Value for the buggy function
76 ## 5b … 86 ## 86 ## 86 ## 2b … 76 ## ff ## cf

Byte3 of Micro-trace Value for the patched function
22 ## 27 … 44 ## 44 ## 44 ## fb … 22 ## ff ## 2b

Byte4 of Micro-trace Value for the buggy function
ae ## 66 … d2 ## d2 ## d2 ## 69 … ae ## d7 ## 98

Byte4 of Micro-trace Value for the patched function
92 ## a1 … 9a ## 9a ## 9a ## d6 … 92 ## d7 ## 7b

Fig. 11. Difference of micro-trace value sequences for the bug and patch functions shown in Fig. 10.

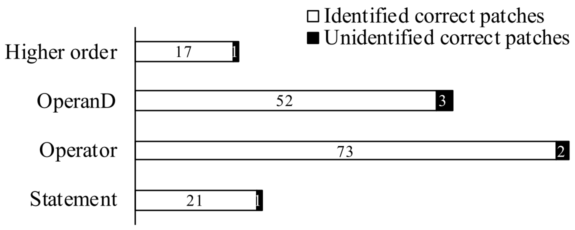


Fig. 12. # of (un)identified correct APR patches with AST types of bugs.

to the Codeflaws benchmark that has been widely used in the APR community to conduct our pioneer exploration. As for the data used in prediction, as many APR-generated patches should be considered as possible. Unfortunately, only CoCoNut made its patches publicly available. We failed to re-run other C program targeted APR tools mainly because they are not maintained anymore. Collecting more bug-patch pairs from open-source C programs and collecting APR-generated patches by repackaging C program targeted APR tools is considered as a future work of a deeper exploration on using execution semantics for predicting patch correctness in C programs.

Threats to Internal Validity: A major threat to the internal validity is from the input limitation of the pre-trained model for execution semantics, which constrains the length of each micro-trace sequence into 512 characters. We thus have to filter out some data in our experiments. We plan to build an attention neural-network-based model to address this limitation by concatenating the execution semantics for the long-sequence functions. The other threat to internal validity is that Crex can only identify the correctness of patches generated by APR-tools for single bugs. We plan to address this threat from two aspects: (1) untangling the patches of multiple bugs into several independent single patches, and (2) exploring the new identifying approaches that can be suitable for the correctness identification of patches for single bugs and multiple ones.

Threats to Construct validity: For our experiment, the considered classification algorithms are traditional machine learning algorithms, which might not be as powerful as deep learning algorithms. Our future studies on collecting more data for training the deep learning models will mitigate this threat. In the literature, several state-of-the-art deep learning based approaches have been proposed to predict patch correctness and achieved promising results, but all of them focused on the task of Java programs. We failed to replicate them on C program. It is the other threat to the construct validity for the comparison of this work. We also plan it as a future work with a complete comparison about different learning based approaches to boost the patch correctness identification.

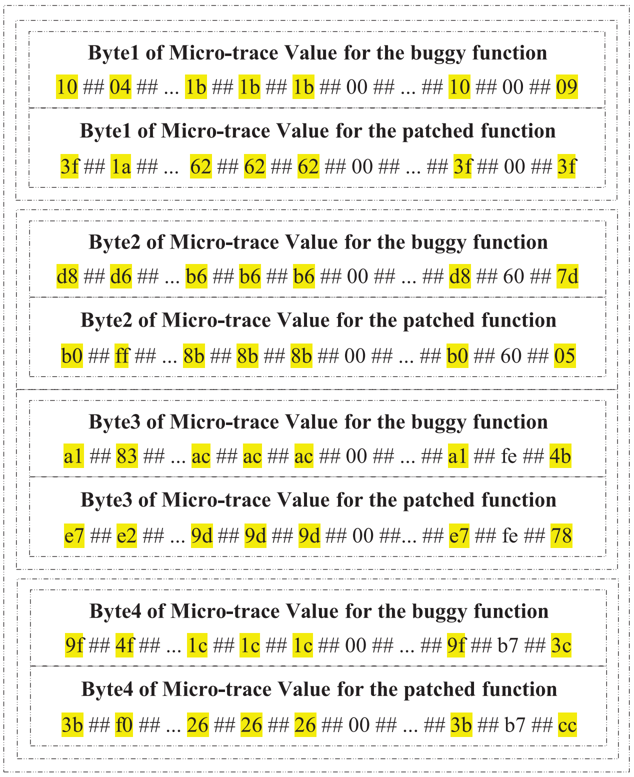
6. Related work

Analyzing patch correctness. The correctness of patches is the key to evaluating the performance of repair methods and tools to repair bugs. Unfortunately, this task was initially ignored by researchers until the emergence of research by Smith et al. [6]. They solved the shortcomings of early automatic repair technology evaluation. Early evaluation of automatic repair technology cannot correctly distinguish between correct patches and plausible (i.e., test oracle overfitting) patches. In a contemporary study, Qi et al. [4] analyzed the quality of the patches generated by three patch generation systems (GenProg [36], RSRepair [37], and AE [38]), and they found that most of those patches are actually incorrect but just overfitting to the test cases. Since then, the problem of patch overfitting has been attracting attention from researchers in the APR community. However, the patch correctness validation in APR community mainly relies on the manual identification with practitioners' knowledge [5,39–43]. Nevertheless, such manual intervening cannot boost the patch validation of APR. In this work, we explore learning the execution semantics of programs to predict patch correctness without human efforts.

```
Ground-truth patch:
$ diff 158-C-9815086.c  158-C-9815194.c
- for(i=1;param[i]!='\0';i++)
+ for(i=0;param[i]!='\0';i++)

CoCoNut-generated patch:
$ diff 158-C-9815086.c 158-C-9815086-CoCoNut.c
- for(i=1;param[i]!='\0';i++)
+ for( i =0; param [ i ] != '\0'; i ++ )
```

Fig. 13. The diff for the APR patched program and ground-truth program of bug "158-C-bug-9815086-9815194".



```
Byte1 of Micro-trace Value for the buggy function
10 ## 04 ## ... 1b ## 1b ## 1b ## 00 ## ... ## 10 ## 00 ## 09
Byte1 of Micro-trace Value for the patched function
3f ## 1a ## ... 62 ## 62 ## 62 ## 00 ## ... ## 3f ## 00 ## 3f

Byte2 of Micro-trace Value for the buggy function
d8 ## d6 ## ... b6 ## b6 ## b6 ## 00 ## ... ## d8 ## 60 ## 7d
Byte2 of Micro-trace Value for the patched function
b0 ## ff ## ... 8b ## 8b ## 8b ## 00 ## ... ## b0 ## 60 ## 05

Byte3 of Micro-trace Value for the buggy function
a1 ## 83 ## ... ac ## ac ## ac ## 00 ## ... ## a1 ## fe ## 4b
Byte3 of Micro-trace Value for the patched function
e7 ## e2 ## ... 9d ## 9d ## 9d ## 00 ##... ## e7 ## fe ## 78

Byte4 of Micro-trace Value for the buggy function
9f ## 4f ## ... 1c ## 1c ## 1c ## 00 ## ... ## 9f ## b7 ## 3c
Byte4 of Micro-trace Value for the patched function
3b ## f0 ## ... 26 ## 26 ## 26 ## 00 ## ... ## 3b ## b7 ## cc
```

Fig. 14. Difference of micro-trace value sequence for the bug and patch function in Fig. 13. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Identifying patch correctness. To contribute to patch correctness identification, researchers have proposed various approaches. Yang et al. [44] proposed generating better test cases to improve the process of validating APR-generated patches by enhancing existing test cases (using fuzzing to generate new test cases). Xin and Reiss [8] used new test inputs to get the original semantic differences between the error program and the patch program, then tested the patch program based on these differences and generated the final test case to identify the over-fitting patch program. Xiong et al. [9] leveraged the behavior similarity generated by the test case execution with new test input to enhance the test suite to determine the correctness of the new method of the patch. Experiments by Yu et al. [45] proved that incomplete repair and regression introduction are two common causes of overfitting. In this regard, they proposed a method called UnsatGuided, which uses additional tests to alleviate the overfitting problem of synthetic-based repair techniques. The above experiments are all used to enhance the test cases to predict the correctness of the patch. In the experiment of Ye et al. [11] pointed out that even very high test coverage may not be available. However, enhancing test cases for patch correctness identification always presents the generalizing problem [46]. Different from these studies, we resort to predict the correctness of the patch by learning the execution semantics from the micro-traces of programs.

In various recent studies, representational learning technology has also been widely used in the task of program repair. Hoang et al. [22] used the attention mechanism to model the hierarchy of code changes in Java programs and learn the representation of code changes guided by the accompanying log messages. Ye et al. [11] proposed a new type of overfitting detection system called ODS to predict the overfitting patches generated by APR. The model first statically extracts code features from the AST editing script between the generated patch and the error code. After that, ODS learns an ensemble probability model from the extracted static features, and the learning model classifies and ranks new potential overfitting patches. ODS requires manual identification of features, which makes these features unable to be generalized to other programming languages and data sets. Csuviket al. [10] utilized Doc2Vec and BERT to embed the textual and structural features of the original (error) program and to predict the correct patch. Tian et al. [12] evaluated the possibility of predicting the patch correctness with deep representation learning of code changes. Nevertheless, all of these state-of-the-art learning-based approaches focus on learning the potential semantic feature from the code syntax and structure. Comparing with them, we are the first to predict patch correctness for C programs by leveraging the transfer learning technique to extract the execution semantics from the micro-traces of C program functions.

7. Conclusion

To boost the momentum of APR, predicting the correctness of APR-generated patches has been explored with various approaches in the community. In this work, we proposed to predict the patch correctness for C programs, topic that has not been yet explored in the literature. To that end, we implemented a patch correctness prediction tool, named Crex. Specially, Crex leverages a transfer learning technique to extract the execution semantics from the micro-traces of C program functions. According to the different execution similarities, Crex is implemented with six different classification algorithms to predict the correctness of patches generated by a given APR tool. Our experimental results show that Crex achieves promising performance on predicting the patch correctness, which outperforms the state-of-the-art learning based approaches that leverage BERT and Word2Vec models to embed semantic features from the code syntax and structure. The transfer learned execution semantics point out the new potential direction for patch correctness validation in automated program repair.

CRediT authorship contribution statement

Dapeng Yan: Conceptualization, Methodology, Software, Writing – original draft. **Kui Liu:** Supervision, Methodology, Writing – review & editing. **Yuqing Niu:** Visualization, Investigation. **Li Li:** Visualization, Investigation. **Zhe Liu:** Funding acquisition. **Zhiming Liu:** Validation. **Jacques Klein:** Supervision, Writing – review & editing. **Tegawendé F. Bissyandé:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

For the sake of Open Science, we provide to the community the replication package with source code and all collected data for this study at the following address: https://github.com/1993ryan/crex.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62172214), the National Key R&D Program of China (No. 2020AAA0107704), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06).

References

[1] C.L. Goues, M. Pradel, A. Roychoudhury, Automated program repair, Commun. ACM 62 (12) (2019) 56–65.
[2] M. Monperrus, Automatic software repair: a bibliography, ACM Comput. Surv. 51 (1) (2018) 1–24.
[3] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, T.F. Bissyandé, A critical review on the evaluation of automated program repair systems, J. Syst. Softw. 171 (2021) 110817.
[4] Z. Qi, F. Long, S. Achour, M. Rinard, An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, in: Proceedings of the 24th International Symposium on Software Testing and Analysis, 2015, pp. 24–36.
[5] K. Liu, S. Wang, A. Koyuncu, K. Kim, T.F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, Y.L. Traon, On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 615–627.
[6] E.K. Smith, E.T. Barr, C. Le Goues, Y. Brun, Is the cure worse than the disease? overfitting in automated program repair, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 532–543.
[7] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, H. Jin, Automated patch correctness assessment: How far are we?, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 968–980.
[8] Q. Xin, S.P. Reiss, Identifying test-suite-overfitted patches through test case generation, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 226–236.
[9] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang, Identifying patch correctness in test-based program repair, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 789–799.
[10] V. Csuvik, D. Horváth, F. Horváth, L. Vidács, Utilizing source code embeddings to identify correct patches, in: Proceedings of the 2nd International Workshop on Intelligent Bug Fixing, IEEE, 2020, pp. 18–25.
[11] H. Ye, J. Gu, M. Martinez, T. Durieux, M. Monperrus, Automated classification of overfitting patches with statically extracted code features, IEEE Trans. Softw. Eng. (2021).
[12] H. Tian, K. Liu, A.K. Kaboré, A. Koyuncu, L. Li, J. Klein, T.F. Bissyandé, Evaluating representation learning of code changes for predicting patch correctness in program repair, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2020, pp. 981–992.
[13] F. Tsimpourlas, A. Rajan, M. Allamanis, Learning to encode and classify test executions, 2020, arXiv preprint arXiv:2001.02444.
[14] D. McKee, N. Burow, M. Payer, Software ethology: An accurate, resilient, and cross-architecture binary analysis framework, 2019, arXiv preprint arXiv:1906.02928.
[15] K. Pei, Z. Xuan, J. Yang, S. Jana, B. Ray, TREX: Learning execution semantics from micro-traces for binary similarity, 2020, arXiv preprint arXiv:2012.08680.
[16] P. Godefroid, Micro execution, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 539–549.
[17] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31th International Conference on Machine Learning, PMLR, 2014, pp. 1188–1196.
[18] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for ComputationalLinguistics: Human Language Technologies, 2019, pp. 4171–4186, http://dx.doi.org/10.18653/v1/n19-1423.
[19] K. Liu, D. Kim, T.F. Bissyandé, S. Yoo, Y.L. Traon, Mining fix patterns for FindBugs violations, IEEE Trans. Softw. Eng. 47 (1) (2021) 165–188, http://dx.doi.org/10.1109/TSE.2018.2884955.
[20] K. Liu, D. Kim, T.F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, Y. Le Traon, Learning to spot and refactor inconsistent method names, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 1–12.
[21] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, in: Proceedings of the ACM on Programming Languages, Vol. 3, ACM New York, NY, USA, 2019, pp. 1–29, POPL.
[22] T. Hoang, H.J. Kang, D. Lo, J. Lawall, Cc2vec: Distributed representations of code changes, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 518–529.
[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.
[24] D. Harris, S. Harris, Digital Design and Computer Architecture, Morgan Kaufmann, 2010.
[25] S.H. Ding, B.C. Fung, P. Charland, Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 472–489.
[26] E.T. Barr, Y. Brun, P. Devanbu, M. Harman, F. Sarro, The plastic surgery hypothesis, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 306–317.
[27] J. Chen, A.F. Donaldson, A. Zeller, H. Zhang, Testing and verification of compilers (Dagstuhl seminar 17502), Dagstuhl Rep. 7 (12) (2017) 50–65, http://dx.doi.org/10.4230/DagRep.7.12.50.
[28] J. Jiang, L. Ren, Y. Xiong, L. Zhang, Inferring program transformations from singular examples via big code, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 255–266.
[29] K. Liu, D. Kim, A. Koyuncu, L. Li, T.F. Bissyandé, Y. Le Traon, A closer look at real-world patches, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 275–286.
[30] M. Martinez, M. Monperrus, Mining software repair models for reasoning on the search space of automated program fixing, Empir. Softw. Eng. 20 (1) (2015) 176–205.
[31] S.H. Tan, J. Yi, Yulis, S. Mechtaev, A. Roychoudhury, Codeflaws: a programming competition benchmark for evaluating automated program repair tools, in: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion, 2017, pp. 180–182, http://dx.doi.org/10.1109/ICSE-C.2017.76.
[32] M. Böhme, E.O. Soremekun, S. Chattopadhyay, E. Ugherughe, A. Zeller, Where is the bug and how is it fixed? an experiment with practitioners, in: Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2017, 2017, pp. 1–11.
[33] C. Le Goues, N. Holtschulte, E.K. Smith, Y. Brun, P. Devanbu, S. Forrest, W. Weimer, The ManyBugs and IntroClass benchmarks for automated repair of C programs, IEEE Trans. Softw. Eng. 41 (12) (2015) 1236–1256.
[34] C. Le Goues, M. Pradel, A. Roychoudhury, S. Chandra, Automatic program repair, IEEE Softw. 38 (4) (2021) 22–27.
[35] T. Lutellier, H.V. Pham, L. Pang, Y. Li, M. Wei, L. Tan, Coconut: combining context-aware neural translation models using ensemble for program repair, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 101–114.
[36] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, in: Proceedings of the 34th International Conference on Software Engineering, IEEE, 2012, pp. 3–13.
[37] Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang, The strength of random search on automated program repair, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 254–265, http://dx.doi.org/10.1145/2568225.2568254.

[38] W. Weimer, Z.P. Fry, S. Forrest, Leveraging program equivalence for adaptive program repair: Models and first results, in: E. Denney, T. Bultan, A. Zeller (Eds.), Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2013, pp. 356–366, http://dx.doi.org/10.1109/ASE.2013.6693094.

[39] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 416–426.

[40] K. Liu, A. Koyuncu, D. Kim, T.F. Bissyandé, Avatar: Fixing semantic bugs with fix patterns of static analysis violations, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 1–12.

[41] K. Liu, A. Koyuncu, D. Kim, T.F. Bissyandé, Tbar: revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 31–42.

[42] K. Liu, A. Koyuncu, K. Kim, D. Kim, T.F. Bissyandé, LSRepair: Live search of fix ingredients for automated program repair, in: Proceedings of the 25th Asia-Pacific Software Engineering Conference, IEEE, 2018, pp. 658–662, http://dx.doi.org/10.1109/APSEC.2018.00085.

[43] M. Wen, J. Chen, R. Wu, D. Hao, S.-C. Cheung, Context-aware patch generation for better automated program repair, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 1–11.

[44] J. Yang, A. Zhikhartsev, Y. Liu, L. Tan, Better test cases for better automated program repair, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 831–841.

[45] Z. Yu, M. Martinez, B. Danglot, T. Durieux, M. Monperrus, Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system, Empir. Softw. Eng. 24 (1) (2019) 33–67.

[46] H. Tian, Y. Li, W. Pian, A.K. Kaboré, K. Liu, J. Klein, T.F. Bissyande, Checking patch behaviour against test specification, 2021, arXiv preprint arXiv:2107.13296.