Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Jordan Henkel
University of Wisconsin–Madison, USA
jjhenkel@cs.wisc.edu

Shuvendu K. Lahiri
Microsoft Research, USA
Shuvendu.Lahiri@microsoft.com

Ben Liblit
University of Wisconsin–Madison, USA
liblit@cs.wisc.edu

Thomas Reps
Univ. of Wisconsin–Madison and GrammaTech, Inc., USA
reps@cs.wisc.edu

ABSTRACT

With the rise of machine learning, there is a great deal of interest in treating programs as data to be fed to learning algorithms. However, programs do not start off in a form that is immediately amenable to most off-the-shelf learning techniques. Instead, it is necessary to transform the program to a suitable representation before a learning technique can be applied.

In this paper, we use abstractions of traces obtained from symbolic execution of a program as a representation for learning word embeddings. We trained a variety of word embeddings under hundreds of parameterizations, and evaluated each learned embedding on a suite of different tasks. In our evaluation, we obtain 93% top-1 accuracy on a benchmark consisting of over 19,000 API-usage analogies extracted from the Linux kernel. In addition, we show that embeddings learned from (mainly) semantic abstractions provide nearly triple the accuracy of those learned from (mainly) syntactic abstractions.

CCS CONCEPTS

• **Theory of computation** → *Abstraction*; • **Computing methodologies** → *Machine learning*; Symbolic and algebraic manipulation; • **Software and its engineering** → **Automated static analysis**; *General programming languages*; *Software verification and validation*; *Software testing and debugging*;

KEYWORDS

Word Embeddings, Analogical Reasoning, Program Understanding, Linux

ACM Reference Format:
Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code Vectors: Understanding Programs Through, Embedded Abstracted Symbolic Traces. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236085

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236085

1 INTRODUCTION

Computer science has a long history of considering programs as data objects [13, 16]. With the rise of machine learning, there has been renewed interest in treating programs as data to be fed to learning algorithms [3]. However, programs have special characteristics, including several layers of structure, such as a program's context-free syntactic structure, non-context-free name and type constraints, and the program's semantics. Consequently, programs do not start off in a form that is immediately amenable to most off-the-shelf learning techniques. Instead, it is necessary to transform the program to a suitable representation before a learning technique can be applied.

This paper contributes to the study of such representations in the context of *word embeddings*. Word embeddings are a well-studied method for converting a corpus of natural-language text to vector representations of words embedded into a low-dimensional space. These techniques have been applied successfully to programs before [17, 37, 43], but different encodings of programs into word sequences are possible, and some encodings may be more appropriate than others as the input to a word-vector learner.

The high-level goals of our work can be stated as follows:

> Devise a parametric encoding of programs into word sequences that (i) can be tuned to capture different representation choices on the spectrum from (mainly) syntactic to (mainly) semantic, (ii) is amenable to word-vector-learning techniques, and (iii) can be obtained from programs efficiently.

We also wish to understand the advantages and disadvantages of our encoding method. §5–§8 summarize the experiments that we performed to provide insight on high-level goal (ii).

We satisfy high-level goals (i) and (iii) by basing the encoding on a lightweight form of *intraprocedural symbolic execution*.

- We base our technique on *symbolic execution* due to the gap between syntax (e.g., tokens or abstract syntax trees (ASTs)) and the semantics of a procedure in a program. In particular, token-based techniques impose a heavy burden on the embedding learner. For instance, it is difficult to encode

the difference between constructions such as `a == b` and `!(a != b)` via a learned, low-dimensional embedding [5].
- Our method is *intraprocedural* so that different procedures can be processed in parallel.
- Our method is *parametric* in the sense that we introduce a level of mapping from symbolic-execution traces to the word sequences that are input to the word-vector learner. (We call these *abstraction mappings* or *abstractions*, although strictly speaking they are not abstractions in the sense of abstract interpretation [11].) Different abstraction mappings can be used to extract different word sequences that are in different positions on the spectrum of (mainly) syntactic to (mainly) semantic.

We have developed a highly parallelizable toolchain that is capable of producing a parametric encoding of programs to word sequences. For instance, we can process 311,670 procedures in the Linux kernel[1] in 4 hours,[2] using a 64-core workstation (4 CPUs each clocked at 2.3 GHz) running CentOS 7.4 with 252 GB of RAM.

After we present our infrastructure for generating parametric encodings of programs as word sequences (§2), there are a number of natural research questions that we consider.

First, we explore the utility of embeddings learned from our toolchain:

> Research Question 1: Are vectors learned from abstracted symbolic traces encoding *useful* information?

Judging utility is a difficult endeavor. Natural-language embeddings have the advantage of being compatible with several canonical benchmarks for word-similarity prediction or analogy solving [14, 20, 27, 30, 47, 48, 52]. In the domain of program understanding, no such canonical benchmarks exist. Therefore, we designed a suite of over nineteen thousand code analogies to aid in the evaluation of our learned vectors.

Next, we examine the impact of different parameterizations of our toolchain by performing an ablation study. The purpose of this study is to answer the following question:

> Research Question 2: Which abstractions produce the best program encodings for word-vector learning?

There are several examples of learning from syntactic artifacts, such as ASTs or tokens. The success of such techniques raises the question of whether adding a symbolic-execution engine to the toolchain improves the quality of our learned representations.

> Research Question 3: Do abstracted symbolic traces at the semantic end of the spectrum provide more utility as the input to a word-vector-learning technique compared to ones at the syntactic end of the spectrum?

[1]Specifically, we used a prerelease of Linux 4.3 corresponding to commit `fd7cd061adcf5f7503515ba52b6a724642a839c8` in the GitHub Linux kernel repository.

[2]During trace generation, we exclude only `vhash_update`, from `crypto/vmac.c`, due to its size.

Because our suite of analogies is only a proxy for utility in more complex downstream tasks that use learned embeddings, we pose one more question:

> Research Question 4: Can we use pre-trained word-vector embeddings on a downstream task?

The contributions of our work can be summarized as follows:

We created a toolchain for taking a program or corpus of programs and producing intraprocedural symbolic traces. The toolchain is based on Docker containers, is parametric, and operates in a massively parallel manner. Our symbolic-execution engine prioritizes the amount of data generated over the precision of the analysis: in particular, no feasibility checking is performed, and no memory model is used during symbolic execution.

We generated several datasets of abstracted symbolic traces from the Linux kernel. These datasets feature different parameterizations (abstractions), and are stored in a format suitable for off-the-shelf word-vector learners.

We created a benchmark suite of over 19,000 API-usage analogies.

We report on several experiments using these datasets:

- In §5, we achieve 93% top-1 accuracy on a suite of over 19,000 analogies.
- In §6, we perform an ablation study to assess the effects of different abstractions on the learned vectors.
- In §7, we demonstrate how vectors learned from (mainly) semantic abstractions can provide nearly triple the accuracy of vectors learned from (mainly) syntactic abstractions.
- In §8, we learn a model of a specific program behavior (which error a trace is likely to return), and apply the model in a case study to confirm actual bugs found via traditional static analysis.

Our toolchain, pre-trained word embeddings, and code-analogy suite are all available as part of the artifact accompanying this paper; details are given in §11.

Organization. The remainder of the paper is organized as follows: §2 provides an overview of our toolchain and applications. §3 details the parametric aspect of our toolchain and the abstractions we use throughout the remainder of the paper. §4 briefly describes word-vector learning. §5–§8 address our four research questions. §9 considers threats to the validity of our approach. §10 discusses related work. §11 describes supporting materials that are intended to help others build on our work. §12 concludes.

2 OVERVIEW

Our toolchain consists of three phases: transformation, abstraction, and learning. As input, the toolchain expects a corpus of buildable C projects, a description of abstractions to use, and a word-vector learner. As output, the toolchain produces an embedding of abstract tokens to double-precision vectors with a fixed, user-supplied, dimension. We illustrate this process as applied to the example in Fig. 1.

Phase I: Transformation. The first phase of the toolchain enumerates all paths in each source procedure. We begin by unrolling

```
int example() {
  buf = alloc(12);
  if (buf != 0) {
    bar(buf);
    free(buf);
    return 0;
  } else {
    return -ENOMEM;
  }
}
```

Figure 1: An example procedure

```
call alloc(12);
assume alloc(12) != 0;
call bar(alloc(12));
call free(alloc(12));
return 0;
```

(a) Trace 1

```
call alloc(12);
assume alloc(12) == 0;
return -ENOMEM;
```

(b) Trace 2

Figure 2: Traces from the symbolic execution of the procedure in Fig. 1

```
Called(alloc)
RetNeq(alloc, 0)
Called(bar)
Called(free)
```

(a) Abstracted Trace 1

```
Called(alloc)
RetEq(alloc, 0)
RetError(ENOMEM)
```

(b) Abstracted Trace 2

Figure 3: Result of abstracting the two traces in Fig. 2b

(and truncating) each loop so that its body is executed zero or one time, thereby making each procedure loop-free at the cost of discarding many feasible traces. We then apply an intraprocedural symbolic executor to each procedure. Fig. 2 shows the results of this process as applied to the example code in Fig. 1.

Phase II: Abstraction. Given a user-defined set of abstractions, the second phase of our toolchain leverages the information gleaned from symbolic execution to generate abstracted traces. One key advantage of performing some kind of abstraction is a drastic reduction in the number of possible tokens that appear in the traces. Consider the transformed trace in Fig. 2b. If we want to understand the relationship between allocators and certain error codes, then we might abstract procedure calls as *parameterized tokens* of the form `Called`(*callee*); comparisons of returned values to constants as parameterized `RetEq`(*callee*, *value*) tokens; and returned error codes as parameterized `RetError`(*code*) tokens. Fig. 3 shows the result of applying these abstractions to the traces from Fig. 2.

Phase III: Learning. Our abstracted representation discards irrelevant details, flattens control flows into sequential traces, and exposes key properties in the form of parameterized tokens that capture domain information such as Linux error codes. These qualities make abstracted traces suitable for use with a word-vector learner. Word-vector learners place words that appear in similar contexts close together in an embedding space. When applied to natural language, learned embeddings can answer questions such as "king is to queen as man is to what?" (Answer: woman.) Our goal is to learn embeddings that can answer questions such as:

- If a lock acquired by calling `spin_lock` is released by calling `spin_unlock`, then how should I release a lock acquired by calling `mutex_lock_nested`? That is, `Called`(`spin_lock`) is to `Called`(`spin_unlock`) as `Called`(`mutex_lock_nested`) is to what? (Answer: `Called`(`mutex_unlock`).)
- Which error code is most commonly used to report allocation failures? That is, which `RetError`(*code*) is most related to `RetEq`(`alloc, 0`)? (Answer: `RetError`(`ENOMEM`).)
- Which procedures and checks are most related to `alloc`? (Answers: `Called`(`free`), `RetNeq`(`alloc, 0`), etc.)

The remainder of the paper describes a framework of abstractions and a methodology of learning embeddings that can effectively solve these problems. Along the way, we detail the challenges that arise in applying word embeddings to abstract path-sensitive artifacts.

3 ABSTRACTIONS

One difference between learning from programs and learning from natural language is the size of the vocabulary in each domain. In natural language, vocabulary size is bounded (e.g., by the words in a dictionary, ignoring issues like misspellings). In programs, the vocabulary is essentially unlimited: due to identifier names, there are a huge number of distinct words that can occur in a program. To address the issue of vocabulary size, we perform an abstraction operation on symbolic traces, so that we work with abstracted symbolic traces when learning word vectors from programs.

3.1 Abstracted Symbolic Traces

We now introduce the set of abstractions that we use to create abstracted symbolic traces. Selected abstractions appear in the conclusions of the deduction rules shown in Fig. 4. The abstractions fall into a few simple categories. The `Called`(*callee*) and `AccessPathStore`(*path*) abstractions can be thought of as "events" that occur during a trace. Abstractions like `RetEq`(*callee*, *value*) and `Error` serve to encode the "status" of the current trace: they provide contextual information that can modify the meaning of an "event" observed later in the trace. Near the end of the trace, the `RetError`(*code*), `RetConst`(*value*), and `PropRet`(*callee*) abstractions provide information about the result returned at the end of the trace. Taken together, these different pieces of information abstract the trace; however, the abstracted trace is still a relatively rich digest of the trace's behavior.

With the abstractions described above, we found that the learned vectors were sub-optimal. Our investigation revealed that some of the properties we hoped would be learned required leveraging contextual information that was outside the "window" that a word-vector learner was capable of observing. For example, to understand the relationship between a pair of functions like `lock` and `unlock`, a word-vector learner must be able to cope with an arbitrary number of words occurring between the functions of interest. Such distances

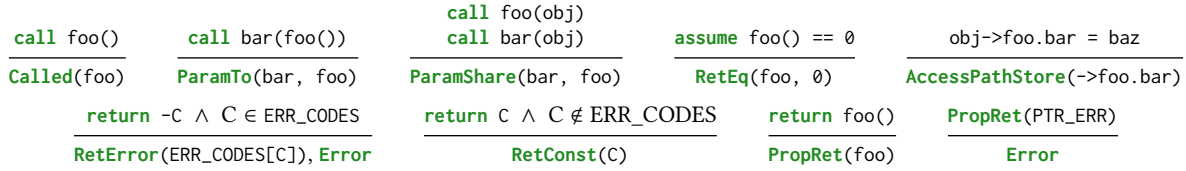

Figure 4: Example derivations for selected abstractions

```
                    // 1,2 FunctionStart
lock(&obj->lock);   // 1,2 Call(lock)
foo = alloc(12);    // 1,2 Call(alloc)
if (foo != 0) {     // 1 RetNeq(alloc, 0)
  obj->baz =        // 1 AccessPathStore(->baz)
    bar(foo);       // 1 ParamTo(bar, alloc)
                    // 1 Call(bar)
} else {            // 2 RetEq(alloc, 0)
  unlock(           // 2 ParamShare(unlock, lock)
    &obj->lock);    // 2 Call(unlock)
  return -ENOMEM;   // 2 RetError(ENOMEM)
                    // 2 Error
}                   // 2 FunctionEnd
unlock(             // 1 ParamShare(unlock, lock)
  &obj->lock);      // 1 Call(unlock)
return 0;           // 1 RetConst(0)
                    // 1 FunctionEnd
```

Figure 5: Sample procedure with generated abstractions shown as comments

are a problem, because lengthening the history given to a word-vector learner also increases the computational resources necessary to learn good vectors.

Due to the impracticality of increasing the context given to a word-vector learner, we introduced two additional abstractions: `ParamTo` and `ParamShare`. These abstractions encode the flow of data in the trace to make relevant contextual information available without the need for arbitrarily large contexts. As shown in §6, the abstractions that encode semantic information, such as dataflow facts, end up adding the most utility to our corpus of abstracted traces. This observation is in line with the results of Allamanis et al. [4], who found that dataflow edges positively impact the performance of a learned model on downstream tasks.

We augment the abstractions shown in Fig. 4, with the following additional abstractions, which are similar to the ones discussed above:

- `RetNeq`(*callee*, *value*), `RetLessThan`(*callee*, *value*), and others are variants of the `RetEq`(*callee*, *value*) abstraction shown in Fig. 4.
- `FunctionStart` and `FunctionEnd` are abstractions introduced at the beginning and end of each abstracted trace.
- `AccessPathSensitive`(*path*) is similar to `AccessPathStore`; it encodes any complex field and array accesses that occur in `assume` statements.

3.2 Encoding Abstractions as Words

We now turn to how the encoding of these abstractions as words and sentences (to form our trace corpus) can impact the utility of learned vectors. To aid the reader's understanding, we use a sample procedure and describe an end-to-end application of our abstractions and encodings.

Fig. 5 shows a sample procedure along with its corresponding abstractions. The number(s) before each abstraction signify which of the two paths through the procedure the abstraction belongs to. To encode these abstractions as words, we need to make careful choices as to what pieces of information are worthy of being represented as words, and how this delineation affects the questions we can answer using the learned vectors.

For instance, consider the `RetNeq(alloc, 0)` abstraction. There are several simple ways to encode this information as a sequence of words:

(1) `RetNeq(alloc, 0)` ⟹ `alloc` , `$NEQ` , `0`
(2) `RetNeq(alloc, 0)` ⟹ `alloc` , `$NEQ_0`
(3) `RetNeq(alloc, 0)` ⟹ `alloc_$NEQ` , `0`
(4) `RetNeq(alloc, 0)` ⟹ `alloc_$NEQ_0`

Each of these four encodings comes with a different trade-off. The first encoding splits the abstraction into several fine-grained words, which, in turn, reduces the size of the overall vocabulary. This approach may benefit the learned vectors because smaller vocabularies can be easier to work with. On the other hand, splitting the information encoded in this abstraction into several words makes some questions more difficult to ask. For example, it is much easier to ask what is most related to `alloc` being not equal to zero when we have just a single word, `alloc_$NEQ_0`, to capture such a scenario.

In our implementation, we use the fourth option. It proved difficult to ask interesting questions when the abstractions were broken down into fine-grained words. This decision did come with the cost of a larger vocabulary.[3] Encodings for the rest of our abstractions are shown in Fig. 6.[4] The sentences generated by applying these encodings to Fig. 5 are shown in Fig. 7.

4 WORD2VEC

Word2Vec is a popular method for taking words and embedding them into a low-dimensional vector space [30]. Instead of using a

[3] We mitigate the increase in vocabulary size from constructions like `alloc_$NEQ_0` by restricting the constants we look for. Our final implementation only looks for comparisons to constants in the set {−2, −1, 0, 1, 2, 3, 4, 8, 16, 32, 64}.
[4] Because it is not possible to have `ParamShare(X, Y)` or `ParamTo(X, Y)` without a `Called(X)` following them, the abstractions `ParamShare(X, Y)` and `ParamTo(X, Y)` are encoded as `Y` to avoid duplicating `X`.

```
match abstraction with
| Called (x)       -> x
| ParamTo (_,x)    -> x
| ParamShare (_,x) -> x
| RetEq (x,c)      -> x ^ "_$EQ_" ^ c
| RetNeq (x,c)     -> x ^ "_$NEQ_" ^ c
(* ... *)
| PropRet (x)      -> "$RET_" ^ x
| RetConst (c)     -> "$RET_" ^ c
| RetError (e)     -> "$RET_" ^ ERR_CODES[e]
| FunctionStart    -> "$START"
| FunctionEnd      -> "$END"
| Error            -> "$ERR"
| AccessPathStore (p)     -> "!" ^ p
| AccessPathSensitive (p) -> "?" ^ p
```

Figure 6: Encoding of abstractions

```
$START lock alloc alloc_$NEQ_0 !->baz
alloc bar lock unlock $RET_0 $END
```

(a) Trace 1

```
$START lock alloc alloc_$EQ_0 lock
unlock $ERR $RET_ENOMEM $END
```

(b) Trace 2

Figure 7: Traces for Fig. 5 generated by the encoding from Fig. 6

one-hot encoding—where each element of a vector is associated with exactly one word—word2vec learns a denser representation that captures meaningful syntactic and semantic regularities, and encodes them in the cosine distance between words.

For our experiments, we used GloVe [41] due to its favorable performance characteristics. GloVe works by leveraging the intuition that word-word co-occurrence probabilities encode some form of meaning. A classic example is the relationship between the word pair "ice" and "steam" and the word pair "solid" and "gas." Gas and steam occur in the same sentence relatively frequently, compared to the frequency with which the words gas and ice occur in the same sentence. Consequently, the following ratio is significantly less than 1:

$$\frac{\Pr(\text{gas} \mid \text{ice})}{\Pr(\text{gas} \mid \text{steam})}$$

If, instead, we look at the frequency of sentences with both solid and ice compared to the frequency of sentences with both solid and steam, we find the opposite. The ratio

$$\frac{\Pr(\text{solid} \mid \text{ice})}{\Pr(\text{solid} \mid \text{steam})}$$

is much greater than 1. This signal is encoded into a large co-occurrence matrix. GloVe then attempts to learn word vectors for which the dot-product of two vectors is close to the logarithm of their probability of co-occurrence.

5 RQ1: ARE LEARNED VECTORS USEFUL?

Research Question 1 asked whether vectors learned from abstracted symbolic traces encode useful information. We assess utility via three experiments over word vectors. Each of the following subsections describes and interprets one experiment in detail.

5.1 Experiment 1: Code Analogies

An interesting aspect of word vectors is their ability to express relationships between analogous words using simple math and cosine distance. Encoding analogies is an intriguing byproduct of a "good" embedding and, as such, analogies have become a common proxy for the overall quality of learned word vectors.

No standard test suite for *code* analogies exists, so we created such a test suite using a combination of manual inspection and automated search. The test suite consists of twenty different categories, each of which has some number of function pairs that have been determined to be analogous. For example, consider `mutex_lock_nested/mutex_unlock` and `spin_lock/spin_unlock`; these are two pairs from the "lock / unlock" category given in Tab. 1. We can construct an analogy by taking these two pairs and concatenating them to form the analogy "`mutex_lock_nested` is to `mutex_unlock` as `spin_lock` is to `spin_unlock`." By identifying high-level patterns of behavior, and finding several pairs of functions that express this behavior, we created a suite that contains 19,042 code analogies.

Tab. 1 lists our categories and the counts of available pairs, along with a representative pair from each category. Tab. 1 also provides accuracy metrics generated using the vectors learned from what we will refer to as the "baseline configuration,"[5] which abstracts symbolic traces using all of the abstractions described in in §3. We used a grid-search over hundreds of parameterizations to pick hyperparameters for our word-vector learner. For the results described in this section, we used vectors of dimension 300, a symmetric window size of 50, and a vocabulary-minimum threshold of 1,000 to ensure that the word-vector learner only learns embeddings for words that occur a reasonable number of times in the corpus of traces. We trained for 2,000 iterations to give GloVe ample time to find good vectors.

In each category, we assume that any two pairs of functions are sufficiently similar to be made into an analogy. More precisely, we form a test by selecting two distinct pairs of functions $(\mathcal{A}, \mathcal{B})$ and $(C, \mathcal{D})$ from the same category, and creating the triple $(\mathcal{A}, \mathcal{B}, C)$ to give to an analogy solver that is equipped with our learned vectors. The analogy solver returns a vector $\mathcal{D}'$, and we consider the test passed if $\mathcal{D}' = \mathcal{D}$ and failed otherwise. Levy and Goldberg [25] present the following objective to use when solving analogies with word-vectors:

$$\mathcal{D}' = \underset{d \in \mathbb{V} \setminus \{\mathcal{A}, \mathcal{B}, C\}}{\arg\max} \cos(d, \mathcal{B}) - \cos(d, \mathcal{A}) + \cos(d, C)$$

Results. The "Accuracy" column of Tab. 1 shows that overall accuracy on the analogy suite is excellent. Our embeddings achieve

[5] The baseline configuration is described in more detail in §6, where it is also called configuration (1).

Table 1: Analogy Suite Details

Type	Category	Representative Pair	# of Pairs	Passing Tests	Total Tests	Accuracy
Calls	16 / 32	`store16/store32`	18	246	306	80.39%
Calls	Add / Remove	`ntb_list_add/ntb_list_rm`	9	72	72	100.0%
Calls	Create / Destroy	`device_create/device_destroy`	19	302	342	88.30%
Calls	Enable / Disable	`nv_enable_irq/nv_disable_irq`	62	3,577	3,782	94.58%
Calls	Enter / Exit	`otp_enter/otp_exit`	12	122	132	92.42%
Calls	In / Out	`add_in_dtd/add_out_dtd`	5	20	20	100.0%
Calls	Inc / Dec	`cifs_in_send_inc/cifs_in_send_dec`	10	88	90	97.78%
Calls	Input / Output	`ivtv_get_input/ivtv_get_output`	5	20	20	100.0%
Calls	Join / Leave	`handle_join_req/handle_leave_req`	4	8	12	66.67%
Calls	Lock / Unlock	`mutex_lock_nested/mutex_unlock`	53	2,504	2,756	90.86%
Calls	On / Off	`b43_led_turn_on/b43_led_turn_off`	19	303	342	88.60%
Calls	Read / Write	`memory_read/memory_write`	64	3,950	4,032	97.97%
Calls	Set / Get	`set_arg/get_arg`	22	404	462	87.45%
Calls	Start / Stop	`nv_start_tx/nv_stop_tx`	31	838	930	90.11%
Calls	Up / Down	`ixgbevf_up/ixgbevf_down`	24	495	552	89.67%
Complex	Ret Check / Call	`kzalloc_$NEQ_0/kzalloc`	21	252	420	60.00%
Complex	Ret Error / Prop	`write_bbt_$LT_0/$RET_write_bbt`	25	600	600	100.0%
Fields	Check / Check	`?->dmaops/?->dmaops->altera_dtype`	50	2,424	2,450	98.94%
Fields	Next / Prev	`!.task_list.next/!.task_list.prev`	16	240	240	100.0%
Fields	Test / Set	`?->at_current/!->at_current`	39	1,425	1,482	96.15%
Totals:			**508**	**17,890**	**19,042**	**93.95%**

greater than 90% top-1 accuracy on thirteen out of the twenty categories. The learned vectors do the worst on the "Ret Check / Call" category where the top-1 accuracy is only 60%. This category is meant to relate the checking of the return value of a call with the call itself. However, we often find that one function allocates memory, while a different function checks for allocation success or failure. For example, a wrapper function may allocate complex objects, but leave callers to check that the allocation succeeds. Because our vectors are derived from intraprocedural traces, it is sensible that accuracy suffers for interprocedural behaviors.

By contrast, our vectors perform extraordinarily well on the "Ret Error / Prop" category (100% top-1). This category represents cases where an outer function (i) performs an inner call, (ii) detects that it has received an error result, and (iii) returns ("propagates") that error result as the outer function's own return value. Unlike for the "Ret Check / Call" category, the nature of the "Ret Error / Prop" category ensures that both the check and the return propagation can be observed in intraprocedural traces, without losing any information.

5.2 Experiment 2: Simple Similarity

One of the most basic word-vector tasks is to ask for the k nearest vectors to some chosen vector (using cosine distance). We expect the results of such a query to return a list of relevant words from our vocabulary. Our similarity experiments were based on two types of queries: (i) given a word, find the closest word, and (ii) given a word, find the five closest words.

```
ret = new(/*...*/, &priv->bo);
if (!ret) {
  ret = pin(priv->bo, /*...*/);
  if (!ret) {
    ret = map(priv->bo);
    if (ret)
      unpin(priv->bo);
  }
  if (ret)
    ref(NULL, &priv->bo);
}
```

Figure 8: Excerpt from `nv17_fence.c`. Names have been shortened to conserve space.

Similar pairs. We identified the single most similar word to each word in our vocabulary $\mathbb{V}$. This process produced thousands of interesting pairs. In the interest of space, we have selected four samples which are representative of the variety of high-level relationships encoded in our learned vectors[6]:

- `sin_mul` and `cos_mul`
- `dec_stream_header` and `dec_stream_footer`
- `rx_b_frame` and `tx_b_frame`

[6]The artifact accompanying this paper includes a full listing of these pairs, ordered by cosine-similarity.

- `nouveau_bo_new_$EQ_0` and `nouveau_bo_map` [7]

The last pair is of particular interest, because it expresses a complex pattern of behavior that would be impossible to encode without some abstraction of the path condition. The last pair suggests that there is a strong relationship between the function `new` returning `0` (which signals a successful call) and then the subsequent performance of some kind of map operation with the `map` call. To gain a deeper understanding of what the vectors are encoding, we searched for instances of this behavior in the original source code. We found several instances of the pattern shown in Fig. 8.

The code in Fig. 8 raise a new question: why isn't `pin` more closely related to `new_$EQ_0` ? We performed additional similarity queries to gain a deeper understanding of how the learned vectors have encoded the relationship between `new` , `pin` , and `map` .

First, we checked to see how similar `pin` is to `new_$EQ_0` . We found that `pin` is the fourth-most related word to `new_$EQ_0` , which suggests that a relationship does exist, but that the relationship between `new_$EQ_0` and `pin` is not as strong as the one between `new_$EQ_0` and `map` . Looking back at the code snippet (and remembering that several more instances of the same pattern can be found in separate files), we are left with the fact that `pin` directly follows from the successful `new` . Therefore, intuition dictates that `pin` should be more strongly related to `new` than `map` . The disagreement between our intuition and the results of our word-vector queries motivated us to investigate further.

By turning to the traces for an answer, we uncovered a more complete picture. In 3,194 traces, `new` co-occurs with `pin` . In 3,145 traces, `new` co-occurs with `map` . If we look at traces that do *not* contain a call to `new` , there are 11,354 traces that have no call to `new` , but still have a call to `pin` . In contrast, only 352 traces have no call to `new` , but still have a call to `map` . Finally, we have a definitive answer to the encoding learned by the vectors: it is indeed the case that `new` and `map` are more related in our corpus of traces, because almost every time a call to `map` is made, a corresponding call to `new` precedes it. Our intuition fooled us, because the snippets of source code only revealed a partial picture.

Top-5 similar words and the challenge of prefix dominance. Another similarity-based test is to take a word and find the top-k closest words in the learned embedding space. Ideally, we'd see words that make intuitive sense. For the purpose of evaluation, we picked two words: `affs_bread` , a function in the AFS file system that reads a block, and `kzalloc` , a memory allocator. For each target word, we evaluated the top-5 most similar words for relevance. In the process, we also uncovered an interesting challenge when learning over path-sensitive artifacts, which we call *prefix dominance*.

Our corpus of symbolic traces can be thought of as a corpus of execution trees. In fact, in the implementation of our trace generator, the traces only exist at the very last moment. Instead of storing traces, we store a tree that encodes, without unnecessary duplication, the information gained from symbolically executing a procedure. If we think about the dataset of traces as a dataset of trees (each of which holds many traces that share common prefixes), we begin to see that learning word vectors from traces is an approximation of learning directly from the execution trees.

[7] In the following text, and in Fig. 8, we remove the `nouveau_bo_` prefix to conserve space.

Table 2: Top-5 closest words to `affs_bread` and `kzalloc`

`affs_bread`	`kzalloc`
`affs_bread_$NEQ_0`	`kzalloc_$NEQ_0`
`affs_checksum_block`	`kfree`
`AFFS_SB`	`_volume`
`affs_free_block`	`snd_emu10k1_audigy_write_op`
`affs_brelse`	`?->output_amp`

The approximation of trees by traces works, in the sense that we can use the traces to learn meaningful vectors, but the approximation is vulnerable to learning rare behaviors that exist at the beginning of a procedure whose trace-tree has many nested branches. These rare behaviors occur only once in the original procedure text and corresponding execution tree, but are replicated many times in the traces. In a procedure with significant branching complexity, a single occurrence of rare behavior can easily overwhelm any arbitrary number of occurrences of expected behavior.

In Tab. 2, we see two words, `affs_bread` and `kzalloc` , and the five most similar words to each of them. Word similarity has captured many expected relationships. For example, the fact that `kzalloc` is most commonly checked to be non-null (`kzalloc_$NEQ_0`) and then also `kfree` d is what we would expect, given the definition of an allocator. Similarly, we can see that `affs_bread` is also checked to be non-null, checksummed, freed, released, etc. However, in addition to these expected relationships, the last three entries for `kzalloc` seem out of place. These unexpected entries are present in the top-5 answer because of prefix dominance.

We searched our traces for places where `kzalloc` and the three unexpected entries in the table co-occur. We found one function with 5,000 paths (5,000 being our "budget" for the number of traces we are willing to generate via symbolic execution for a single procedure), of which 4,999 have several instances of the pattern `kzalloc` followed by `snd_emu10k1_audigy_write_op` . This one function, with its multitude of paths, overwhelms our dataset, and causes the word vectors to learn a spurious relationship. Prefix dominance also explains the strong associations between `kzalloc` and `_volume` and `?->output_amp` .

On the other hand, `affs_bread` is relatively unaffected by prefix dominance. Examining our traces for the `affs` file system that contains this function, we found that no procedures had an overwhelming number of paths. Therefore, we never see an overwhelming number of `affs_bread` usage patterns that are rare at the source level but common in our set of traces.

5.3 Experiment 3: Queries Via Word-Vector Averaging

Word vectors have the surprising and useful ability to encode meaning when averaged [23, 24]. We devised a test to see if our learned vectors are able to leverage this ability to capture a relationship between allocation failure and returning `-ENOMEM`.

To understand whether our word vectors are capable of answering such a high-level question, we evaluated their performance on increasingly targeted queries (represented by averaged vectors). Each query was restricted to search only for words in the subspace of the embedding space that contains kernel error-codes. (Narrowing to the subspace of error codes ensures that we are only looking at relevant words, and not at the whole vocabulary.)

Results. We identified twenty different functions that act as allocators in the Linux kernel.

First, for each such allocator, we took its word vector $\mathcal{A}$, and queried for the closest vector to $\mathcal{A}$ (in the subspace of error codes). This method found the correct error code only twice out of twenty tests (i.e., 10% accuracy).

Second, we asked for the vector closest to an average vector that combined the vector for the allocator $\mathcal{A}$ of interest and the vector $\overrightarrow{\$ERR}$ for a generic error:[8] $(\mathcal{A} + \overrightarrow{\$ERR})/2$. This query found the correct `ENOMEM` code fourteen times out of twenty (i.e., 70% accuracy).

Third, instead of averaging the allocator's $\mathcal{A}$ vector with $\overrightarrow{\$ERR}$, we tried averaging $\mathcal{A}$ with the vector for the special `$END` token that signals the end of a trace. Seeking the error code closest to $(\mathcal{A} + \overrightarrow{\$END})/2$ found the correct result for sixteen of twenty test cases (i.e., 80% accuracy). The fact that this method outperforms our previous query reveals that the call to an allocator being near the end of a trace is an even stronger signal than the `$ERR` token.

Finally, we mixed the meaning of the allocator, the error token, and the end-of-trace token by averaging all three: $(\mathcal{A} + \overrightarrow{\$ERR} + \overrightarrow{\$END})/3$. The error code whose vector is closest to this query is the correct `ENOMEM` code for eighteen of the twenty tests (i.e., 90% accuracy). The steadily increasing performance indicates that targeted queries encoded as average word vectors can indeed be semantically meaningful.

The effectiveness of these queries, and the results from §5.1 and §5.2, support a positive answer to Research Question 1: learned vectors do encode useful information about program behaviors.

6 RQ2: ABLATION STUDY

In this section, we present the results of an ablation study to isolate the effects that different sets of abstractions have on the utility of the learned vectors. We used the benchmark suite of 19,042 code-analogies from §5 to evaluate eight different configurations. We scored each configuration according to the number of analogies correctly encoded by the word vectors learned for that configuration (i.e., we report top-1 results).

[8] The `$ERR` word is added to any trace that returns either (i) the result of an `ERR_PTR` call, or (ii) a constant less than zero that is also a known error code. Consequently, a vector $\overrightarrow{\$ERR}$ is learned for the word `$ERR`.

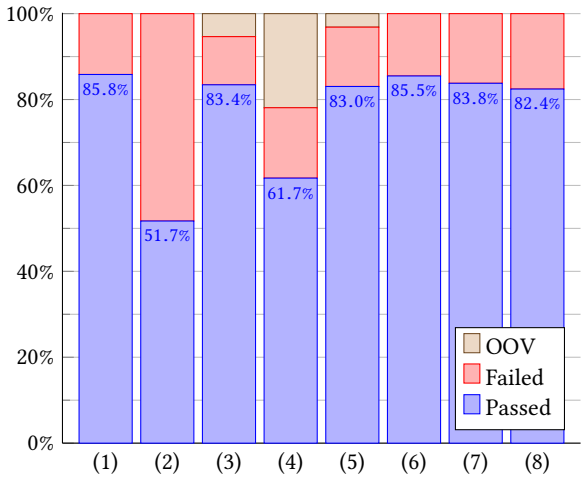

Figure 9: Ablation study: top-1 analogy results for eight configurations (baseline (1) with up to one individual abstraction class removed). The vocabulary minimum was 0, and the number of training iterations was 1,000.

In addition to the baseline configuration from §5.1, we partitioned the abstractions into six classes[9] and generated six new embeddings, each with one of the six abstraction classes excluded. We also used one more configuration in which *stop words* were included. In natural language processing, stop words are words that are filtered out of a processing toolchain. Sometimes these are the most common words in a language, but any group of words can be designated as stop words for a given application. In our context, stop words are function names that occur often, but add little value to the trace. Examples are `__builtin_expect` and automatically generated `__compiletime_assert`s.

We evaluated the following eight configurations:

(1) **baseline**: all abstractions from §3;
(2) baseline without `ParamTo` and `ParamShare`;
(3) baseline without `RetEq`, `RetNeq`, etc.;
(4) baseline without `AccessPathStore` and `AccessPathSensitive`;
(5) baseline without `PropRet`, `RetError`, and `RetConst`;
(6) baseline without `Error`;
(7) baseline without `FunctionStart` and `FunctionEnd`; and
(8) baseline with stop words included.

Fig. 9 compares the accuracy of for these eight configurations. Blue bars indicate the number of tests in the analogy suite that passed; red indicates tests that failed; and brown indicates out-of-vocabulary (OOV) tests. Configuration (4) had the most out-of-vocabulary tests; in this configuration, we do not have words like `!->next` and `!->prev`, which leaves several portions of the analogy suite essentially unanswerable. Thus, we count out-of-vocabulary tests as failed tests.

To create a fair playing field for evaluating all eight configurations, we chose a single setting for the hyper-parameters that were used when learning word vectors. We reduced the threshold for how often a word must occur before it is added to the vocabulary from 1,000 to 0. The latter parameter, which we refer to as

[9] Except for `Called`, which was used in all configurations.

the *vocabulary minimum*, significantly impacts performance by forcing the word-vector learner to deal with thousands of rarely-seen words. To understand why we must set the vocabulary minimum to zero, effectively disabling it, consider the following example trace: `Called(foo), ParamShare(foo, bar), Called(bar)`. In configuration (2), where we ignore `ParamShare`, we would encode this trace as the sentence `foo bar`. In configuration (1), this same trace is encoded as `foo foo bar`. The fact that some abstractions can influence the frequency with which a word occurs in a trace corpus makes any word-frequency-based filtering counterproductive to our goal of performing a fair comparison.

We also lowered the number of training iterations from 2,000 to 1,000 to reduce the resources required to run eight separate configurations of our toolchain. (These changes are responsible for the change in the top-1 accuracy of the baseline configuration from 93.9% in Tab. 1 to 85.8% in Fig. 9.)

In Fig. 9, one clearly sees that configuration (2) (the one without any dataflow-based abstractions) suffers the worst performance degradation. Configuration (4), which omits access-path-based abstractions, has the second-worst performance hit. These results indicate that dataflow information is critical to the quality of learned vectors. This conclusion further confirms findings by Allamanis et al. [4] regarding the importance of dataflow information when learning from programs.

Fig. 9 also reveals that removing "state" abstractions (`RetEq`, `RetNeq`, etc. and `Error`) has little effect on quality. However, these abstractions still add useful terms to our vocabulary, and thereby enlarge the set of potentially answerable questions. Without these abstractions, some of the questions in §5 would be unanswerable.

These results support the following answer to Research Question 2: dataflow-based abstractions provide the greatest benefit to word-vector learning. These abstractions, coupled with access-path-based abstractions, provide sufficient context to let a word-vector learner create useful embeddings. Adding abstractions based on path conditions (or other higher-level concepts like `Error`) adds flexibility without worsening the quality of the learned vectors. Therefore, we recommend including these abstractions, as well.

7 RQ3: SYNTACTIC VERSUS SEMANTIC

Now that we have seen the utility of the generated corpus for word-vector learning (§5) and the interplay between the abstractions we use (§6), we compare our recommended configuration (1) from §6 with a simpler syntactic-based approach.

We explored several options for a syntactic-based approach against which to compare. In trying to make a fair comparison, one difficulty that arises is the amount of data our toolchain produces to use for the semantics-based approach. If we were to compare configuration (1) against an approach based on ASTs or tokens, there would be a large disparity between the paucity of data available to the AST/token-based approach compared to the abundance of data available to the word-vector learner: an AST- or token-based approach would only have one data point per procedure, whereas the path-sensitive artifacts gathered using configuration (1) provide the word-vector learner with hundreds, if not thousands, of data points per procedure.

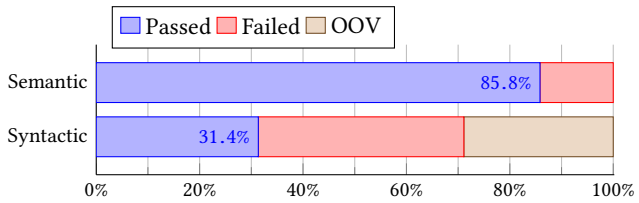

Figure 10: Top-1 analogy results for syntactic versus semantic abstractions. (The vocabulary minimum was 0, and the number of training iterations was 1,000.)

To control for this effect and avoid such a disparity, we instead compared configuration (1) against a configuration of our toolchain that uses only "syntactic" abstractions—i.e., abstractions that can be applied without any information obtained from symbolic execution. Thus, the syntactic abstractions are:

- `FunctionStart` and `FunctionEnd`,
- `AccessPathStore(`*path*`)`, and
- `Called(`*callee*`)`.

The rest of our abstractions use deeper semantic information, such as constant propagation, dataflow information, or the path condition for a given trace.

Using only the syntactic abstractions, we generated a corpus of traces, and then learned word vectors from the corpus. We compared the newly learned word vectors to the ones obtained with configuration (1). Fig. 10 clearly shows that semantic abstractions are crucial to giving the context necessary for successful learning. Even if we assess performance using only the analogies that are in-vocabulary for the syntactic-based approach, we find that the syntactic-based approach achieves only about 44% accuracy, which is *about half* the accuracy of vectors learned from (mainly) semantic abstractions.

These results support an affirmative answer to Research Question 3: abstracted traces that make use of semantic information obtained via symbolic execution provide more utility as the input to a word-vector learner than abstracted traces that use only syntactic information.

8 RQ4: USE IN DOWNSTREAM TASKS

Research Question 4 asks if we can utilize our pre-trained word-vector embeddings on some downstream task.

To address this question, we selected a downstream task that models bug finding, repair, and code completion in a restricted domain: error-code misuse. We chose error-code misuse because it allows us to apply supervised learning. Because there are only a finite number of common error codes in the Linux kernel, we can formulate a multi-class labeling problem using traces generated via our toolchain and our pre-trained word-vector embeddings.

To build an effective error-code-misuse model, we gathered a collection of failing traces (traces in which the `$ERR` token occurs). We then constructed a dataset suitable for supervised learning as follows: we took each trace from configuration (2)[10] and removed the

[10]The dataflow abstractions present in (1) were created to aid word-vector learners; for this experiment, we use configuration (2) to exclude those abstractions.

last three abstract tokens, namely, `$ERR`, `$RET_E*`, and `$END`;[11] we used the `$RET_E*` token as the label for the trimmed trace. We sampled a subset of 20,000 traces from this large trace collection to use for training our model.

This dataset is a good starting point, but feeding it to a machine-learning technique that accepts fixed-length inputs requires further processing. To preprocess the data, we kept only the last 100 tokens in each trace. We then took the trimmed traces, and used our learned word-vector embedding to transform each sequence of words into a sequence of vectors (of dimension 300). If, originally, a trace had fewer than 100 tokens, we padded the beginning of the trace with the zero vector. We paired each of the trimmed and encoded traces with its label (which we derived earlier). Finally, to complete the preprocessing of the dataset we attached a one-hot encoding of the label.

To collect a challenging test set to evaluate our learned model, we turned to real bug-fixing commits applied to the Linux kernel. We searched for commits that referenced an "incorrect return" in their description. In addition, we leveraged Min et al.'s [31] list of incorrect return codes fixed by their JUXTA tool. Next, we generated abstracted symbolic traces both before applying the fixing commit and after. Finally, we kept the traces generated before applying the fix that, after the fix, had changed only in the error code returned. Using this process, we collected 68 traces—from 15 unique functions—that had been patched to fix an incorrect return code.

Using the preprocessed dataset, we trained a model to predict the error code that each trace should return. We used a recurrent neural network with long short-term memory (LSTM) [22]. We evaluated the trained model, using our test set, in two different ways:

(1) *Bug Finding:* we use our learned model to predict the three most likely error codes for each trace in our test set. If a given trace initially ended in the error code `A`, but was patched to return the error code `B`, we check to see if the incorrect `A` error code is absent from our model's top-3 predictions.
(2) *Repair / Suggestion:* we again use the learned model to predict the three most likely error codes for each trace in the test set. This time, we determine the fraction of traces for which the correct error code (i.e., `B`) is present in the top-3 prediction made by the model.

In evaluation (1), we found that the learned model identified an incorrect error code in 57 of our 68 tests. This result is promising, because it suggests that there is enough signal in the traces of encoded vectors to make good predictions that could be used to detect bugs early.

In evaluation (2), we observed that the learned model had a top-3 accuracy of 76.5%, meaning that the correct error code is among our top suggested fixes for more than three fourths of the buggy traces. This result is a strong indicator that the learned vectors and abstracted symbolic traces are rich enough to make high-level predictions that could be used to augment traditional IDEs with predictive capabilities. Such a feature could operate like autocomplete, but with an awareness of what other contributors have done and how their (presumably correct) code should influence new contributions. This feature would be similar to the existing applications of statistical modeling to programming tasks such as autocompletion [2, 10, 34, 38, 45].

These results support an affirmative answer to Research Question 4: our pre-trained word-vector embeddings can be used successfully on downstream tasks. These results also suggest that there are many interesting applications for our corpus of abstracted symbolic traces. Learning from these traces to find bugs, detect clones, or even suggest repairs, are all within the realm of possibility.

9 THREATS TO VALIDITY

There are several threats to the validity of our work.

We leverage a fast, but imprecise, symbolic-execution engine. It is possible that information gained from the detection of infeasible paths and the use of a memory model would improve the quality of our learned vectors. In addition, it is likely that a corpus of interprocedural traces would impact our learned vectors.

We chose to focus our attention on the Linux kernel. It is possible that learning good word-embeddings using artifacts derived from the Linux kernel does not translate to learning good word-embeddings for programs in general. To mitigate this risk, we maximized the amount of diversity in the ingested procedures by ingesting the Linux kernel with all modular and optional code included.

Our analogies benchmark and the tests based on word-vector averaging are only proxies for meaning, and, as such, only serve as an indirect indicator of the quality of the learned word vectors. In addition, we created these benchmarks ourselves, and thus there is a risk that we introduced bias into our experiments. Unfortunately, we do not have benchmarks as extensive as those created throughout the years in the NLP community. Similar to Mikolov et al. [29], we hope that our introduction of a suitable benchmark will facilitate comparisons between different learned embeddings in the future.

10 RELATED WORK

Recently, several techniques have leveraged learned embeddings for artifacts generated from programs. Nguyen et al. [36, 37] leverage word embeddings (learned from ASTs) in two domains to facilitate translation from Java to C#. Pradel and Sen [43] use embeddings (learned from custom tree-based contexts built from ASTs) to bootstrap anomaly detection against a corpus of JavaScript programs. Gu et al. [17] leverage an encoder/decoder architecture to embed whole sequences in their DeepAPI tool for API recommendation. API2API by Ye et al. [51] also leverages word embeddings, but it learns the embeddings from API-related natural-language documents instead of an artifact derived directly from source code.

Moving toward more semantically rich embeddings, DeFreez et al. [12] leverage labeled pushdown systems to generate rich traces which they use to learn function embeddings. They apply these embeddings to find function synonyms, which can be used to improve traditional specification mining techniques. Alon et al. [8] learn from paths through ASTs to produce general representations of programs; in [7] they expand upon this general representation by

[11] We exclude traces that included the `$RET_PTR_ERR` token because these traces do not have an associated error code.

leveraging attention mechanisms. Ben-Nun et al. [9] utilize an intermediate representation (IR) to produce embeddings of programs that are learned from both control flow and data flow information.

Venturing into general program embeddings, there are several recent techniques that approach the problem of embedding programs (or, more generally, symbolic-expressions/trees) in unique ways. Using input/output pairs as the input data for learning, Piech et al. [42] and Parisotto et al. [39] learn to embed whole programs. Using sequences of live variable values, Wang et al. [49] produce embeddings to aid in program repair tasks. Allamanis et al. [4] learn to embed whole programs via Gated Graph Recurrent Neural Networks (GG-RNNs) [26]. Allamanis et al. [5] approach the more foundational problem of finding continuous representations of symbolic expressions. Mou et al. [32] introduce tree-based convolutional neural networks (TBCNNs), another model for embedding programs. Peng et al. [40] provide an AST-based encoding of programs with the goal of facilitating deep-learning methods. Allamanis et al. [3] give a comprehensive survey of these techniques, and many other applications of machine learning to programs.

We are not aware of any work that attempts to embed traces generated from symbolic execution. On the contrary, Fowkes and Sutton [15] warn of possible difficulties learning from path-sensitive artifacts. We believe that our success in using symbolic traces as the input to a learner is due to the addition of path-condition and dataflow abstractions—the extra information helps to ensure that a complete picture is seen, even in a path-sensitive setting.

In the broader context of applying statistical NLP techniques to programs, there has been a large body of work using language models to understand programs [1, 6, 21, 35, 46]; to find misuses [33, 50]; and to synthesize expressions and code snippets [18, 44].

11 EXPERIMENTAL ARTIFACTS

Our `c2ocaml` tool, which performs a source-to-source transformation during the compilation of C projects (to generate inputs to our lightweight symbolic execution engine) is available at `https://github.com/jjhenkel/c2ocaml`.

Our lightweight symbolic execution engine, `lsee`, is also available at `https://github.com/jjhenkel/lsee`.

Additionally, we provide tools to demonstrate our experiments at `https://github.com/jjhenkel/code-vectors-artifact` [19]. This artifact allows the user to run our toolchain end-to-end on a smaller open-source repository (Redis). The artifact uses pre-built Docker [28] images to avoid complex installation requirements. Our raw data (two sets of learned vectors and a full collection of abstracted symbolic traces) are also included in this artifact.

12 CONCLUSION

The expanding interest in treating programs as data to be fed to general-purpose learning algorithms has created a need for methods to efficiently extract, canonicalize, and embed artifacts derived from programs. In this paper, we described a toolchain for efficiently extracting program artifacts; a parameterized framework of abstractions for canonicalizing these artifacts; and an encoding of these parameterized embeddings in a format that can be used by off-the-shelf word-vector learners.

Our work also provides a new benchmark to probe the quality of word-vectors learned from programs. Our ablation study used the benchmark to provide insight about which abstractions contributed the most to our learned word vectors. We also provided evidence that (mostly) syntactic abstractions are ill-suited as the input to learning techniques. Lastly, we used these tools and datasets to learn a model of a specific program behavior (answering the question, "Which error is a trace likely to return?"), and applied the model in a case study to confirm actual bugs found via traditional static analysis.

ACKNOWLEDGMENTS

This research was supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by ONR under grant N00014-17-1-2889; by NSF under grants CCF-1318489, CCF-1420866, and CCF-1423237; and by the UW–Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[3] Miltiadis Allamanis, Earl T Barr, Premkumar T Devanbu, and Charles A Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *CoRR* abs/1709.0 (2017). arXiv:1709.06182 http://arxiv.org/abs/1709.06182

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.0 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[5] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. 2016. Learning Continuous Semantic Representations of Symbolic Expressions. *arXiv preprint arXiv:1611.01423* (2016).

[6] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 2123–2132. http://dl.acm.org/citation.cfm?id=3045118.3045344

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *CoRR* abs/1803.09473 (2018). arXiv:1803.09473 http://arxiv.org/abs/1803.09473

[8] 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

[9] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* abs/1806.07336 (2018). arXiv:1806.07336 http://arxiv.org/abs/1806.07336

[10] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2933–2942. http://dl.acm.org/citation.cfm?id=3045390.3045699

[11] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 238–252.

[12] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-Based Function Embedding and its Application to Specification Mining. *CoRR* abs/1802.07779 (2018). arXiv:1802.07779 http://arxiv.org/abs/1802.07779

[13] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. 1980. *Programming environments based on structured editors: The MENTOR experience.* Technical Report 26. INRIA.

[14] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing Search in Context: The Concept Revisited. In *Proceedings of the 10th International Conference on World Wide Web (WWW '01)*. ACM, New York, NY, USA, 406–414. https://doi.org/10.1145/371920.372094

[15] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

[16] H. Ganzinger and N.D. Jones (Eds.). 1986. *Proc. Programs as Data Objects*. LNCS, Vol. 217.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[18] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 416–432. https://doi.org/10.1145/2814270.2814295

[19] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Artifact for Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. https://doi.org/10.5281/zenodo.1297689

[20] Felix Hill, Roi Reichart, and Anna Korhonen. 2015. Simlex-999: Evaluating Semantic Models with Genuine Similarity Estimation. *Comput. Linguist.* 41, 4 (Dec. 2015), 665–695. https://doi.org/10.1162/COLI_a_00237

[21] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[23] Tom Kenter, Alexey Borisov, and Maarten de Rijke. 2016. Siamese CBOW: Optimizing Word Embeddings for Sentence Representations. In *Proceedings of the The 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*.

[24] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, II–1188–II–1196. http://dl.acm.org/citation.cfm?id=3044805.3045025

[25] Omer Levy and Yoav Goldberg. 2014. Linguistic Regularities in Sparse and Explicit Word Representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Ann Arbor, Michigan, 171–180. http://www.aclweb.org/anthology/W/W14/W14-1618

[26] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S Zemel. 2015. Gated Graph Sequence Neural Networks. *CoRR* abs/1511.0 (2015). arXiv:1511.05493 http://arxiv.org/abs/1511.05493

[27] Thang Luong, Richard Socher, and Christopher D Manning. 2013. Better Word Representations with Recursive Neural Networks for Morphology. In *CoNLL*.

[28] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger (Eds.). Curran Associates, Inc., 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[31] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[32] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775/11735

[33] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3106237.3106284

[34] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2012. GraPacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1407–1410. http://dl.acm.org/citation.cfm?id=2337223.2337431

[35] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 858–868. http://dl.acm.org/citation.cfm?id=2818754.2818858

[36] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API Elements for Code Migration with Vector Representations. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 756–758. https://doi.org/10.1145/2889160.2892661

[37] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 438–449. https://doi.org/10.1109/ICSE.2017.47

[38] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 532–542. https://doi.org/10.1145/2491411.2491458

[39] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. *CoRR* abs/1611.0 (2016). arXiv:1611.01855 http://arxiv.org/abs/1611.01855

[40] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building Program Vector Representations for Deep Learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403 (KSEM 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 547–553. https://doi.org/10.1007/978-3-319-25159-2_49

[41] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[42] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1093–1102. http://dl.acm.org/citation.cfm?id=3045118.3045235

[43] Michael Pradel and Koushik Sen. 2017. *Deep Learning to Find Bugs*. Technical Report TUD-CS-2017-0295. Technische Universität Darmstadt, Department of Computer Science.

[44] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 357–367. https://doi.org/10.1145/2884781.2884808

[45] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". *Popl* (2015). https://doi.org/10.1145/2676726.2677009

[46] 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

[47] Herbert Rubenstein and John B Goodenough. 1965. Contextual Correlates of Synonymy. *Commun. ACM* 8, 10 (Oct. 1965), 627–633. https://doi.org/10.1145/365628.365657

[48] Sean Szumlanski, Fernando Gomez, and Valerie K Sims. 2013. A new set of norms for semantic relatedness measures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 890–895.

[49] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *CoRR* abs/1711.07163 (2017). arXiv:1711.07163 http://arxiv.org/abs/1711.07163

[50] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug Detection with N-gram Language Models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 708–719. https://doi.org/10.1145/2970276.2970341

[51] X Ye, H Shen, X Ma, R Bunescu, and C Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1145/2884781.2884862

[52] Geoffrey Zweig and Chris J C Burges. 2011. *The Microsoft Research Sentence Completion Challenge*. Technical Report. https://www.microsoft.com/en-us/research/publication/the-microsoft-research-sentence-completion-challenge/