Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs

Anonymous Author(s)

ABSTRACT

When working on techniques to address the wide-spread problem of software bugs, one often faces the need for a large number of realistic bugs in real-world programs. Such bugs can either help evaluate an approach, e.g., in form of a bug benchmark or a suite of program mutations, or even help build the technique, e.g., in learning-based bug detection. Because gathering a large number of real bugs is difficult, a common approach is to rely on automatically seeded bugs. Prior work seeds bugs based on syntactic transformation patterns, which often results in unrealistic bugs and typically cannot introduce new, application-specific code tokens. This paper presents SemSeed, a technique for automatically seeding bugs in a semantics-aware way. The key idea is to imitate how a given real-world bug would look like in other programs by semantically adapting the bug pattern to the local context. To reason about the semantics of pieces of code, our approach builds on learned token embeddings that encode the semantic similarities of identifiers and literals. Our evaluation with real-world JavaScript software shows that the approach effectively reproduces real bugs and clearly outperforms a semantics-unaware approach. The seeded bugs are useful as training data for learning-based bug detection, where they significantly improve the bug detection ability. Moreover, we show that SemSeed-created bugs complement existing mutation testing operators, and that our approach is efficient enough to seed hundreds of thousands of bugs within an hour.

1 INTRODUCTION

Bugs are one of the key challenges in software development, and various techniques have been proposed for bug detection, bug fixing, and bug prevention. A common problem faced when working on bug-related techniques is the need for large amounts of known, realistic bugs. Such bugs can serve multiple purposes. One of them is to provide a benchmark for evaluating and comparing bug-related tools. For example, static bug detectors and fuzz testing tools are evaluated against sets of known bugs [10, 15, 19], and bugs created via mutations are useful for evaluating the effectiveness of test suites [21]. Unfortunately, real bugs are scarce and without precise knowledge about where exactly a bug is, assessing whether a problem reported by a tool is indeed a bug, requires manual effort. As a result, many tool evaluations are limited to a small number (typically, a few dozens) of bugs, e.g., bugs manually gathered from open-source projects [24, 51].

Another purpose of known bugs is to help build a bug-related technique. For example, learning-based bug detectors [33, 34, 46], defect prediction models [60], and repair tools [5, 32] rely on bugs to learn from. These techniques require large amounts of training data, typically in the form of code known to contain a (specific kind of) bug. Since obtaining large amounts of bugs is non-trivial, current techniques either focus on bugs created through simple code transformations [46], on noisy datasets that, e.g., approximate buggy code as any code changed in the next version of a program [60], on manually curated bug datasets [24, 51], or on code changes that are heuristically linked with bug reports [33].

Table 1: Comparison with other bug seeding techniques.

Approach	Kinds of bugs	Target locations (C1)	Adaptation to target location (C2)	Unbound tokens (C3)
Mutation operators [23, 40]	Few, manually defined	Everywhere	Syntactic	Not supported
Inferred mutat. operators [9]	Many, inferred	Everywhere	Syntactic	Not supported
Neural machine translation [58]	Many, inferred	Implicit by model	Implicit by model	Not supported
Bug synthesis [15, 50]	Memory updates	Hard to trigger paths	N/A	N/A
This work	Many, inferred	Based on semantic fit	Semantic	Supported

This paper presents SemSeed, which addresses the need for large amounts of known, realistic bugs through a semantics-aware bug seeding technique. The key idea is to generalize a bug observed in the past and to seed variants of the bug at other code locations. To reason about the semantics of code, we exploit token embeddings [8, 59], a learned representation of code elements, such as identifier names and literals. To the best of our knowledge, we are the first to use learned embeddings for bug seeding.

SemSeed addresses three important challenges not sufficiently considered in previous work. (C1—Where) *Where in a target program to seed* bugs that resemble a given bug-to-imitate? We address this challenge by checking which locations in the target program semantically fit the bug-to-imitate. (C2—How) *How to adapt the bug-to-imitate to the target program*? SemSeed addresses this challenge by semantically adapting identifiers and literals to the target location. (C3—Unbound tokens) How to handle tokens in the buggy code that do not occur in the correct code, e.g., when the buggy code refers to an application-specific identifier name or literal? We address this challenge, called *unbound tokens*, through semantic analogy queries in the token embedding space that find a token that resembles the bug-to-imitate but fits the bug seeding location.

Table 1 summarizes and contrasts SemSeed with other work on automatically seeding bugs. First, mutation testing [23, 40] seeds bugs based on pre-defined code transformations. However, mutation operators cover only a small set of the syntactic transformations that occur in the wild and only sometimes represent real-world bugs [18]. Second, some work infers mutation operators from past bug fixes [9]. Both pre-defined and inferred mutation operators are applied in a purely syntactic way, without considering whether a code transformation semantically fits a code location (C1) or how

to adapt the transformation to the location (C2). Third, neural models can learn from past bug fixes how to inject bugs [58]. Such approaches implicitly select target locations for seeding bugs and adapt the seeding to these target locations, but the details are hidden within the neural network. Finally, work aimed at evaluating fuzz testing tools [15, 50] seeds bugs along execution paths that are non-trivial to trigger. Even though these bugs may appear realistic from an execution perspective, they are easy to detect statically, making the approach unfit for evaluating or training static bug detectors. None of the above approaches addresses the problem of unbound tokens (C3), which our evaluation shows to prevent them from seeding the majority of bugs that appear in the wild.

We evaluate SemSeed by learning from real-world bugs and by seeding hundreds of thousands of new bugs. The evaluation focuses on JavaScript, because it has become one of the most popular languages and is used in various domains, but the approach is not specific to this language. The results show that SemSeed is effective at creating realistic bugs, that the seeded bugs complement bugs created with traditional mutation operators [40], and that our implementation can seed hundreds of thousands of bugs within an hour. Using the seeded bugs as training data for a learning-based bug detector [46] significantly improves the bug detection ability compared to the state of the art.

In summary, this paper makes the following contributions:

- We are the first to *use learned token embeddings for bug seeding*.
- We present a *semantics-aware technique* to decide where to seed a bug, how to adapt a given bug-to-imitate to the target location, and how to handle unbound tokens.
- We present an *efficient algorithm* for semantic bug seeding, which chooses from thousands of candidate bugs the semantically most suitable within about 0.01 seconds, on average.
- We show *empirical evidence* that SemSeed seeds realistic bugs, outperforms a purely syntactic bug seeding technique, complements traditional mutation operators, and yields bugs useful for training more effective bug detection models.

2 OVERVIEW

This section illustrates the key ideas of our approach with an example. At a high level, SemSeed consists of three main steps: abstraction, semantic matching, and pattern application. Given a set of concrete bug fixes, e.g., gathered from version histories, the first step abstracts away project-specific details, such as the identifier names. This results in bug seeding patterns that describe how to syntactically transform a piece of code to introduce a new bug.

The top part of Figure 1 shows one concrete bug fix that the approach takes as an input. The middle part of the figure shows the corresponding bug seeding pattern. The concrete identifiers, e.g., `process` and `platform` are abstracted based on their syntactic category, e.g., into `id1` and `id2`. Intuitively, the bug pattern could be described as "wrong comparison with wrong literal".

The second step of the approach matches the inferred bug seeding patterns with a given target program, addressing challenge C1. A naive baseline approach would apply a pattern at every syntactically matching location. For our example target program in Figure 1, the "wrong comparison with wrong literal" pattern could be applied at every binary expression that compares some `id1.id2` with some

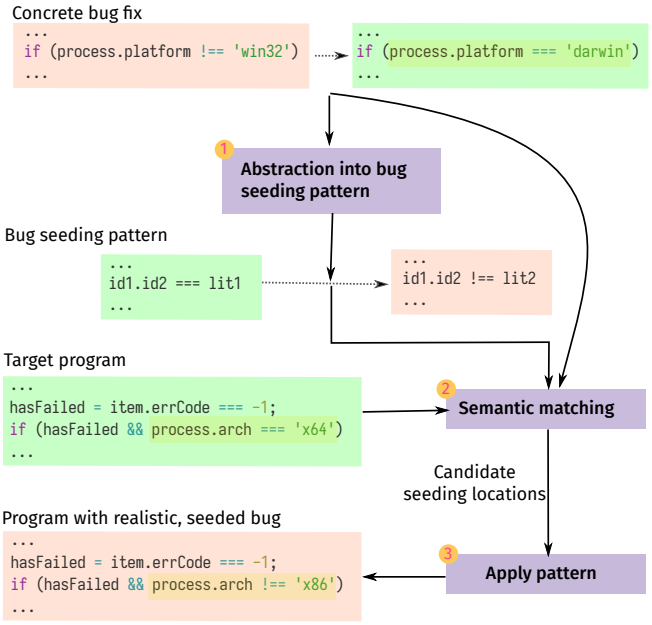


Figure 1: Overview of the approach and running example.

`lit1` using `===`. However, such a purely syntactic approach will lead to a large number of unrealistic bugs.

A key idea of SemSeed is to not apply patterns at every syntactically matching code location, but only at locations that are semantically similar to the locations where a pattern was derived from. For the example in Figure 1, SemSeed's semantic matching component may select the code location `process.arch === 'x64'` because it also is about checking whether some platform matches a string descriptor of a platform. The challenge in finding such a location is that the semantic similarity may not be obvious to a program analysis that is unaware of domain knowledge. For our example, the approach needs to understand that the identifiers `platform` and `architecture` refer to similar concepts, and that the literals `"darwin"` and `"x64"` both describe platforms.

The third and final step of SemSeed applies bug seeding patterns at the candidate seeding locations identified by the second step, addressing challenges C2 and C3. The key idea is to adapt the syntactic bug seeding pattern to the selected location in the target program. Specifically, the approach instantiates the pattern with identifiers and literals that are semantically similar to a location where the pattern was derived from (C2), and it finds suitable tokens for identifiers not present in the original target program (C3). As a result, the approach semantically generalizes the given concrete bugs to other code locations, which yields fewer, but more realistic seeded bugs than syntactic bug seeding. To determine how similar two code locations are, we rely on neurally learned embeddings of identifier names and literals, which have been used for other program analysis tasks in the past [3, 38, 43, 46], but to the best of our knowledge have not yet been used for bug seeding.

To seed a bug at the selected location, SemSeed transforms the code as shown at the bottom part (target program) of the figure. This transformation not only instantiates the bug seeding pattern, but it also picks a suitable platform descriptor, `"x86"`. The approach picks this literal based on identifiers and literals used in the vicinity of the bug seeding location, mimicking a mistake a developer might

also make. As a result, the seeded bug resembles the original bug that the pattern was derived from, while adapting it to the local context.

3 APPROACH

This section presents the three steps of our SemSeed approach in detail. At first, Section 3.1 describes how to extract bug seeding patterns from concrete bug fixes in version histories. Then, Section 3.2 presents how our approach semantically matches these bug seeding patterns against previously unseen code to find candidate locations for seeding new bugs. Finally, Section 3.3 describes how to apply the patterns to a given code location by semantically adapting them.

3.1 Abstraction into Bug Seeding Patterns

The first step of SemSeed analyzes bug-fixing code changes in the version histories of popular code repositories to generalize them into bug seeding patterns. The basic idea is that reverting and generalizing a code change that fixes a bug will yield a pattern that we can then use to introduce this kind of bug into other code.

3.1.1 Selecting Bug-Fixing Commits. To gather examples of bug-fixing code changes, we mine the commit histories of code repositories. For a given repository, SemSeed filters all commits based on four criteria, which are designed to identify simple, bug-fixing commits. First, we select only those commits where the commit message contains any one of the words "bug", "fix", "error", "issue", "problem", and "correct", which we assume to indicate that the commit is fixing a bug. Second, we select only those commits that have a single parent commit, to avoid merged commits. Third, we select commits where the number of changed files is one and where the changed file is written in the target programming language, i.e., JavaScript in this paper. Finally, our fourth criterion is to omit commits where the number of changed lines is higher than one. Both the third and the fourth criterion help with omitting commits that fix more than a single bug or that intermingle a bug fix with other code changes. Prior work shows single-line bugs to be relevant and frequent in practice [16, 26, 48].

Since identifying bug-fixing code changes is a non-trivial problem, our four filters are designed to rather exclude some bug-fixing commits than to include many other commits. Of course, there is no guarantee that the commits obtained using these four filters are bug-fixing code changes. Manual inspection by previous research [58] of commits mined with a similar approach [illegible]own 97% of their commits to be bug-fixing.

3.1.2 Extracting Concrete Changes From Commits. Given a set of bug-fixing code commits, SemSeed next extracts code changes into a format suitable for the remainder of the approach. Due to our filtering of commits, each commit changes exactly one line of code. One option would be to consider the entire changed line, which may, however, include parts unrelated to the bug fix. Including such "noise" would make it harder to identify recurring patterns and to find suitable locations for seeding bugs based on these patterns. Another option would be to consider only those tokens of the line that have been modified, which may, however, miss surrounding tokens important to capture the context of the change. Including some contextual information helps SemSeed identify the most suitable locations for seeding bugs with a given pattern.

To extract the changed tokens along with some context, SemSeed uses the AST of the old and the new file to find a subsequence of the changed line's tokens that forms a complete syntactic entity. Focusing on complete syntactic entities, instead, e.g., on all tokens in a changed line, increases the chance to find recurring patterns. To this end, the approach converts both the buggy and the correct file into ASTs and maps each AST node to its corresponding range of line numbers in the file. Next, the approach prunes all AST nodes that do not comprise any changed line. From the remaining nodes of a file, SemSeed selects one of the nodes with the maximum number of source code tokens in the changed line, and then emits the sequence of tokens rooted at this node. The result is two sequences of tokens, for the buggy and correct files, respectively:

Definition 1. A *concrete bug fix* (C_{bug}, C_{corr}) is a pair of sequences of tokens, where the sequence $C_{bug} = [t^b_1, ..., t^b_m]$ corresponds to a subtree in the AST of the buggy file, and the sequence $C_{corr} = [t^c_1, ..., t^c_n]$ corresponds to a subtree in the AST of the corrected file.

For example, consider the concrete bug fix in Figure 1. Analyzing the modified line in the buggy file (shown in red, on the top-left), SemSeed selects the AST subtree that represents the `process.platform !== 'win32'` expression and hence yields the tokens in this expression. For the correct file (shown in green, on the top-right), the analysis yields the tokens in `process.platform === 'darwin'`. Even though both extracted token sequences correspond to the same kind of AST subtree in this example, the sequences may correspond to different syntactic entities in general.

3.1.3 From Concrete Fixes to Bug Seeding Patterns. To enable SemSeed to seed new bugs based on the extracted concrete bug fixes, the approach generalizes bug fixes into bug seeding patterns. During this step, we abstract identifier tokens and literal tokens. The rationale is that these kinds of tokens often are application-specific and hence must be adapted to a specific bug seeding location.

Definition 2. A *bug seeding pattern* (P_{corr}, P_{bug}) is a pair of sequences of tokens, where a token is either id_k or lit_k (for some $k \in \mathbb{N}$), or a non-identifier and non-literal token of the target programming language.

Our approach for abstracting a concrete bug fix into a bug seeding pattern starts from the token sequence in the correct part of the change, i.e., C_{corr}. The algorithm traverses all tokens in the concrete bug fix and abstracts all identifiers and literals into placeholders id_i and lit_j, where i and j are incremented whenever a new identifier or literal occurs. To consistently abstract tokens that occur multiple times, the algorithm maintains for each concrete bug fix a map M from concrete to abstract tokens [56]. Finally, we discard concrete bug fixes that have more than 40 tokens and that occur only once, which discards about 15% of all bug seeding patterns, to avoid learning obscure patterns unlikely to apply anywhere else.

For our running example, the middle part of Figure 1 shows the bug seeding pattern. The algorithm abstracts the identifiers `process` and `platform` into id_1 and id_2, respectively, and the literals `'win32'` and `'darwin'` into lit_1 and lit_2, respectively.

3.2 Matching Bug Seeding Patterns against Code

Based on the inferred bug seeding patterns, the second step of SemSeed is to find code locations for seeding the bug defined by the pattern into a target program. The approach matches the correct part of a pattern against token sequences extracted from the target program. We call the matching token sequences *candidate seeding locations*. One key contribution of SemSeed is to determine candidate seeding locations not only by syntactically matching a pattern against the target program, but also by semantically reasoning about the similarity of the involved identifiers and literals.

3.2.1 Extracting Token Sequences from Target Program. Given a target program where SemSeed should seed bugs, the approach extracts various token sequences to match against the inferred bug seeding patterns. Similar to the pattern extraction step, SemSeed starts by parsing the target program into an AST and then extracts sequences of tokens that correspond to subtrees of the AST. Given a node and its corresponding token sequence $C = [t_1, \ldots, t_n]$, the approach applies the same abstraction algorithm as in Section 3.1.3 to get the abstracted token sequence C_{abstr}.

For example, consider the target program in Figure 1. The approach extracts multiple AST subtrees and corresponding token sequences. Two of the extracted subtrees represent binary expressions, and the corresponding token sequences are [`item`, `.`, `errCode`, `===`, `-1`] and [`process`, `.`, `arch`, `===`, `'x64'`]. For both sequences, abstracting identifiers and literals results in the abstracted token sequence [`id1`, `.`, `id2`, `===`, `lit1`].

3.2.2 Syntactic Matching. Given a set of token sequences extracted from the target program, SemSeed matches each abstracted token sequence against each bug seeding pattern. For a sequence C_{abstr} and a pattern (P_{corr}, P_{bug}), the approach checks whether C_{abstr} matches P_{corr}, i.e., the correct part of the pattern. As a first step, SemSeed performs a simple *syntactic matching*, where C_{abstr} and (P_{corr}, P_{bug}) match if C_{abstr} is equal to P_{corr}. Note that the syntactic matching is similar to a corresponding step in previous work on seeding bugs with inferred mutation operators [9].

For our example, the two token sequences abstracted into [`id1`, `.`, `id2`, `===`, `lit1`] are both equal to the correct part of the bug seeding pattern from Section 3.1. Hence, both binary expressions in the target program are retained as candidate seeding locations.

3.2.3 Semantic Matching. Syntactically matching bug seeding patterns against a target program yields a large number of candidate seeding locations. Unfortunately, seeding bugs at all these locations would result in many seeded bugs that do not semantically resemble the concrete bugs that SemSeed is learning from. For example, applying the bug seeding pattern of our running example both to `item.errCode === -1` and to `process.arch === 'x64'` would yield two seeded bugs. However, only the second seeded bug would be semantically similar to the concrete bug that the pattern was learned from: The bug is about incorrectly checking the name of a platform against a string that describes a platform, `process.platform === 'darwin'`, and a similar bug could occur in the `process.arch === 'x64'` expression. In contrast, the other possible candidate seeding location `item.errCode === -1` matches the original bug only syntactically, but not semantically.

Algorithm 1 Semantically match a token sequence against a bug seeding pattern.

Input: Token sequence C and concrete bug fix (C_{bug}, C_{corr})
Output: *True* if C is a semantic match, *False* otherwise

```
[t_1, ..., t_n] ← C                          ▷ Tokens of target location
[t'_1, ..., t'_n] ← C_corr                   ▷ Tokens where real bug occurred
S ← []
for i = 1 to n do
    if kind(t_i) ∈ {Identifier, Literal} then
        v ← emb(t_i)
        v' ← emb(t'_i)
        Append simil(v, v') to S
return avg(S) >= matching threshold m
```

To ensure that SemSeed seeds realistic bugs, the approach focuses on bugs that semantically resemble a given concrete bug fix. Checking whether two code locations are semantically similar is a hard problem, which we address by borrowing ideas from machine learning-based natural language processing (NLP). In NLP, an important research problem is to identify semantically similar words, such as "chicken" and "fowl". A state-of-the-art approach to address this problem is *word embeddings* learned from a corpus of text, e.g., using Word2vec [39] or FastText [8]. An embedding maps each word into a real-valued vector so that semantically similar words have similar vectors. For example, the word vectors of "chicken" and "fowl" will be close to each other in the embedding space. To determine how similar two word vectors v, w are in an embedding space, we compute their cosine similarity: $simil(v, w) = \frac{v \cdot w}{\|v\| \|w\|}$

SemSeed computes embeddings of source code tokens and uses them to reason about the semantic similarity of tokens. As the embedding technique, we build on FastText [8], which we choose for two reasons. First, FastText does not suffer from the out-of-vocabulary problem [27], because it reasons about the n-grams in a token instead of relying on a fixed-size vocabulary. Second, FastText has been shown to more accurately represent the semantic similarities of code tokens than other popular embeddings [59].

Algorithm 1 summarizes how SemSeed checks whether a given token sequence semantically matches a bug seeding pattern. To this end, the approach semantically compares the concrete tokens C_{corr} where the bug described by the pattern has occurred with the tokens C in the target program. The algorithm computes for each identifier and literal token in C its semantic similarity to the corresponding token in C_{corr}. If the average similarity for all tokens in C exceeds a threshold m, which we call the *matching threshold*, then the algorithm returns $True$, and SemSeed marks C as a candidate seeding location. Averaging across embeddings of individual tokens is inspired by work on representing natural language sentences and documents [4, 30]. For bug seeding patterns derived from multiple concrete bug fixes, the approach invokes the algorithm multiple times, and considers C a candidate seeding location if C resembles at least one of the concrete bug fixes. Our evaluation studies the impact of the matching threshold m in practice.

For the example, SemSeed invokes Algorithm 1 for the two syntactically matching code locations. The first invocation, where C contains the tokens in `item.errCode === -1`, is likely to return

False (depending on the matching threshold) because the compared tokens, e.g., `item` vs. `process`, or `'darwin'` vs. `-1`, are dissimilar. In contrast, the second invocation is likely to return *True* because the tokens in `process.arch === 'x64'` have a high pairwise similarity with the tokens in `process.platform === 'darwin'`.

3.3 Applying Bug Seeding Patterns

The third and final step of SemSeed is to apply bug seeding patterns at the bug seeding locations in the target program.

3.3.1 Unbound Tokens. The main challenge in this step is tokens that appear in the buggy part but not in the correct part of the pattern, which we call *unbound tokens*. For example, recall the bug seeding pattern in Figure 1, and in particular, the `'lit2'` token in the buggy code. When applying the pattern to a program, this token is unbound, i.e., it is unclear what concrete token to insert instead of `'lit2'`. Prior work on automatically seeding bugs based on inferred bug patterns [9, 58] ignores the problem of unbound tokens, and hence can apply only bug seeding patterns without any such tokens. However, as we show in our evaluation, the majority of all bug seeding patterns contains unbound tokens, i.e., ignoring them would ignore many bug seeding opportunities.

Before presenting our approach for applying bug seeding patterns with unbound tokens, we consider a few alternatives. Suppose we want to apply a bug seeding pattern (P_{corr}, P_{bug}), which was inferred from a concrete bug fix (C_{bug}, C_{corr}) and has an unbound token $t_?$. The question is which concrete token to use instead of $t_?$ when concretizing P_{bug} in the target program.

One option would be to replace $t_?$ with the concrete token it is bound to in C_{bug}. However, this token may not be a natural fit for the target program. For example, when $t_?$ is an identifier, then simply replacing it with the corresponding identifier from C_{bug} is likely to result in an undefined variable, resulting in a rather unrealistic bug. Another option would be to replace $t_?$ with a random token sampled from the vocabulary of all tokens, which again is likely to result in an unrealistic bug. A third option would be to sample a token from all tokens in the same file or same function as the bug seeding location. While this approach increases the chance of resulting in realistic code, it is still likely to yield a token that does not fully fit the context of the bug seeding location, e.g., because it uses a variable of a wrong type.

3.3.2 Binding Tokens via Analogy Queries. To address the challenge of binding an unbound token in a way that fits the bug seeding location, and hence ultimately, create a realistic bug, we again take inspiration from NLP. Given a learned word embedding, word analogy tasks intend to answer similarity questions involving two or more pairs of words. For example, one may ask the analogy question *What word is to "France" what "Tokyo" is to "Japan"?*, which is likely to yield the answer *"Paris"* [39]. Adapting this idea to unbound tokens, SemSeed uses the bound tokens of a bug seeding pattern to resolve any unbound tokens in the same pattern.

Figure 2 illustrates our analogy-based technique for binding unbound tokens using the example in Figure 1. Given the bug seeding pattern, the token `lit2` is unbound. In contrast, `process` and `arch` are bound, because `id1` and `id2` occur both in the correct and the buggy part of the pattern. SemSeed searches for a suitable token for `lit2` by asking three analogy questions:

- *What token is to `'x64'` what `'win32'` is to `'darwin'`?*
- *What token is to `arch` what `'win32'` is to `platform`?*
- *What token is to `process` what `'win32'` is to `process`?*

The token embedding answers these questions by returning the three blue points in the vector space (we explain below how exactly these points are computed), and SemSeed combines the answers by averaging these three points. This average, shown as a pink point can be thought of as a target location in the embedding space. SemSeed retrieves a suitable token for `lit2` by searching the nearest neighbor of the target location, as indicated by the gray sphere in Figure 2. The nearest neighbor in our example is the literal token `'x86'`, and hence SemSeed seeds a bug using this token.

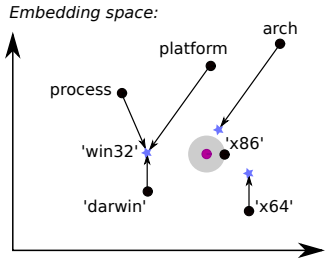


Figure 2: Example of analogy queries to bind unbound tokens.

Algorithm 2 Apply bug seeding pattern to candidate token sequence

Input: A candidate token sequence C, a concrete bug fix (C_{bug}, C_{corr}) and its corresponding bug seeding pattern (P_{corr}, P_{bug}), a set T of identifier and literal tokens.
Output: Tokens C_{seed} of seeded bug

$C_{seed} \leftarrow [\,]$
for $i \leftarrow 1$ **to** $length(C_{bug})$ **do**
  if $kind(C_{bug}[i]) \notin \{Identifier, Literal\}$ **then**
    Append $C_{bug}[i]$ to C_{seed} ▷ Copy token
  else if $P_{bug}[i]$ bound to t_{bound} **then**
    Append t_{bound} to C_{seed} ▷ Use bound token
  else
    $V_{tgt} \leftarrow \varnothing$ ▷ Bind token via analogy queries
    for $t_{abstr} \in P_{corr}$ **do**
      $t_{orig} \leftarrow$ token that t_{abstr} is bound to in C_{bug}
      $t_{seed} \leftarrow$ token that t_{abstr} is bound to in C
      $v_{tgt} \leftarrow emb(t_{seed}) + emb(C_{bug}[i]) - emb(t_{orig})$
      Add v_{tgt} to V_{tgt}
    $t_? \leftarrow \arg\max_{t \in T} simil(emb(t), avg(V_{tgt}))$
    Append $t_?$ to C_{seed}

3.3.3 Algorithm. After providing an intuition of the approach, Algorithm 2 presents in detail how SemSeed applies a bug seeding pattern. The algorithm takes three inputs. First, a candidate token sequence C, identified as described in Section 3.2, which the algorithm will mutate to seed a bug. Second, a concrete bug fix (C_{bug}, C_{corr}) and its corresponding bug seeding pattern (P_{corr}, P_{bug}). The algorithm will seed a new bug by imitating the given bug and by

semantically adapting it to the context of the candidate token sequence. Third, a set T of literal and identifier tokens from which the algorithm selects tokens to use for unbound tokens. For example, this set may consist of all identifiers and literals in the file where the bug is seeded or the n most common tokens in a corpus of code. Our evaluation compares different ways of computing the set T.

The main loop of the algorithm goes through all tokens in the bug-to-imitate C_{bug}, and iteratively builds a new sequence C_{seed} of buggy tokens. For each token to generate, the algorithm distinguished three cases. The first case (line 3) handles tokens that are neither identifiers nor literals, but standard tokens of the programming language, such as operators or parentheses. Each such token is directly copied from the bug-to-imitate into C_{seed}. The second case (line 5) handles bound identifier and literal tokens, i.e., tokens that appear in the candidate token sequence. The algorithm uses the concrete token t_{bound} from the candidate token sequence and appends it to C_{seed}. For our running example, the first two cases handle the tokens `process`, `.`, `arch`, and `!==`. These cases are sufficient to handle bug seeding patterns without any unbound tokens, where it suffices to rearrange the tokens in the candidate token sequence into the buggy token sequence. In contrast, unbound tokens, such as `lit2` in our example, require including a new token into the sequence C_{seed}.

The third case (line 7 in Algorithm 2) handles unbound tokens by computing a set V_{tgt} of target points in the vector space of the token embedding. For each abstract token t_{abstr} that appears in the correct part P_{corr} of the bug seeding pattern, the algorithm computes a target point based on the concrete tokens t_{orig} and t_{seed} that t_{abstr} is bound to in the bug-to-imitate and the candidate token sequence, respectively. The target point is computed at line 12, which implements an analogy query. The query starts from the embedding of t_{seed} and adapts it by adding the vector that leads from the embedding of t_{orig} to the corresponding token $C_{bug}[i]$ in the bug-to-imitate. For our running example, the algorithm computes three target locations (which correspond to the three analogy questions from above):

$$\begin{aligned} V_{tgt} = \{&emb(\texttt{'x64'}) + emb(\texttt{'win32'}) - emb(\texttt{'darwin'}),\\ &emb(\texttt{arch}) + emb(\texttt{'win32'}) - emb(\texttt{platform}),\\ &emb(\texttt{process}) + emb(\texttt{'win32'}) - emb(\texttt{process})\} \end{aligned}$$

That is, the algorithm finds the difference between the vectors of `'darwin'` and `'win32'` and adds it to the vector of `'x64'`, and similar for the other two queries. The resulting three target locations are the blue points in Figure 2.

Given the set V_{tgt}, the algorithm queries T for the token whose embedding is most similar to the average of all target points. Intuitively, this token is the available token that is semantically closest to the token observed in the bug-to-imitate. Once retrieved, the algorithm adds the token to the sequence C_{seed} of result tokens. Our implementation uses a variant of Algorithm 2, which binds unbound tokens not only with the available token that is most similar to the average of the target points, but to consider the k nearest neighbors of the average. For a given candidate token sequence and bug-to-imitate, this variant seeds not only one but k bugs.

To avoid breaking the syntactic correctness of target programs, SemSeed checks for each seeded bug whether it yields syntactically correct code by parsing the complete file after seeding the bug. For example, a bug seeding pattern where the correct part is `[id1, =, lit1]` and the buggy part is `[var, id1, =, lit2]` may deem a candidate location like `var num = 0` for seeding bug. The seeded bug may result in code such as `var var num = 1`, which is syntactically incorrect. In practice, we find that 97% of all seeded bugs are syntactically correct and filter out the remaining ones.

4 IMPLEMENTATION

We implement SemSeed as an end-to-end bug seeding tool with JavaScript as the target programming language. We use the API provided by GitHub to get a list of the most popular JavaScript repositories that we clone locally. After the initial filtering of commits based on the commit message etc., the correct and buggy JavaScript files are obtained using built-in commands in `git`. The static analysis on the JavaScript programs to extract nodes, the corresponding tokens, the kind of tokens etc., has been implemented using *esprima*. To obtain token embeddings, we pre-train FastText [8] on token sequences extracted from a corpus of 150K JavaScript [47] files.

5 EVALUATION

We evaluate SemSeed based on bug fixes extracted from the version histories of 100 popular JavaScript projects. The evaluation addresses the following research questions:

- RQ1: How effective is SemSeed in reproducing real-world bugs?
- RQ2: How does SemSeed compare to a semantics-unaware variant of the approach?
- RQ3: What is the impact of the configuration parameters of the approach?
- RQ4: How useful are the seeded bugs for training a learning-based bug detector?
- RQ5: How do the seeded bugs compare to bugs created with traditional mutation operators?
- RQ6: How efficient is SemSeed in seeding bugs?

Our implementation and all experimental results are available[1].

5.1 Experimental Setup

We gather bug-fixing commits from the version histories of the 100 JavaScript projects that have most stars on GitHub. For each repository, we extract all commits and filter them as explained in Section 3.1.1, resulting in 3,600 concrete bug fixes. We split the bugs into 2,880 *guiding bugs*, used to extract bug seeding patterns and as concrete bugs-to-imitate, and 720 *held-out bugs*. The split is date-based, using older commits as guiding bugs and newer commits as held-out bugs, so we can evaluate whether imitating bugs from the past creates bugs that have occurred later on. Extracting bug seeding patterns from the guiding bugs yields 2,201 bug seeding patterns. The frequency of the patterns follows a long tail distribution, which shows that real-world bugs are diverse, and that extracting bug seeding patterns from a large dataset is worthwhile.

The approach depends on three configuration parameters that control how many and which bugs get seeded: the matching threshold m, the set T of tokens to choose unbound tokens from, and the number k of bugs to seed per code location. As a default, we

[1] http://u.pc.cd/MrH

use $m = 0.2$, $k = 10$, and T as all tokens in the file where the bug gets seeded plus the 1,000 most frequent tokens across all files with guiding bugs. RQ3 further evaluates the impact of these parameters.

5.2 RQ1: Effectiveness in Reproducing Real-World Bugs

We evaluate SemSeed's ability to seed realistic bugs by comparing the seeded bugs with the held-out bugs. There are two preconditions for SemSeed to be able to reproduce a specific bug. First, the bug seeding pattern of the bug must occur across the guiding set and the held-out set. Due to the long-tail distribution of bug seeding patterns, many of the patterns occur only once, and we focus on the 151 concrete bugs that have a pattern in the intersection of guiding bugs and held-out bugs. Second, for bugs that involve tokens not present in the correct code, i.e., unbound tokens, the unbound token must be in the set T of tokens the approach chooses from when applying a bug seeding pattern. For our default configuration of T, 53 out of the 151 bugs that fulfill the first precondition also fulfill the second precondition. We use this set of 53 held-out bugs as the *target bugs*, and compute how many of them SemSeed reproduces, i.e., the seeded bug is exactly the same as the original bug.

Given the files in which the 53 target bugs should be seeded, the semantic matching identifies all 53 locations as a target location. 16 of the target bugs are rearrangements of existing tokens, i.e., similar to the inferred mutation operators of prior work [9]. Seeding these bugs is straightforward and SemSeed reproduces all of them. The remaining 37 involve unbound tokens, and SemSeed's algorithm for binding these tokens is successful for all but six of the bugs. Overall, the approach reproduces 47 out of the 53 target bugs.

Table 2 shows three examples of successfully reproduced real-world bugs. For each example, we show the correct and buggy variant of both the bug-to-imitate and the seeded bug. The first example is a bug without unbound tokens, but which requires rearranging existing tokens only. The second example is a bug with an unbound identifier token, where the following analogy queries help to select the identifier `stdout`: *What token is to `parent` what `official` is to `catalog`? What token is to `stderr` what `official` is to `complete`? What token is to on what `official` is to `getReleaseVersion`?* Finally, seeding the third bug requires binding two unbound tokens, which SemSeed again successfully finds by searching for tokens similar to the tokens in the buggy code, e.g., finding `timeout` as a token similar to `connectionTimeout`.

SemSeed reproduces 47 out of 53 bugs that are in scope for the approach.

5.3 RQ2: Comparison with Semantics-Unaware Bug Seeding

SemSeed relies on the semantic information embedded in identifiers and literals in two ways: (i) to select the locations for imitating a given bug, and (ii) to bind unbound tokens. To show the importance of these ideas, we compare our approach against a semantics-unaware variant of SemSeed, which (i) applies a bug pattern at every syntactically matching location, and (ii) binds unbound tokens by randomly picking from all tokens in the set T. This baseline approach reproduces only 16 out of the 53 target bugs from RQ1.

Table 2: Examples of reproduced real-world bugs.

Correct code	Buggy code
Bug to imitate: Commit b776e2b7 of jQuery	
`var opt = speed && typeof speed === "object"`	`var opt = typeof speed === "object"`
Seeded bug: Commit b94532c2 of Chart.js	
`if ( style && typeof style === 'object') {`	`if (typeof style === 'object') {`
Bug to imitate: Commit ad708ca5 of Meteor	
`catalog. complete . getReleaseVersion`	`catalog. official . getReleaseVersion`
Seeded bug: Commit bd74fb4c of Node.js	
`parent. stderr .on('data', function() { ... });`	`parent. stdout .on('data', function() { ... });`
Bug to imitate: Commit 1027871e of webpack	
`optimization: { chunkIds : "named" }`	`optimization: { namedChunks : true }`
Seeded bug: Commit 28f346e8 of freeCodeCamp	
`db: { connectionTimeout : 15000 }`	`db: { timeout : 10000 }`

Table 3: Five most frequent and five randomly selected bug seeding patterns. Unbound tokens are highlighted.

Correct	Buggy	Nb.
`id1 : lit1`	`id1 : lit2`	99
`lit1 : lit2`	`lit1 : lit3`	71
`id1.id2(lit1);`	`id1.id2( lit2 );`	40
`var id1 = lit1;`	`var id1 = lit2 ;`	33
`id1 : lit1`	`id2 : lit1`	18
`id1 = lit1 in id2`	`id1 = !!id2. id3`	1
`id1.id2(lit1 + id3).id4);`	`id1.id2(lit1 + id3);`	2
`id1.id2(id3[id4.id5]);`	`id1.id2(id4.id5)`	2
`var id1 = id2.id3(id4);`	`var id1 = id2.id3;`	1
`var id1 = id2.id1;`	`var id1=id2. id3 ;`	5

All of these 16 bugs do not have any unbound tokens. For bugs that need unbound token, we repeat the experiment for ten times with different seed values and randomly select a token from T. In none of the ten repetitions does the random selection pick the correct token required to seed a bug. The reason why randomly binding unbound tokens is ineffective is that picking the right token by chance from T is unlikely. In our default configuration, T contains more than 1,000 tokens, and even when T consists only of tokens

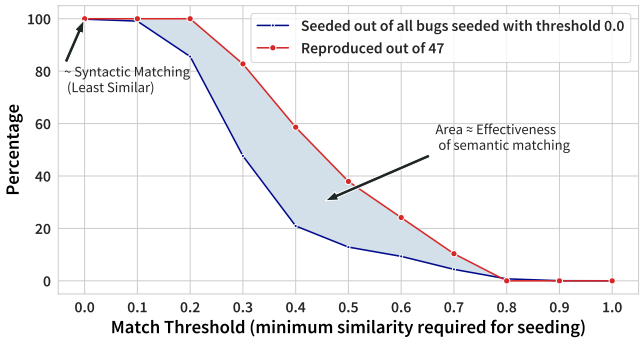


Figure 3: Influence of matching threshold m on seeded bugs.

that appear in the same function, there typically are several dozens of identifiers and literals to choose from.

To further illustrate the importance of handling unbound tokens, Table 3 lists some bug seeding patterns that SemSeed finds, along with their frequency in our dataset. All of the five most common bug seeding patterns (top-5 shown in table) and 62% of all bug seeding patterns contains at least one unbound token. Prior work on bug seeding based on past bug fixes does not handle unbound tokens [9, 58], and hence, could not benefit from these patterns.

A semantics-unaware variant of SemSeed reproduces only 16 out of 53 target bugs, and not handling unbound tokens misses 62% of all bug seeding patterns.

5.4 RQ3: Impact of Configuration Parameters

5.4.1 Matching Threshold m. The matching threshold m determines in Algorithm 1 whether to apply a bug pattern to a code location. A threshold of 0 means that the seeding location need not be similar to the bug-to-imitate at all, i.e., the decision to apply a bug seeding pattern is purely syntactic. In contrast, a threshold of 1 requires the tokens to perfectly match the original bug.

Figure 3 shows how the matching threshold influences the bugs that SemSeed creates. The two curves show two percentages: in blue, the percentage of seeded bugs out of all bugs that a purely syntactic approach, i.e., with $m = 0$, would seed; in red, the percentage of reproduced target bugs (RQ1) among the seeded bugs. As expected, both percentages decrease when the matching threshold increases. The gap between the curves shows that the semantic matching of target locations is effective. For example, with a matching threshold of 0.4, the approach seeds a bug at only 20% of all possible locations, but still reproduces 60% of all bugs that SemSeed can reproduce.

Compared to purely syntactic matching of bug patterns, the semantic matching increases the chance to seed realistic bugs.

5.4.2 Token Set T and Number k of Bugs to Seed. Another parameter is the set T of tokens to consider when binding unbound tokens (Section 3.3.3). We experiment with three variants of T:

(1) T_{fct}: Search for unbound tokens only in the function where the bug gets seeded.
(2) T_{file}: In addition to T_{fct}, search among all tokens in the file where the bug gets seeded.

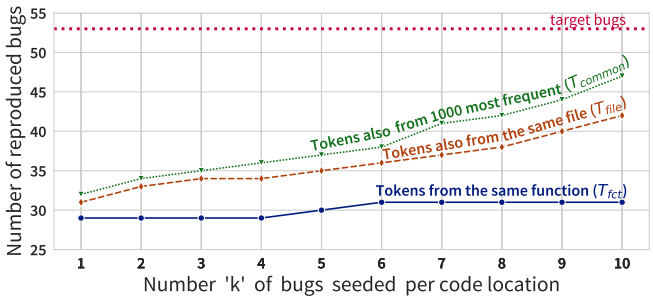


Figure 4: Reproduced real-world bugs depending on token set T and number k if bugs to seed.

(3) T_{common}: In addition to T_{file}, search among the 1,000 most frequent tokens across all files in the guiding set.

A larger search space increases the chance that the token required to reproduce a bug exists in T, but also makes it more difficult to choose the right token. A related parameter is how many bugs to seed for a given code location and bug seeding pattern. Our approach seeds one bug for each of the k most likely tokens found by Algorithm 2, and we evaluate values of k ranging from 1 to 10.

Figure 4 illustrates the effect that the token set T and the number k of bugs to seed have on the number of real-world bugs that SemSeed reproduces. We see that using a more restricted search space of tokens yields fewer reproduced bugs than a larger search space. Regarding the influence of k, considering more than the single most likely token significantly increases the number of reproduced bugs, in particular for larger T. Our default configuration of $T = T_{common}$ and $k = 10$ yields 47 reproduced bugs.

Depending on the token set T and the number k of bugs to seed, SemSeed reproduces between 29 and 47 of the target bugs.

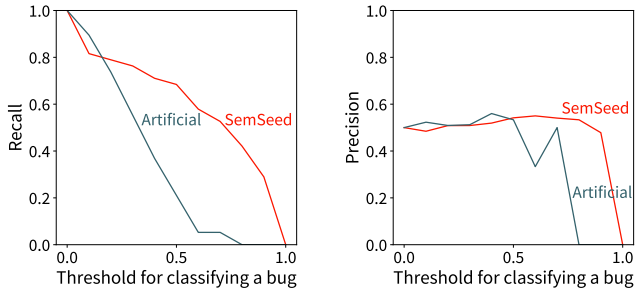
5.5 RQ4: Usefulness for Training a Learning-Based Bug Detector

To evaluate the usefulness of semantic bug seeding, we explore one of the applications of SemSeed: seeded bugs as training data for learning-based bug detection. We build on DeepBugs [46], which learns from examples of correct and incorrect code, and then predicts bugs in previously unseen code. DeepBugs supports several bugs patterns, of which we focus on two that are particularly challenging to seed bugs for:

- Wrong assignment bugs, where the right hand side of an assignment is incorrect, e.g., writing `i=o;` instead of `i=0;`.
- Wrong binary operands, where a developer uses an incorrect operand in a binary expression, e.g., accidentally writing `length * height` instead of `length * breadth`.

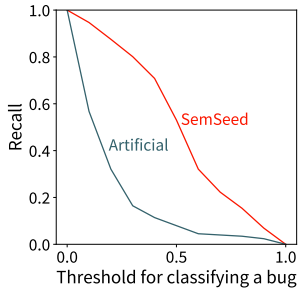
The other bug patterns [46], e.g., swapping function arguments, are simpler to seed and do not require to select unbound tokens.

We train DeepBugs using two configurations that differ in the way the incorrect code examples are generated. One configuration, called "artificial", uses DeepBugs's default generation of incorrect code examples, which randomly applies purely syntactic transformations and binds unbound tokens at random from T_{file}. The other configuration generates incorrect examples with SemSeed, which we configure to seed only bugs that match the two bug patterns

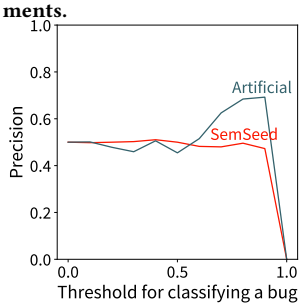


(a) Recall wrong assignments.

(b) Precision wrong assignments.



(c) Recall wrong binary operand.



(d) Precision wrong binary operand.

Figure 5: Precision and recall of DeepBugs with artificially seeded and SemSeed-seeded bugs.

targeted by DeepBugs. We apply both configurations to the same code corpus: a de-duplicated version [1] of a JavaScript corpus [47], which consist of 120K files. Generating for each correct example at most one incorrect example, the "artificial" configuration yields 1.1 million wrong assignments and 2.6 million wrong binary operands. Since SemSeed focuses on locations that have a semantic match with one of the guiding bugs, it creates fewer incorrect examples, namely 248K wrong assignments and 267K wrong binary operands.

Once trained, we measure the ability of DeepBugs to detect real-world bugs. As the bug patterns targeted by DeepBugs are relatively rare, we gather the bugs in three ways: (i) Those 8 of the held-out bugs that match the two bug patterns; (ii) Additional bugs gathered from 900 popular GitHub JavaScript projects using the methodology in Section 3.1.1; and (iii) Bugs from the JavaScript variant of an existing dataset of single-statement bugs [26]. This process yields 412 bugs (35 wrong assignments and 377 wrong binary operands).

We measure precision, i.e., how many of all reported warnings are among the known bugs, and recall, i.e., how many of all known bugs DeepBugs finds. For each warning, DeepBugs returns a probability for the location to be buggy. Figure 5 shows the precision and recall of DeepBugs depending on the probability threshold used to decide which warnings to report. Overall, using SemSeed-generated bugs instead of artificial bugs significantly increases the effectiveness of DeepBugs, with clearly improved recall and roughly the same precision. For example, using a threshold of 0.5, SemSeed increases the detected bugs from 7% to 53%.

To understand why SemSeed improves the bug detection ability of learned bug detectors, consider two bugs seeded into the following code:

```
for (var i = 0; i < coordinates.length ; i += 2)
```

SemSeed seeds a bug by turning `length` into another identifier that also refers to a dimension and that semantically fits the surrounding tokens, i.e., a bug that a developer might actually introduce:

```
for (var i = 0; i < coordinates.offsetHeight ; i += 2)
```

In contrast, DeepBugs uses an arbitrary other identify from the same file, resulting in a rather unrealistic bug:

```
for (var i = 0; i < coordinates.enableClickBuster ; i += 2)
```

As illustrated by this example, a model trained on the artificial bugs tends to identify obvious yet unrealistic mistakes. Instead, training DeepBugs with SemSeed's bugs teaches the model to identify subtle yet more realistic mistakes. More broadly, the results also illustrate a quality-versus-quantity tradeoff in bug seeding: The SemSeed-generated bugs yield more effective bug detectors, despite being an order of magnitude fewer than the artificially created bugs.

Using semantically seeded bugs as training data for a learning-based bug detector allows for finding significantly more bugs.

5.6 RQ5: Comparison with Traditional Mutation Operators

Existing code mutation approaches, such as Mutandis [40] for JavaScript and Major [23] for Java, use pre-defined mutation operators. We compare the mutation operators in Mutandis with the bugs created by SemSeed. Based on the 2,880 guiding bugs, we seed 677,217 bugs into a random sample of 1,000 JavaScript files and then compare the seeded bugs to the 23 mutation operators in Mutandis.[2]

98.2% of the SemSeed-generated bugs go beyond the 23 pre-defined mutation operators. The 1.8% of the bugs that are shared with Mutandis correspond to 165 out of the 2,880 guiding bugs. For example, one of the Mutandis patterns is about changing a literal in a condition, a change SemSeed also performs. Another example is about removing the `var` keyword from a variable declaration, a pattern that SemSeed also learns and applies. Inversely, Mutandis also creates some bugs that SemSeed cannot seed. Out of the 23 Mutandis operators, SemSeed has a corresponding bug seeding pattern for 16. For 13 out of these 16, SemSeed seeds at least one bug, while for the remaining three no suitable bug seeding location is found. Among the remaining 23 − 16 = 7 Mutandis operators, two are out of scope for SemSeed because the code transformation affects more than one line, e.g., swapping two nested loops. For the other five operators, SemSeed could in principle seed bugs, but there is no corresponding guiding bug. These are mostly about changes to JavaScript APIs, e.g., removing the integer base argument `10` from calls like `parseInt('09/11/08', 10)`.

SemSeed complements traditional mutations by seeding many bugs beyond a fixed set of pre-defined mutation operators.

5.7 RQ6: Efficiency

We measure the efficiency of SemSeed by keeping track of the time it needs to seed the 677,217 bugs into the 1,000 files from RQ5. Because some files allow for thousands of seeded bugs, we set a time

[2] Mutandis can also use runtime information to decide which bugs to seed, which we ignore here because our focus is on static bug seeding.

limit of 30 minutes per file. Out of the 1,000 files, SemSeed could seed bugs into 902 files where it found at least one matching bug seeding pattern. In total, seeding 677,217 bugs takes 140 minutes. Analyzing what part of the approach takes the most time, we find that the analogy queries are the biggest bottleneck.

SemSeed takes, on average, 0.01 seconds to seed a bug and hence can generate a large number of bugs in very little time.

6 LIMITATIONS AND THREATS TO VALIDITY

SemSeed focuses on single-line bugs, for two reasons: (i) we can gather a large set of these bugs automatically, which facilitates the evaluation, and (ii) these bugs are relevant and important in practice [16, 26, 48]. For example, Karampatsis and Sutton [26] show that there is an instance of one out of 16 common patterns of single-line bugs every 1,600 to 2,500 lines of code. To generalize SemSeed to more complex bugs, one would consider token sequence that span multiple lines. One challenge we anticipate is that the probability that a complex bug-to-imitate syntactically matches in another program is smaller than for single-line bugs. Addressing this challenge, e.g., by approximately matching bug seeding patterns to code locations, remains for future work.

Among the many applications of bug seeding, we select learning-based bug detection to evaluate SemSeed's usefulness. Based on our comparison with traditional mutation operators, we are optimistic that the approach could also be useful, e.g., for mutation testing, and envision future work on this and other applications.

We implement the approach for JavaScript and cannot draw conclusions about how well it would work for other languages. The fundamental challenges that SemSeed addresses, i.e., where to seed bugs, how to adapt a given example bug to a target location, and how to handle unbound tokens, are language-independent.

7 RELATED WORK

Bug Seeding. One approach to bug seeding is to apply mutations, e.g., based on a predefined set of transformation patterns [20, 23]. Similar to our work, [9] propose to infer such patterns from code changes. In contrast to these approaches, SemSeed decides where to apply a bug seeding pattern and how to adapt it to the local code context based on semantic similarities of code elements. Tufano et al. [56] describe a neural machine translation-based approach to learn and apply mutations. Their approach requires hundreds of thousands of bug-fixing commits to be trained properly. In contrast, SemSeed learns from few examples – in the extreme case, one can use a single example bug to seed similar bugs at various target locations. An important difference between SemSeed and both [9] and [58] is that our approach handles unbound tokens, seeding bugs even if this requires an application-specific identifier or literal.

Tailored mutation operators [2, 13, 23, 36, 40], e.g., insert code fragments that occur elsewhere in a project. In contrast to such approaches, the mutations applied by us are based on previously seen bug fixing patterns and not project-specific as in [2] or hard coded as in [23, 40]. IBIR also learns from past bugs how to seed new bugs [28]. It uses natural language in a bug report to decide where to seed a bug, whereas SemSeed focuses on the tokens (including natural language identifiers) in the code. IBIR neither adapts bugs to a target location and nor addresses the unbound token problem, which we show to be crucial for the majority of bug patterns.

Motivated by the abundance of fuzz testing tools [6, 17, 45, 53], automatically seeded bugs have been proposed for evaluating fuzz testing [15, 50]. These seeded bugs aim at being non-trivial to trigger in an execution, but are easy to detect on the source code level, e.g., because the seeded bug relies on magic numbers.

Mining Code Change Patterns. Osman et al. [44] describe an empirical study of frequent bug fixing code changes. Negara et al. [41] identify repetitive code changes from fine-grained sequences of code changes recorded in an IDE. In contrast, our approach mines only concrete changes that correspond to a bug fix rather than any change made by a developer. Nguyen et al. [42] mine semantic change patterns by converting the correct and buggy files to program dependence graphs. Instead of a graph, we leverage embeddings of tokens as the semantic representation and extract changes as a sequence of tokens. Kim et al. [29] manually inspect human-written patches to infer common fix patterns. Neural machine translation can learn to apply bug fixes [57]. SemSeed addresses the inverse problem of seeding bugs, instead of fixing them.

Bug Benchmarks. Several bug benchmarks have been proposed, including SIR [14], Defects4J [24], BugSwarm [55] Bugbench [37], BegBunch [11], iBugs [12], ManyBugs [31], Codeflaws [54], DbgBench [7]. Our work complements such manually curated bug datasets by automatically seeding bugs into a target program.

Finding Matching Code. Code clone detection [22, 25, 35, 49, 52] relates to the semantic matching part of SemSeed. These approaches find matching code pieces via string-based, parse tree-based, or token-based comparisons. Our semantic matching relates to the token-based techniques, but differs by using token embeddings to find a match.

Token Embeddings. Recent work shows that token embeddings enable learning-based program analysis, e.g., to detect bugs [46], to predict types [38], to de-obfuscate code [3], or to map APIs across programming languages [43]. Our work is the first to use pre-trained token embeddings for bug seeding.

8 CONCLUSION

This paper presents SemSeed, an approach for seeding bugs in a semantics-aware way. Given a possibly small set of example bugs, the approach infers bug seeding patterns and then imitates the given bugs at various code locations in a target program. The key novelty is to go beyond purely syntactic bug seeding by (i) checking if a code location semantically matches the bug-to-imitate, (ii) adapting the bug seeding pattern to the local code context, and (iii) binding unbound tokens based on semantic analogy queries. To reason about the semantics of code elements, SemSeed builds on learned token embeddings, which have not been used for bug seeding before. Our evaluation with thousands of real-world bugs shows that the approach effectively seeds realistic bugs, while being efficient enough for creating hundreds of thousands of bugs within an hour. The created bugs complement traditional mutation operators and are useful as training data for learning-based bug detectors, allowing them to find many otherwise missed bugs.

REFERENCES

[1] Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. *arXiv preprint arXiv:1812.06469* (2018).

[2] Miltiadis Allamanis, Earl T Barr, René Just, and Charles Sutton. 2016. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516* (2016).

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *PLDI*.

[4] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SyK00v5xx

[5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. In *OOPSLA*.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Software Eng.* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

[7] Marcel Böhme, Ezekiel Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 117–128. https://publications.cispa.saarland/1468/

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146. https://transacl.org/ojs/index.php/tacl/article/view/999

[9] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 511–522.

[10] DARPA CGC. 2018. Darpa Cyber Grand Challenge (CGC) Binaries. https://github.com/CyberGrandChallenge/.

[11] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. 2009. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 16–20.

[12] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 433–436.

[13] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. 2017. Assessment of Class Mutation Operators for C + + with the MuCPP Mutation System. *Inf. Softw. Technol.* 81, C (Jan. 2017), 169–184. https://doi.org/10.1016/j.infsof.2016.07.002

[14] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.* 10, 4 (Oct. 2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[15] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 110–121.

[16] Aryaz Eghbali and Michael Pradel. 2020. No Strings Attached: An Empirical Study of String-related Software Bugs. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 956–967. https://ieeexplore.ieee.org/document/9286132

[17] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*.

[18] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close are they to Real Faults?. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 189–200. https://doi.org/10.1109/ISSRE.2014.40

[19] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *ASE*.

[20] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[21] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.

[22] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[23] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 433–436.

[24] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 437–440.

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[26] Rafael-Michael Karampatsis and Charles A. Sutton. 2019. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. *CoRR* abs/1905.13334 (2019). arXiv:1905.13334 http://arxiv.org/abs/1905.13334

[27] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *ICSE*.

[28] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2020. IBIR: Bug Report driven Fault Injection. *CoRR* abs/2012.06506 (2020). arXiv:2012.06506 https://arxiv.org/abs/2012.06506

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches.. In *International Conference on Software Engineering (ICSE)*. 802–811.

[30] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[31] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[32] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *ICSE*.

[33] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. In *OOPSLA*.

[34] Zhen Li, Shouhuai Xu Deqing Zou and, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *NDSS*.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192.

[36] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/3106237.3106275

[37] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[38] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *ICSE*.

[39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.

[40] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. 2013. Efficient JavaScript Mutation Testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 74–83.

[41] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

[42] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 819–830.

[43] 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 2017, Buenos Aires, Argentina, May 20-28, 2017*. 438–449.

[44] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. 2014. Mining frequent bug-fix code changes. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 343–347.

[45] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 697–710.

[46] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[47] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 761–774.

[48] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[49] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181.

[50] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: challenging bug-finding tools with deep faults. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 224–234.

[51] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 10–13.

[52] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[54] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[55] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 339–349. https://doi.org/10.1109/ICSE.2019.00048

[56] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[57] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[58] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 301–312. https://doi.org/10.1109/ICSME.2019.00046

[59] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.

[60] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE*. 297–308.