Learning to Find Naming Issues with Big Code and Small Supervision

Jingxuan He
ETH Zurich, Switzerland
jingxuan.he@inf.ethz.ch

Cheng-Chun Lee
EPFL, Switzerland
cheng-chun.lee@alumni.epfl.ch

Veselin Raychev
Snyk, Switzerland
veselin.raychev@snyk.io

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

We introduce a new approach for finding and fixing naming issues in source code. The method is based on a careful combination of unsupervised and supervised procedures: (i) unsupervised mining of patterns from Big Code that express common naming idioms. Program fragments violating such idioms indicates likely naming issues, and (ii) supervised learning of a classifier on a small labeled dataset which filters potential false positives from the violations.

We implemented our method in a system called Namer and evaluated it on a large number of Python and Java programs. We demonstrate that Namer is effective in finding naming mistakes in real world repositories with high precision (~70%). Perhaps surprisingly, we also show that existing deep learning methods are not practically effective and achieve low precision in finding naming issues (up to ~16%).

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; • **Theory of computation** → **Program analysis**.

Keywords: Name-based program analysis, Static analysis, Bug detection, Anomaly detection, Machine learning

ACM Reference Format:
Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. 2021. Learning to Find Naming Issues with Big Code and Small Supervision. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454045

1 Introduction

Names in source code are important as they often convey program semantics and capture developers' intention, helping with tasks such as code understanding [8], completion [7], and debugging [38]. Names also play a key role in code quality and software maintenance [18], while some naming problems are misuses and even bugs [9, 28, 41]. The importance of names is reassured by many works in academia [30, 34] and industry [41]. Thus, a system able to automatically detect and fix naming issues is highly desirable.

Key challenge. While important, detecting and fixing naming issues in programs is a difficult task. Conventional methods such as abstract interpretation [21], symbolic execution [19], or testing [1], do not capture names and thus fail to detect naming issues. Indeed, since identifier names are closer to natural language, differentiating between good and bad names inherently requires statistical reasoning [13, 40]. Statistical naming issue detectors are ideally built on a large scale supervised dataset where *real* programs are labeled with having naming issues or not. However, to the best of our knowledge, such a large dataset does not exist and is hard to obtain with either manual or automatic approaches.

To address this challenge, existing works rely on either generating synthetic mistakes or on methods where labels are not required. Deep neural networks are currently trained and tested on large labeled datasets constructed by injecting *synthetic* defects in programs [9, 28, 45, 47]. However, despite achieving high test accuracy, we found these methods to have low precision in finding real world issues (discussed in § 5.6). While initially surprising, further thought reveals that the issue is caused by the fundamental problem of distribution mismatch: the distribution induced by generating synthetic defects does not match the distribution of real world mistakes. Alternatively, anomaly detection based approaches do not require labeled data but often require substantial manual effort for creating effective rules and tuning appropriate thresholds [34, 38, 41]. As a result, these systems are often limited to certain types of bugs (e.g., wrong usages of argument names) and language (e.g., Java).

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454045

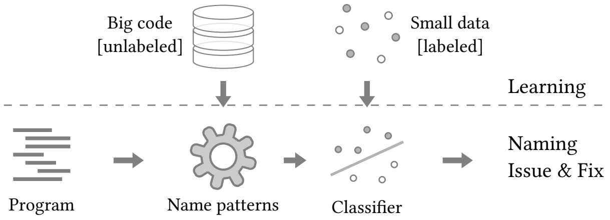


Figure 1. Our recipe for finding and fixing naming issues.

This work. In this work, we propose a new recipe for learning to detect naming issues. The key idea is to split the learning procedure into two steps, see top of Figure 1: (i) mine *name patterns*, interpretable naming rules that capture a diverse set of naming idioms, from a large dataset of programs (i.e., Big Code), and (ii) train a binary classifier that uses high-level expressive features and predicts whether a violation of a name pattern is an issue. Importantly, step (i) does not require programs to be labeled with having issues or not, while step (ii) only requires a small labeled dataset, which is feasible to produce manually. This combination obviates the need to provide synthetic data and ensures that one trains and tests on the same real world data. To use the method at inference time, we follow the steps shown at the bottom of Figure 1. Given a program fragment (with its representation): (i) we query the mined name patterns to check if the fragment violates any of them, and (ii) if it does, we query our learned binary classifier. A naming issue is reported only if the classifier predicts the violation to be true. The suggested fix is to change the relevant parts of the fragment so the originally violated pattern is satisfied.

We implemented our recipe in a system called Namer and evaluated it on both statically typed (Java) and dynamically typed (Python) languages. We show that Namer is significantly more effective at finding naming issues than state-of-the-art deep networks trained on synthetic defects.

Main contributions. Our key contributions are:

- A novel method for detecting and fixing naming issues in code consisting of interpretable name patterns mined from Big Code which capture a diverse set of naming idioms (§ 3.2, § 3.3), and a machine learning classifier with expressive, high-level features to filter false positives (§ 4.2).
- An end-to-end implementation of our method in a tool called Namer. Namer supports Python and Java, leverages powerful static analyses for extracting semantic information, and is readily applicable to other languages (§ 5.1).
- An extensive evaluation of Namer on a large, real world dataset containing millions of files from GitHub, demonstrating that Namer achieves high precision (~70%) and is practically effective (§ 5). Our evaluation also shows that despite being accurate on programs with synthetic defects, current deep learning techniques [9, 28] achieve very low precision (i.e., up to ~16%) on detecting real issues (§ 5.6).

2 Overview

In this section, we provide an overview of Namer on an illustrative example. We run the inference pipeline in Figure 1 to show how Namer finds and fixes a naming issue.

Example program. In Figure 2 (a), we present a code snippet taken from an open-source Python project hosted on GitHub[1]. This snippet defines a class `TestPicture` inheriting from `TestCase`, which is from the `unittest` library and is used for representing test units. The `TestPicture` class contains a function `test_angle_picture` that calls `assertTrue`, a function from the parent class `TestCase`, with two arguments `picture.rotate_angle` and `90`. We underline the described call statement in the figure.

The use of `assertTrue` in Figure 2 (a) is incorrect. Based on the documentation [6], the second argument of `assertTrue` is an optional error message to display instead of a value to compare to. Therefore, we can infer that this code calls the wrong API function, which will cause unexpected behavior at runtime. Based on the code context, the correct function to call is `assertEqual`, which checks whether the values of the first two arguments are equal. Namer finds this bug and suggests to replace `assertTrue` with `assertEqual`. Next, we run the pipeline in Figure 1 on the buggy program statement and describe in detail how Namer finds and fixes this bug.

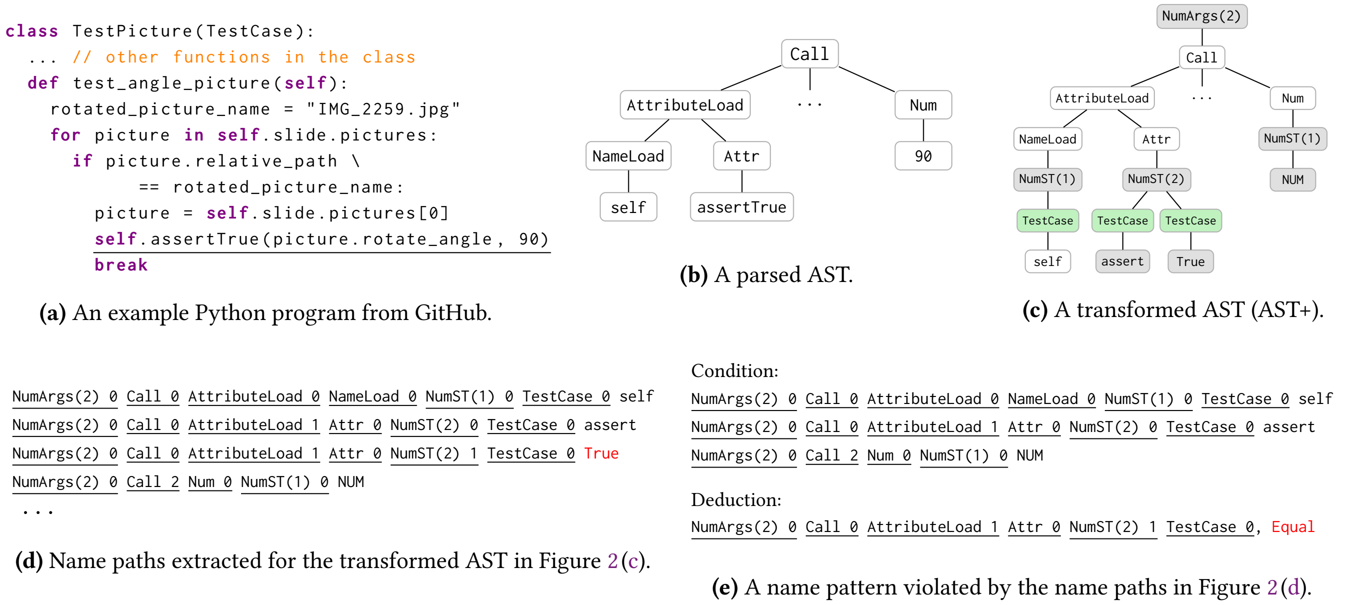
Program statement to AST+. Namer first parses the input program and obtains an abstract syntax tree (AST) for each program statement. Figure 2 (b) shows the AST for the buggy statement in our example with the nodes for `picture.rotate_angle` omitted for simplicity. Then the following transformations are applied on the parsed tree to obtain a transformed tree (AST+) as shown in Figure 2 (c):

1. Replace integer values (i.e., `90`) with a special token `NUM`.
2. Add a special node `NumArgs(2)` as the parent of the function call node `Call`, showing that the function call has two arguments (i.e., `picture.rotate_angle` and `NUM`).
3. Split each terminal node into subtokens based on standard naming conventions and insert a node `NumST(`k`)` where k is the number of subtokens in the original node. In the example, `assertTrue` is split into `assert` and `True`, and `NumST(2)` is inserted as a parent of the subtoken nodes. The nodes with value `NumST(1)` are added for node `self` and node `NUM` because they are unsplittable and contain only one subtoken.

These transformations allow Namer to detect naming issues on the *subtoken* level and make decisions based on additional information such as the number of function arguments.

Then, Namer applies an additional *semantic* transformation. Namer runs interprocedural points-to and data flow analyses which track the origin of each object and decorates

[1] https://github.com/paulhildebrandt/python-keynote/blob/master/tests/test_keynote_api.py



```
class TestPicture(TestCase):
  ... // other functions in the class
  def test_angle_picture(self):
    rotated_picture_name = "IMG_2259.jpg"
    for picture in self.slide.pictures:
      if picture.relative_path \
          == rotated_picture_name:
        picture = self.slide.pictures[0]
        self.assertTrue(picture.rotate_angle, 90)
        break
```

(a) An example Python program from GitHub.

(b) A parsed AST.

(c) A transformed AST (AST+).

```
NumArgs(2) 0 Call 0 AttributeLoad 0 NameLoad 0 NumST(1) 0 TestCase 0 self
NumArgs(2) 0 Call 0 AttributeLoad 1 Attr 0 NumST(2) 0 TestCase 0 assert
NumArgs(2) 0 Call 0 AttributeLoad 1 Attr 0 NumST(2) 1 TestCase 0 True
NumArgs(2) 0 Call 2 Num 0 NumST(1) 0 NUM
 ...
```

(d) Name paths extracted for the transformed AST in Figure 2 (c).

Condition:

```
NumArgs(2) 0 Call 0 AttributeLoad 0 NameLoad 0 NumST(1) 0 TestCase 0 self
NumArgs(2) 0 Call 0 AttributeLoad 1 Attr 0 NumST(2) 0 TestCase 0 assert
NumArgs(2) 0 Call 2 Num 0 NumST(1) 0 NUM
```

Deduction:

```
NumArgs(2) 0 Call 0 AttributeLoad 1 Attr 0 NumST(2) 1 TestCase 0, Equal
```

(e) A name pattern violated by the name paths in Figure 2 (d).

Figure 2. Overview of Namer on an example program.

the AST with semantic information obtained from the analyses results. For our example program, the analyses identify that the origin of the object `self` is `TestCase`. Namer then incorporates the analysis results in the AST. In Figure 2 (c), Namer inserts nodes with value `TestCase` (filled with green) as the parents of node `self`, `assert` and `True`. The analyses are described in § 4.1. We show in § 5 that running the analyses is a key factor for Namer to achieve high precision.

AST+ to name paths. Namer then traverses the transformed AST in a top-down fashion to extract *name paths*, our program abstraction for identifier name usages. A name path represents a path from the tree root to a leaf subtoken. In Figure 2 (d), we show a list of name paths extracted from the transformed AST in Figure 2 (c). Each name path contains two parts. The first part is a list where each element consists of a node and the index of the next node in the current node's children list. The second part of a name path is a leaf subtoken. For instance, the first line represents the name path from the tree root to the leaf `self`. Along this path, the first child is always selected so the indices are all zero. We formally define name paths in § 3.1.

Compared to existing syntactically constructed path abstractions [11–13, 33], our name paths contain richer semantic information from the transformed ASTs and have finer granularity as they can be used for modeling subtokens.

Pattern matching. The extracted name paths are then checked against a set of interpretable naming rules called *name patterns*. The name patterns are automatically mined from a large dataset of open source repositories and capture a wide range of common naming idioms. If the name paths violate any of the name patterns, then the statement deviates from common naming behaviors and a potential naming issue is detected by Namer. Moreover, the violated name pattern suggests how to fix the issue: modify the statement so that the violated pattern becomes satisfied. We formally define name patterns in § 3.2 and describe the mining algorithm in § 3.3. In Figure 2 (e), we show an example name pattern which consists of two sets of name paths, the *condition* and the *deduction*. At a high level, the name pattern indicates if a program statement calls a function that satisfies the following three requirements, as specified by the three name paths in the condition:

1. the receiver is `self` and is an instance of `TestCase`, and
2. the first subtoken of the function name is `assert`, and
3. the second function argument is a numerical value,

then the second subtoken of the function name should be `Equal`, as specified by the name path in the deduction. Our example statement `self.assertTrue(picture.rotate_angle, 90)` violates the name pattern in Figure 2 (e) because its name paths satisfy the condition but contradict with the deduction. The violation exhibits a potential naming issue and the fix suggested by Namer is to replace `True` with `Equal`.

Classifying violated patterns. So far, the mined name patterns can already identify anomalous naming behaviors and report them as naming issues. Balancing the tradeoff between recall and precision is known to be difficult for anomaly detectors [38, 41], especially when our mining process is performed on millions of source files. As a result, with different choices of hyperparameters in the matching and mining processes, the mined patterns will either find very few issues because the violations are rarely triggered, or will inevitably have lower precision because more violations are triggered on less anomalous code.

In Namer, we allow violations to be triggered at lower confidence so that most issues are not missed. Then, to prune false positives from the triggered violations, we build a binary *defect classifier* that uses various statistical measures from the violations as features. The classifier outputs true if the report should be reported to the user as a naming issue, or false otherwise. We describe the classifier in § 4.2. Our evaluation in § 5 shows that the classifier is a crucial element for Namer to achieve a high precision.

Issues handled by Namer. Namer targets a wide range of program elements (e.g., variables, functions, API calls, types, etc.) and usually reveals issues (e.g., indescriptive names, wrong API usages, wrong types, etc.) where the names of the elements do not correctly capture the underlying code semantics. On the AST level, Namer targets all terminal nodes with code names. Examples of naming issues detected by Namer are shown in Tables 3 and 6. In Figure 2, we show a *semantic defect* that causes an unexpected program behavior. The vast majority of Namer's reports are *code quality issues* that may impair code readability and maintainability.

3 Program Abstraction for Names

In this section, we describe our program abstractions for code names. We start by defining how we augment abstract syntax trees (ASTs) with results of program analyses and then how we extract name paths from these ASTs. Next, we define name patterns for representing common usages of identifier names. Finally, we present the algorithms for mining name patterns from a large dataset of code.

3.1 Abstract Syntax Tree and Name Path

We start by formally defining ASTs for program statements. Intuitively, this is part of the abstract syntax tree of the whole program, projected on a specific statement only.

Definition 3.1 (Abstract Syntax Tree)**.** An Abstract Syntax Tree (AST) for a program statement is a tuple $\langle N, T, r, \delta, V, \phi \rangle$, where N is a set of non-terminal nodes, T is a set of terminal nodes, $r \in N$ is the root node, $\delta : N \longrightarrow (N \cup T)^+$ is a function mapping a non-terminal node to the list of child nodes, V is a set of node values, and $\phi : (N \cup T) \longrightarrow V$ is a function mapping a node (either terminal or non-terminal) to its associated value. A parsed AST is shown in Figure 2(b).

Given a parsed AST, Namer performs the following steps to obtain a transformed AST:

1. Transform all numerical values into a special token `NUM`, all string values into `STR`, and all boolean values into `BOOL` to improve generalization.
2. For each function definition or function call node n, insert a node n' with value `NumArgs(`k`)` between n and n's parent, where k is the number of arguments in the definition or the call.
3. For each terminal node n whose value is an identifier name, split the name into subtokens based on standard naming conventions such as `camelCase` and `snake_case`. Then, replace n by a non-terminal node n' with value `NumST(`k`)`, where k is the number of produced subtokens. The subtokens are appended as n''s children.
4. For each terminal node representing an object name, insert a node whose value is the origin of the object as the parent. For each terminal node representing a function call, insert a node whose value is the origin of the receiver object (if any) as the parent. The information about object origin is obtained via interprocedural points-to and data flow analyses described later in § 4.1.

Note that due to variadic or keyworded arguments, calls to the same function may have a different number of arguments. Such cases are captured by the AST and are not treated specially. The pattern mining process in § 3.3 will automatically learn the behaviors of such functions. We show an example of transformed ASTs in Figure 2(c). Transformed ASTs contain more fine-grained (e.g., subtokens), semantic information (e.g., results from static analyses) than parsed ASTs, which strengthens the precision of our name path and name pattern abstractions. Given a transformed AST, the next step is to extract name paths defined in the following.

Definition 3.2 (Name path)**.** A name path extracted from a (transformed) AST is a tuple $\langle S, n \rangle$. We call S the prefix and n the end node. S is a list of non-terminal nodes along the path from the AST root to a terminal node. Formally, $S = [(n_j, i_j)]_{j=1}^{k}$ where $n_j \in N$ is a non-terminal node in the AST and $i_j \in \mathbb{N}$ is an index, such that $n_1 = r$ (n_1 is the root node) and $\delta(n_j)[i_j] = n_{j+1}$ (n_{j+1} is the i_jth child of n_j). Node n is either the concrete terminal node where the AST path ends ($\delta(n_k)[i_k] = n$, i.e., n is the i_kth child of n_k) or a special symbolic node ϵ introduced for adding more degrees of freedom to name patterns defined later in § 3.2. We call the name path a concrete (resp., symbolic) name path if n is concrete (resp., symbolic).

Example 3.3. In the following, we show three name paths np_1, np_2 and np_3. np_1 and np_2 are concrete name paths, and np_3 is a symbolic name path.

np_1: `NumArgs(2) 0` `Call 0` `AttributeLoad 1` `Attr 0` `NumST(2) 1` `TestCase 0` `True`

np_2: `NumArgs(2) 0` `Call 0` `AttributeLoad 1` `Attr 0` `NumST(2) 1` `TestCase 0` `Equal`

np_3: `NumArgs(2) 0` `Call 0` `AttributeLoad 1` `Attr 0` `NumST(2) 1` `TestCase 0` ϵ

Definition 3.4 (Relational operators of name paths)**.** Given two input name paths np_1 and np_2, we define the following two relational operators:

- $np_1 \sim np_2$: is true if $np_1.S = np_2.S$. That is, if $np_1.S$ and $np_2.S$ have the same length, and all corresponding elements in $np_1.S$ and $np_2.S$ are equal; is false otherwise.

- $np_1 = np_2$: is true if $np_1 \sim np_2 \wedge (np_1.n = \epsilon \vee np_2.n = \epsilon \vee np_1.n = np_2.n)$; is false otherwise.

Intuitively, $np_1 \sim np_2$ requires that the prefixes of np_1 and np_2 are the same. $np_1 = np_2$ additionally requires that the end nodes of np_1 and np_2 are equal, or either $np_1.n$ or $np_2.n$ is ϵ (that is, any value is equal to ϵ).

Example 3.5. In Example 3.3, $np_1 \sim np_2$ holds, $np_1 = np_2$ does not hold, and both $np_1 \sim np_3$ and $np_1 = np_3$ hold.

Each name path captures one name in the AST and a program statement can be represented by a set of name paths extracted from its AST. An example of name paths extracted from a code statement is shown in Figure 2(d). The name path representation makes it convenient to find common properties of program statements for later construction of name patterns, common coding idioms that generalize across statements. We note that the set of name paths A extracted from a program statement must satisfy two properties: $\forall a \in A.\, a.n \neq \epsilon$, i.e., all name paths in A are concrete, and $\forall a_1, a_2 \in A.\, a_1.S \neq a_2.S$, i.e., all prefixes of the name paths in A are different. These properties are useful when we check the name paths of a statement against name patterns.

3.2 Name Pattern

We now define name patterns which consist of name paths and are used to represent the most common usage of identifier names in a statement.

Definition 3.6 (Name pattern)**.** A name pattern p consists of two sets of name paths, condition C and deduction D. Intuitively, a name pattern requires that if a statement includes certain name paths (defined by C), then name paths that conform to D should also be present in this statement. We provide three relationships between a name pattern p and a given statement s represented by a set of name paths A: match, satisfaction, and violation. In the following, we define the match relationship and briefly introduce the other two at a high level. We formally define the satisfaction and violation relationships individually for two specific pattern types introduced later in this section.

- *Match*: s matches p if $\forall c \in C.\, \exists a \in A.\, c = a$ and $\forall d \in D.\, \exists a \in A.\, d \sim a$, or intuitively, if all name paths in C also exist in A and all prefixes of the name paths in D also exist in A. For example, the name paths in Figure 2(d) match the name pattern in Figure 2(e). Note that there is no requirement on the end nodes of name paths in the deduction D.
- *Satisfaction*: s satisfying p means that s conforms to the naming idiom represented by p.
- *Violation*: s violating p means that s violates the naming idiom represented by p. Therefore, a potential naming issue is revealed and the suggested fix is to modify the statement so that its name paths satisfy p.

The definition of name patterns is generic and allows one to define different types of patterns to capture different kinds of naming idioms. Next, we introduce two types of name patterns in Namer while leaving the addition of more patterns as future work items. The first one is consistency name patterns whose purpose is to ensure that code fragments with the same underlying semantics should be named consistently, which is an important aspect of code quality.

Definition 3.7 (Consistency name pattern)**.** A consistency name pattern requires that $D = \{d_1, d_2\}$ (i.e., the deduction D consists of two name paths d_1 and d_2), and both d_1 and d_2 are symbolic. The satisfaction and violation relationships of consistency name patterns are defined as follows:

- *Satisfaction*: s satisfies p if s matches p and $\forall a_1, a_2 \in A.\, (a_1.S = d_1.S \wedge a_2.S = d_2.S) \implies (a_1.n = a_2.n)$. Intuitively, it requires that any two name subtokens in the statement prefixed by paths d_1 and d_2 must be equal.
- *Violation*: s violates p if s matches p but does not satisfy p.

Example 3.8. In the following, we show a simple consistency name pattern which enforces that in a Python statement of the form `self.<name`$_1$`> = <name`$_2$`>`, the two names `<name`$_1$`>` and `<name`$_2$`>` must be equal.

$$\begin{array}{ll} \text{Condition:} & \underline{\texttt{Assign 0}}\ \underline{\texttt{AttributeStore 0}}\ \underline{\texttt{NameLoad 0}}\ \underline{\texttt{NumST(1) 0}}\ \underline{\texttt{Object 0}}\ \texttt{self} \\ \text{Deduction:} & \underline{\texttt{Assign 0}}\ \underline{\texttt{AttributeStore 1}}\ \underline{\texttt{Attr 0}}\ \underline{\texttt{NumST(1) 0}}\ \epsilon \\ & \underline{\texttt{Assign 1}}\ \underline{\texttt{NameLoad 0}}\ \underline{\texttt{NumST(1) 0}}\ \underline{\texttt{Str 0}}\ \epsilon \end{array}$$

The second type of name pattern used in Namer is the confusing word name pattern. Its goal is to detect words that developers tend to mistakenly use and to suggest a correction. To obtain such patterns, we extract the pairs of mistaken and correct words, referred to as confusing word pairs.

Confusing word pairs. A confusing word pair consists of two words $\langle w_1, w_2 \rangle$ where in some prior version of the code, w_1 was used instead of w_2. We call w_1 the mistaken word and w_2 the correct word. We discover confusing word pairs from all of the (deduplicated) commits in our dataset. First, we apply a diff matching algorithm [37] over the AST of the program before and after the commit. For each pair of matched nodes, the names are split into subtokens and if there is one difference in the subtokens, the pair of subtokens is added as a confusing word pair. We extracted 950K confusing word pairs for Java and 150K for Python. Examples of mined confusing word pairs are ⟨name, key⟩, ⟨value, key⟩, ⟨x, y⟩, ⟨min, max⟩, ⟨True, Equal⟩.

Definition 3.9 (Confusing word name pattern)**.** A name pattern is a confusing word name pattern if $D = \{d\}$ (i.e., the deduction D consists of a single name path d) and there exists a pair $\langle w_1, w_2 \rangle$ in the set of mined confusing word pairs such that $d.n = w_2$. The satisfaction and violation relationships of confusing word patterns are defined as follows:

Algorithm 1: Mining name patterns.

```
1  Procedure minePatterns(progs, t)
       Input   : progs, a set of programs.
                 t, the type of the name patterns to mine.
       Output  : patterns, a set of mined name patterns.
2      asts = getStmtAsts(progs)
3      tree = FPTree()
4      for ast in asts do
5          paths = getNamePaths(ast)
6          for cond, deduct in splitPaths(paths, t) do
7              tree.update(sort(cond) + sort(deduct))
8      patterns = genPatterns(tree.root, list(), t)
9      return pruneUncommon(patterns, asts)
```

- *Satisfaction*: s satisfies p if s matches p and $\forall a \in A.\ a.S = d.S \implies a.n = d.n$. Intuitively, it requires that the name subtoken prefixed by the name path d in the deduction must be equal to the correct word.
- *Violation*: s violates p if s matches p but does not satisfy p.

Figure 2(e) shows an example of a confusing word pattern.

3.3 Mining Name Patterns from Big Code

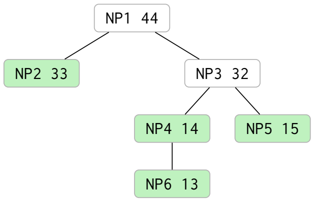
After defining name patterns, the next question then is how to obtain the most *representative* name patterns adopted by most developers. We propose to mine name patterns from a large dataset of open source repositories, i.e., Big Code. To this end, we propose an algorithm based on the frequent pattern trees (FP trees) mining methods [24, 32]. The original procedures from [24, 32] treat all items the same. In our scenario, name paths can be part of a condition or a deduction. Therefore, in our algorithm, we split name paths into condition paths and deduction paths. At a high level, the algorithm iterates through programs in the open source repositories and extracts name paths to grow the FP tree. The nodes of the FP tree stores the extracted name paths and their occurrence counts. After the growth of the FP tree, we traverse the FP tree to generate the most common name patterns based on the frequency information.

The algorithm minePatterns for mining name patterns is outlined in Algorithm 1. minePatterns takes a set of programs from open source repositories and the type of patterns to mine as input, and outputs a set of mined name patterns. First, we obtain the transformed ASTs of the program statements in the repositories by parsing the programs and applying the transformation rules (Line 2), and initialize the FP tree (Line 3). Then, we iterate through the obtained ASTs to update the FP tree (Line 4 to 7). For each AST, we extract the name paths by calling getNamePaths at Line 5. The name paths are then split (with the splitPaths function at Line 6) to condition (variable *cond*) and deduction (variable *deduct*). For the consistency name pattern, we treat pairs of name paths with the same end node as the deduction. For the confusing word name pattern, we treat name paths whose end node is equal to the correct word of a confusing pair as the deduction. The name paths not treated as the deduction are included in the condition. We iterate through all possible ways to split (Line 6), put the sorted condition and the sorted deduction into a single list, and update the FP tree with the resulted list (Line 7). During the update, we maintain the occurrence count of each node and set the flag isLast (initialized to be false) of the last node in the input list to true (not shown in Algorithm 1). The isLast flag is used later to determine when to generate name patterns. After the FP tree is constructed, we traverse its nodes to extract name patterns (Line 8) by calling a recursive algorithm genPatterns described in the next paragraph. Given the extracted name patterns (variable *patterns*), we further call pruneUncommon to prune uncommon patterns. To achieve this, we iterate through *asts* in the input dataset and count the number of matches and satisfactions for each pattern. We only keep the patterns whose ratio between the number of satisfactions and the number of matches is above a certain threshold (we used 0.8), which means that the pattern is commonly adopted in the dataset.

Algorithm 2: Generating name patterns in FP tree.

```
1  Procedure genPatterns(node, paths, t)
       Input   : node, current node in the FP tree.
                 paths, a list of visited name paths.
                 t, the type of the name patterns to mine.
       Output  : patterns, a set of mined name patterns.
2      paths.append(node.path)
3      patterns = set()
4      if node.isLast then
5          deduct = getDeduction(paths, t)
6          conds = getConditons(paths, t)
7          for cond in combinations(conds) do
8              patterns.add((cond, deduct))
9      for child in node.children() do
10         newPatterns = genPatterns(child, paths, t)
11         patterns = patterns ∪ newPatterns
12     return patterns
```

genPatterns is shown in Algorithm 2. It maintains a list of visited name paths in the function argument *paths*. First, the current name path *node*.path is appended to *paths* at Line 2 and a set of mined patterns (variable *patterns*) is initialized at Line 3. If the isLast flag of the current node is set to true (Line 4), we construct new name patterns. To achieve this, we need to assemble the deduction and condition of the new patterns from *paths*. We obtain the deduction *deduct* (by calling getDeduction on *paths* at Line 5) and a set of name paths *conds* that can be included in the condition (by



(a) An example of FP trees.

Condition	Deduction	Count
NP1	NP2	33
NP1, NP3	NP5	15
NP1, NP3	NP4	14
NP1, NP3, NP4	NP6	13

(b) Name patterns extracted by traversing the FP tree example in Figure 3(a) with Algorithm 2.

Figure 3. Examples for our mining algorithm.

calling getCondtions on *paths* at Line 6). For the consistency name pattern, we store the last two name paths of *paths* in *deduct* and the other paths in *conds*. The end nodes of the name paths in *deduct* are set to ϵ. For the confusing word name pattern, we store the last name path of *paths* in *deduct* and the other paths in *conds*. Next, we loop over all possible combinations of name paths in *conds* (Line 7) where each combination can be seen as a valid condition, and add the newly extracted pattern to the set of mined patterns (Line 8). The same procedure is performed recursively for the descendants of the current node (Line 10).

We provide two examples below which illustrate the intermediate results of the algorithms.

Example 3.10 (A constructed FP tree)**.** In Figure 3(a), we show a FP tree constructed after Line 7 of Algorithm 1 for mining confusing word name patterns. Each node represents a name path and shows its occurrence count. Green denotes nodes with the flag isLast set to true.

Example 3.11 (Results of running Algorithm 2)**.** In Figure 3(b), we show all the extracted name patterns by running genPatterns on the FP tree in Figure 3(a). This result corresponds to variable *patterns* before Line 9 of Algorithm 1.

4 Detecting Naming Issues

In this section, we describe the points-to and data flow analyses for extracting the origins of objects, and the defect classifier for pruning false positives.

4.1 Static Analyses for Name Path Context

To augment the ASTs for each statement in a program, Namer statically analyzes each source file of the program. Every file is analyzed in isolation, assuming every public method or function to be a possible entry point. Namer computes flow- and context-sensitive Andersen style points-to analysis by using k-call site sensitivity with k set by default to 5, unless for a specific program this leads to a combinatorial explosion of more than 8 contexts per method on average. Practically, this happens for a few programs in our large dataset. Our points-to analysis is implemented in Datalog. We refer the reader to [44] for definitions of points-to analysis in Datalog. Since our points-to analysis is performed on a per-file basis, any function or method defined outside the file is considered to return a fresh allocation site. As a result, the analysis is not always sound, but this is not a requirement in our setting.

Table 1. Extracted features for a violation consisting a statement s and a violated name pattern p.

Index	Description
1	Number of name paths used to represent s
2	Number of statements identical to s on the file level
3	Number of statements identical to s on the repository level
4	Satisfaction rate of p on the file level
5	Satisfaction rate of p on the repository level
6	Satisfaction rate of p over the entire mining dataset
7	Number of violations for p on the file level
8	Number of violations for p on the repository level
9	Number of violations for p over the entire mining dataset
10	Number of satisfactions for p on the file level
11	Number of satisfactions for p on the repository level
12	Number of satisfactions for p over the entire mining dataset
13	Whether p targets on object name or function name
14	Number of name paths in p's condition
15	Match ratio between p and s
16	Edit distance between the original name and the suggested name
17	Whether the original and suggested names form a confusing word pair

Using the results of the points-to computation, a dataflow analysis is performed for primitive types. This allows us for every variable, field, array access, and other action to obtain possible origins. The origin of objects is their allocation site. The origin of values is a function returning a value or $\top$ if the value was modified after its creation. When the origin sites are precisely computed (i.e. not abstracted to $\top$), this information is added to the AST. This effectively provides an abstraction where placing results of expressions into temporary variables, extracting methods, or doing other simple refactorings does not affect the augmented tree.

4.2 Defect Classifier

A violation of the name patterns indicates a potential naming issue but can result in a false positive. This is because when mining and matching name patterns, we increase the likelihood to trigger more violations so that most issues are not missed. Therefore, it is critical to prune false positives. To this end, Namer feeds the violations of name patterns into a machine learning procedure. Formally, given a program statement s and a name pattern p, Namer invokes a feature extraction function ϕ and a binary classifier ψ. Namer reports the violation consisting of s and p as a naming issue if and only if $\psi(\phi(s, p))$ = true. Intuitively, the violations classified by ψ as true are highly likely to be true naming issues so they are reported. Those classified as false are recognized by Namer as false positives so Namer abstains from reporting them. ψ can be obtained by applying off-the-shelf learning algorithms. To make the classification effective, the key then is to extract a set of expressive features with ϕ.

Features for violations. In Table 1, we show the list of features extracted by ϕ for a violation. In general, these features calculate the statistics of the violation and capture the strength of the violation. In the following, we describe the features one by one.

Feature 1-3 capture characteristics of the program statement s. Feature 1 calculates the number of name paths extracted from s. The more name paths s has, the more complex s is. Feature 2 and 3 count the number of statements identical to s in the file and the repository, respectively. More identical statements indicate a lower degree of anomaly.

Feature 4-14 measure various statistics of the violated pattern p. Feature 4-6 calculate the satisfaction rate (the number of satisfactions divided by the number of matches) of p in the file containing the statement s, in the repository containing s, and over the entire dataset used for mining the name patterns, respectively. A higher satisfaction rate indicates that the naming idiom represented by p is more commonly adopted. Feature 7-9 count the number of violations for p and feature 10-12 count the number of satisfactions for p, both over three different levels. The larger feature 7-9 are or the smaller feature 10-12 are, the more common p is. Feature 13 is a boolean feature that checks whether p targets an object name or a method name. The motivation here is that during our early experiment, we found that most of the false positives came from name patterns targeting method names as inferring correct usages of method names typically requires more contexts. Feature 14 counts the number of name paths in p's condition. The more name paths in the condition, the more difficult to match p.

Feature 15-17 jointly consider the statement s and the violated pattern p. Feature 15 is the match ratio between p and s, i.e., the number of name paths in p's condition divided by the number of name paths in s minus the number of name paths in p's deduction. The higher the match ratio is, the better p captures the structure of s, and the more likely the violation is a true naming issue. Feature 16 calculates the edit distance between the original name that violates the pattern and the suggested name. Smaller edit distance indicates a higher probability of a true naming issue (e.g., typo). Feature 17 is a boolean feature that checks whether the original name and the suggested name form a confusing word pair.

Unlike deep learning approaches [9, 28, 39] which use low-level embedding as initial features, all our features are high level, enabling our classifier to be trained with a small manually labeled dataset. Most of our features consider different levels (i.e., file, repository, and the entire dataset), as opposed to anomaly detection based approaches [34, 38, 41] which rely only on a single or few measures. This is an important design choice to make the classifier effective. As we show in § 5.5, the same feature can have different contributions to the final decision over different levels.

5 Experimental Evaluation

The evaluation of Namer focuses on four main questions:

- What is the precision of Namer on finding naming issues in open source repositories?
- What is the impact of points-to and data flow analyses (§ 4.1) and the defect classifier (§ 4.2)?
- What features are most important in the classifier?
- How does Namer compare to existing deep neural network based approaches?

5.1 Implementation and Evaluation Setup

Namer currently supports Python and Java though our framework is generic and can be applied to other languages.

Dataset. For both Python and Java, we obtained a large real world dataset of open source repositories on GitHub [3]. The Python (resp., Java) dataset contains about 1 million (resp., about 4 million) source files from 33, 144 (resp., 33, 308) repositories. We selected the set of repositories based on the highest number of stars. For finding confusing word pairs, we also used the entire histories of these repositories. Aware of code duplication on Github [35], we pruned our dataset to make it free from project forks and file-level duplicates.

Mining name patterns. To obtain name patterns, we ran the pattern mining algorithm introduced in § 3.3 over the entire dataset. The statistics of the mined patterns are discussed later in the evaluation. To avoid overfitting and strengthen generalization, we adopted standard techniques [12] to regularize name paths and patterns.

- We limited the number of name paths for a statement by simply keeping the first 10 paths.
- At Line 5 of Algorithm 1, we only returned frequent name paths that occurred more than 10 times in our dataset. As a result, over 99% of infrequent name paths were removed.
- At Line 6 of Algorithm 2, we limited the maximal number of name paths used in conditions to 10.
- At Line 9 of Algorithm 1, to prune uncommon patterns, we only returned name patterns that occurred more times than a given threshold. We found that setting the threshold to 100 for Python (500 for Java) yielded good results.

Training the classifier. To train the classifier, we need a supervised dataset where violations are manually labeled to true naming issues or false positives. In general, more training data can help the classifier perform better but the labeling process requires expensive human labor. Since we use simple models and high-level features, a small training set should be sufficient. For both Python and Java, we manually labeled 120 violations, which strikes a good tradeoff between precision and the amount of manual effort needed for building the Namer tool.

Among all labeled violations, one half were labeled true and the other half were labeled false. We leveraged cross-validation for model selection. We selected the support vector machine model with the linear kernel (the other choices are logistic regression and linear discriminant analysis) as it resulted in the best cross-validation results. We used feature standardization and principal component analysis as a preprocessing step for the features.

Testing and metric. At test time, we ran the pattern matching module of Namer over the entire Python and Java dataset excluding the samples used for training the classifier. For each dataset, we *randomly* selected 300 violations and ran the trained classifier on the selected violations to obtain the final reports of Namer. Then, we inspected all of Namer's bug reports and found that Namer is able to report various kinds of naming issues. Following [39], we classify a *naming issue* into two categories:

- *Semantic defect*: a naming issue is a semantic defect if the name in the evaluated program causes unexpected program behavior, is inconsistent with the program semantics, or may introduce errors.
- *Code quality issue*: a naming issue is a code quality issue if it is not a semantic defect but impairs code quality by confusing the readers or is inconsistent with the naming style in the file. It is strongly recommended to fix those issues to improve code readability.

If a report does not belong to the above two categories, we mark it as a *false positive*. We provide examples of semantic defects, code quality issues, and false positives in Tables 3 and 6. We calculate *precision* as the number of reported naming issues (including semantic defects and code quality issues) divided by the total number of reports. Note that unlike standard bugs (e.g., buffer overflow), naming issues are less studied and their importance lacks common understanding. Moreover, the severity of code quality issues can be subjective and varies among developers. To demonstrate that the issues reported by Namer are relevant, we conducted a small user study with professional developers, described in § 5.4.

Speed of Namer. The experiments were performed on a 28-core machine with two 2.60 GHz Intel Xeon E5-2690 CPUs and 512 GB RAM running Ubuntu 18.04. The experiments for [9] and [28] in § 5.6 require GPU so they were performed on a machine with RTX 2080 Ti GPUs.

Namer is quite performant with its runtime dominated by the program analyses described in § 4.1. On average, Namer spent 20ms for Java and 39ms for Python analyzing one source file on a single core of the test server. Because each file is analyzed separately, we parallelized the analysis on all 28 cores of the machine.

Table 2. Precision of Namer and baselines on 300 randomly selected violations from the Python dataset. "C" stands for the defect classifier and "A" stands for the static analyses.

Baseline	Report	Semantic defect	Code quality issue	False positive	Precision
Namer	134	5	89	40	70%
w/o C	300	13	124	163	46%
w/o A	88	2	50	36	59%
w/o C & A	300	12	108	180	40%

5.2 Results for Python

We now evaluate Namer on the Python dataset. We first discuss the precision of Namer. Then, we provide examples of naming issues found by Namer. At last, we show detailed statistics on the mined name patterns and the classifier.

Precision of Namer. For Python, we obtained 134 reports by running the trained classifier on the randomly selected 300 violations. The results of the manual inspection are shown in the first row of Table 2. Among the reports by Namer, we manually found 5 semantic defects, 89 code quality issues, and 40 false positives, resulting in a precision of 70%. The high precision demonstrates the effectiveness of Namer on real world Python code.

Necessity of classifier and analyses. We now investigate the impact of the static analyses and the defect classifier on the effectiveness of Namer. To this end, we constructed three baselines, by excluding the defect classifier, the analysis module, or both, from Namer. For each baseline, we similarly inspected the bug reports on 300 randomly selected violations. The inspection results are reported in rows 2–4 of Table 2. Row 2 shows the results for the baseline where the classifier was excluded from Namer, i.e., only pattern matching was performed. All 300 violations were reported as naming issues where 163 were false positives, resulting in a precision of 46%. Row 3 shows the results for the baseline for which the program analyses in § 4.1 were not performed. Compared to Namer, this baseline found fewer semantic defects and fewer code quality issues and was less precise.

From the comparison, we can see that the analyses and the classifier are both necessary parts of Namer. The analyses not only improved Namer's precision but also helped Namer find more issues. The classifier may filter out some true positives but significantly reduced the number of false positives, resulting in significantly higher precision while retaining similar recall as before, which is a desired property in practice as it allows Namer's users to spend minimal efforts on inspecting the bug reports.

Table 3. Examples of reports by Namer for Python.

Index	Reported statement	Suggested fix
	Semantic defects	
1	`self.assertTrue(vec, 4)`	`Equal`
2	`for i in xrange(10)`	`range`
3	`self.assertEquals(3, val)`	`Equal`
	Code quality issues	
4	`num_or_process = 3`	`of`
5	`def evolve(self, ..., **args):`	`kwargs`
6	`self.sz = N.array(sz)`	`np`
	False positives	
7	`self.assertTrue(os.path.islink(path))`	`exists`
8	`self.read_line_block()`	`log`

Table 4. Manual inspection of 100 Python Namer reports per pattern type with a breakdown of code quality issues.

Inspection outcome	Consistency	Confusing word
Semantic defect	1	9
Code quality issue	71	53
False positive	28	38
Breakdown of code quality issues		
Confusing name	19	14
Indescriptive name	9	6
Inconsistent name	26	23
Minor issue	15	7
Typo	2	3

Examples of reports made by Namer. We ran Namer on more statements in our Python dataset and present examples of reports in Table 3. For each example, we show the reported statement and the fix suggested by Namer:

- *Example 1* is similar to the one described in § 2 but the call has different arguments. The issue may cause unexpected runtime behavior. Namer correctly suggested to change `assertTrue` to `assertEqual`.
- *Examples 2* and *3* show the cases where Namer detects and fixes deprecated API calls. In Example 2, `xrange` is a built-in function in Python 2 but gets removed in Python 3. In Example 3, `assertEquals` is a deprecated function in the `unittest` library.
- *Example 4* contains a typo. Namer proposed a correct fix to the typo by changing `or` to `of`.
- *Examples 5* and *6* are code quality issues where the original names do not conform with the standard naming conventions. In Example 5, `args` is used for keyworded variable length arguments. However, `kwargs` should be used instead for keyworded arguments and `args` should be used for non-keyworded variable length arguments. Namer suggested the exact fix for Example 5. In Example 6, `N` is used as the abbreviation for the library `numpy`. However, `np` is commonly used and is a more informative name.
- *Examples 7* and *8* are false positives of Namer. Example 7 asserts whether a given file path in the `path` variable is a symbolic link by calling the function `islink`. However, Namer wrongly suggested to replace the function with `exists`. This is because the usage of `exists` inside `assertTrue` is more frequent than the usage of `islink` in our dataset. For Example 8, Namer suggested to replace `line` with `log` according to the violated pattern. However, `line` is not an incorrect or confusing name according to the semantics of the function (not shown here).

The above examples show that Namer can find and fix different kinds of naming issues, including those that require a deep semantic understanding of the program (e.g., correct usage of API calls).

Statistics on pattern mining and classification. In total, 65, 619 name patterns were mined for the Python dataset. 496, 306 program statements, 439, 508 source files (50% of all source files), and 30, 388 repositories (92% of all repositories) violated at least one of the mined patterns. These numbers show that the mined patterns did not represent only very rare events in our dataset.

As discussed in § 5.1, we used cross-validation for model selection. Now we report the cross-validation results for our selected model (support vector machine). We randomly took 80% of labeled samples for training the model and then tested the trained model on the remaining 20% samples. We repeated this for 30 times. The average accuracy, precision, recall, and F1 score were 81%, 81%, 81%, and 80%, respectively.

Distribution of naming issues per pattern type. In our evaluation, around 29% of the reports came from consistency name patterns, while 81% of the reports were from confusing word patterns. The sum is above 100% because 10% of the reports were detected by both patterns. To obtain more detailed statistics on the precision of Namer per pattern type, we ran Namer on all of the repositories in our dataset, randomly selected 100 new reports for each of the two name pattern types, and manually inspected the selected reports. The inspection results are shown in Table 4. We can see that the confusing word name pattern tended to recover more semantic defects, while the consistency name pattern produced fewer false positives.

Table 5. Precision of Namer and baselines on 300 randomly selected violations from the Java dataset. "C" stands for the defect classifier and "A" stands for the static analyses.

Baseline	Report	Semantic defect	Code quality issue	False positive	Precision
Namer	97	2	64	31	68%
w/o C	300	2	90	208	31%
w/o A	138	0	66	72	48%
w/o C & A	300	0	87	213	29%

5.3 Results for Java

Next, we evaluate Namer on the Java dataset. The procedures and experiments for the Java dataset were performed in the same way as those for the Python dataset.

Precision of Namer. The results of Namer on the 300 selected violations from the Java dataset are shown in the first row of Table 5. The precision was 68%, similar to the precision of Namer for the Python dataset (70%), showing that the framework is generic and can precisely detect naming issues for different types of languages.

Necessity of classifier and analyses. The last three rows of Table 5 show the results when the classification module, the analyses module, and both of them, were excluded from Namer, respectively. Namer outperformed all three baselines in precision by a large margin, reassuring the importance of the static analyses and the classifier.

Examples of reports found by Namer. We present examples of reports found by Namer for Java in Table 6:

- *Example 1* calls `getStackTrace`, which returns the stack trace information, with receiver `e` (a thrown exception). However, the return value is not assigned to any variable, which makes the call redundant. As a result, the user might miss important information about the exception. Namer suggested to fix it by calling `printStackTrace` to print the stack trace to the standard error stream.
- *Example 2* uses type `double` for loop index variable `i`, which may cause unexpected program behavior due to floating-point operations. Namer suggested to use type `int`.
- *Example 3* catches a `Throwable` variable `e`. `Exception` and `Error` are both subclasses of `Throwable`. Catching `Throwable` includes catching `Error`. However, the official documentation [2] suggests that `Error` should not be caught. Namer correctly suggested to catch `Exception` instead of `Error`.
- *Example 4–6* show cases where Namer suggested more descriptive variable names than the original names. In Example 4, Namer detected and fixed a typo where `public` is misspelled into `publick`. Example 5 is an Android API call for creating a new activity. The argument of the function `startActivity` is of type `Intent` and thus Namer suggested to use the name `intent` instead of `i`. Example 6 dismisses an Android dialog represented by the variable `progDialog`. Namer suggested to replace `prog` with `progress` to indicate that the dialog is a progress dialog.
- *Examples 7* and *8* are two false positives. In Example 7, Namer suggested to change the name `outputWriter` to `stringWriter` to better fit the class name `StringWriter`. However, according to program context, the original name is better. In Example 8, Namer suggested to replace class `ConektaObject` with `JsonObject` because `JsonObject` is used more frequently in our dataset. However, `ConektaObject` should be the correct class.

Table 6. Examples of reports by Namer for Java.

Index	Reported statement	Suggested fix
	Semantic defects	
1	`e.getStackTrace();`	`print`
2	`for(double i = 1; i < chainlength; i++) {`	`int`
3	`} catch (Throwable e) {`	`Exception`
	Code quality issues	
4	`this.publicKey = publickKey;`	`public`
5	`context.startActivity(i);`	`intent`
6	`progDialog.dismiss();`	`progress`
	False positives	
7	`final StringWriter outputWriter = ...;`	`string`
8	`ConektaObject resource = new ConektaObject();`	`Json`

Statistics on pattern mining and classification. For the Java dataset, 79, 417 name patterns were mined, which resulted in over 1.8 million violations. Moreover, 11% (i.e., 453, 450) of all the source files and 77% (i.e., 25, 496) of all the repositories had at least one violation. Similar to Python, the mined patterns had a high coverage for the Java dataset. In the cross-validation, the average accuracy, precision, recall, and F1 score were 90%, 90%, 90%, and 89%, respectively.

Distribution of naming issues per pattern type. For the Java dataset, around 14.5% of the reports came from consistency name patterns and 91.7% of the reports were from confusing word patterns. The sum is above 100%, because 6.2% of the reports were detected by both patterns. To obtain more detailed precision statistics, we ran Namer on all the repositories in our evaluation and obtained 100 new reports of each of the two bug pattern types. Then, we performed a manual inspection. For the consistency pattern, 0, 76 and 24 reports were classified as semantic defects, code quality issues and false positives, respectively. For the confusing word pattern, the number of semantic defects, code quality issues and false positives was 3, 36 and 61, respectively.

Table 7. Code quality issues selected for our user study described in § 5.4.

Issue category	Original code snippet	Detected issue & suggested fix

Inconsistent name	`if docstring is not None:`
	`  self.help = docstring`
		Rename `help` to `docstring`

Minor Issue	`def fullpath_set(self, value):`
	`  self._fullpath = value`
		Rename `value` to a more descriptive name like `fullpath`

Confusing name	`def __init__(self, song, **kwargs):`
	`  self._defaults = CODED_DEFAULTS`
	`  self._factory = song`
	`  self.set(**kwargs)`
		Change some name to avoid code like `self._factory = song`

Typo	`self.port = por`	Rename `por` to `port`

Indescriptive name	`def reset(self, *e):`
	`  self._autostep = 0`
		Rename `e` to a more descriptive name

Table 8. User study results of seven professional developers on whether to accept code quality issues reported by Namer.

Issue category	Not accepted	Accepted with IDE plugin	Accepted with pull request	Would even fix manually
Confusing name	0	3	2	2
Indescriptive name	0	3	2	2
Inconsistent name	2	0	4	1
Minor issue	2	4	0	1
Typo	1	2	1	3

5.4 User Study on Severity of Code Quality Issues

To gain more insights into the severity of code quality issues found by Namer, we performed a small user study showing 5 Python reports to professional developers from a software company with no mandatory naming style guides. The 5 reports, shown in Table 7, were chosen by randomly picking one sample from each category in the breakdown of Table 4 and were shuffled. The participants were not aware of Namer and were not told that the reports were provided by an automatic tool. They were asked if and at what conditions they would accept the changes in the reports as the project maintainers. The conditions include (i) at coding time with an automatic IDE plugin, (ii) after coding time with an automatic pull request, and (iii) the developer is willing to fix the issue manually once seeing it.

The study received 7 responses which are shown in Table 8. The results demonstrate that the developers found the issues relevant. Only in 5 cases, an issue was not accepted and in 9 cases, the developers were even willing to fix the issues themselves. For most cases, developers accepted a report only if an automatic tool like Namer, either in the form of an IDE plugin or pull requests, locates the issue and suggests a fix for them.

Table 9. Feature weights of the learned classifier.

Feature	File level	Repo level	Entire dataset
Identical statement	0.6345	-2.854	-
Satisfaction count	1.86	0.468	-0.7305
Violation count	-1.121	-1.0655	1.5565

5.5 Understanding Classifier Decision Making

Seeing the effectiveness of Namer, one natural question to ask is why the classifier performed well and based on what the classifier made decisions. Since we used a linear model on standardized features, the weights of the learned classifier are useful in measuring how much each individual feature contributes to the final decision. In Table 9, we summarize the feature weights averaged from the learned classifiers for Python and Java. The three rows show the weights of feature 2-3, 4-6, and 7-9, respectively.

We can see that the absolute values of all weights were non-negligible, meaning that the classifier makes decisions based on various features. Moreover, the classifier learned interesting behaviors non-obvious to a human: the contribution of the same type of feature can be completely opposite over different levels. For example, for the violation count feature, the weights over file and repository level were negative but over the entire dataset, the weight was positive. This phenomenon demonstrates that jointly considering both local and global features is another key reason for the classifier to precisely distinguish between true issues and false positives. The above observations are in contrast to previous works [34, 38, 41] which involves only one or few weak predictors, and can give insights for future bug finding tools.

5.6 Comparison with Deep Learning Approaches

We now compare Namer with two state-of-the-art deep learning based approaches for finding name based issues,

Table 10. Precision comparision of GGNN, Great and Namer on 134 randomly selected reports for Python.

System	Semantic defects	Code quality issues	False positives	Precision
GGNN	1	20	113	16%
Great	2	9	123	8%
Namer	5	89	40	70%

Table 11. Precision comparision of GGNN, Great and Namer on 97 randomly selected reports for Java.

System	Semantic defects	Code quality issues	False positives	Precision
GGNN	2	7	88	9%
Great	2	3	92	5%
Namer	2	64	31	68%

GGNN [9] and Great [28]. Both works leverage deep neural networks over program graphs to detect misuses of variable names and reported high accuracy on testing datasets with *synthetic* bugs. However, the high testing accuracy does not necessarily reflect their capability of finding real world issues. This is because the distribution of true issues is not necessarily the same as the distribution of synthetic issues used to train and test the network models. In fact, both works presented only limited evaluation results on finding real problems in code. The authors of GGNN presented case studies on bugs but did not include quantitative measures. Great's precision dropped to 29.4% on a dataset of only hundreds of buggy and non-buggy programs that the authors collected by comparing Github commits. Therefore, the practical effectiveness of GGNN and Great on finding real world issues is still unknown. Through our evaluation, we aim to understand the practical effectiveness of these works and compare their precision with Namer.

Training and measuring accuracy. We re-implemented the pipeline of GGNN and Great based on the machine learning modules open sourced by the authors [4, 5]. To obtain training program graphs, we followed the original works [9, 28] to introduce synthetic changes to the programs in our Python and Java datasets. Note that it is impossible to train GGNN and Great with real issues as we did with Namer. This is because deep networks are data hungry and a dataset with a sufficient amount of real issues does not exist and is hard to obtain with either manual labor or automation. The training processes took around 70h to 130h.

After training, we measured the accuracy of GGNN and Great to ensure that each individual model can achieve high accuracy as reported in the original papers [9, 28]. We did not aim to compare the accuracy across those models. For each model, we sampled a subset of programs from our dataset, introduced synthetic changes to them, and constructed 10, 000 new program graphs. The new graphs were not used in training though the original programs could be. Then, we fed the graphs into the trained models and obtained the accuracy. For GGNN, the accuracy was 71% (resp., 83%) for Python (resp., Java). For Great, the classification accuracy, localization accuracy and repair accuracy for Python (resp., Java) were 91%, 83%, and 79% (resp., 91%, 82%, and 81%), respectively.

From these results on accuracy, we could reproduce the fact that GGNN and Great result in high accuracy on the dataset of synthetic code changes. Further, these numbers closely match the results in the original papers [9, 28].

Precision of detecting issues. Next, we evaluate GGNN and Great on finding real world issues. To this end, we tested GGNN and Great on our dataset without any synthetically introduced changes. Same as Namer, we inspected a randomly selected set of issue reports for GGNN and Great. All true variable misuses were classified as semantic issues as they usually cause unexpected program behaviors. We found that some reports were not variable misuses but pointed to other kinds of naming issues, e.g., unused variables. We included them as true positives and classified them into semantic issues and code quality issues based on our criterion in § 5.1.

A factor that affects the precision and recall of a bug finding tool is the confidence level above which a report is made. For GGNN and Great, we could not determine the total number of true issues that can be theoretically reported by the respective models in comparison to the total number of issues discoverable by Namer, but based on a sample of the frequency of the issues in code, we estimated that a name misuse predictor should be applicable to at least one fifth of the naming problems. To compensate for this effect, we tuned the confidence levels so that both GGNN and Great reported around 5× fewer issues than Namer.

The results on randomly inspected reports for Python are shown in Table 10. Both GGNN and Great found very few true issues, resulting in the reported low precision: GGNN achieved 16% precision with 1 semantic defect and 20 code quality issues; Great achieved 8% precision with 2 semantic defects and 9 code quality issues. This low detection capabilities of the networks do not mean that there are no true issues of the specific kind, it just shows that the probability distribution of issues encoded in the predictor did not correlate with that of real naming issues. As shown in Table 11, the networks also achieved low precision for Java (9% for GGNN and 5% for Great). We tried to increase the confidence levels of GGNN and Great to the maximum level, i.e., inspecting the most confident reports, as done by [9, 39]. This enabled GGNN and Great to find more semantic issues and increased their precision to ~40%. However, with such high confidence levels, the recall became nearly zero, which is not desired in practice, and the precision was still non-satisfactory.

Compared to GGNN and Great, Namer has a much higher precision of ~70%, despite returning 5× more reports. Another important aspect is that the naming rules mined by Namer and the decision making process of the classifier are highly interpretable, while neural code models are non-interpretable and are vulnerable to adversarial attacks [16]. As a result, developers can inspect and better understand the issues reported by Namer. Therefore, we conclude that Namer is practically more effective than GGNN and Great.

6 Related Work

In this section, we discuss works mostly related to ours.

6.1 Naming issue detection and repair

Existing techniques on naming issue detection can be categorized into two kinds: rule-based and learning-based. Namer can be seen as a combination of the two types of techniques.

Rule-based approaches. The works of [30, 34, 38] leverage rule-based anomaly detection for finding naming issues. The detection solely depends on extracted rules [30] or similarity measures [34, 38], while Namer's decision process considers more factors and features. The work of [41] uses a graph matching algorithm to detect argument selection defects in an industry level codebase. The above approaches only focus on a specific type of naming issue (wrong usages of function arguments) and a specific language (Java). On the contrary, Namer can detect more kinds of naming issues and targets both Java and Python.

Learning-based approaches. A number of recent works propose machine learning techniques for name-based bug finding. Apart from GGNN [28] and Great [28] discussed in § 5.6, GINN [47] is another graph neural network for code that leverages a hierarchy of intervals over the control flow graph for better message passing. For detecting variable misuses, GINN achieves higher accuracy than GGNN and Great, but suffers from the same limitation: synthetic issues are used for training. DeepBugs [39] learns embeddings for code snippets to detect three types of naming bugs for Javascript: swapped arguments, wrong binary operator, and wrong binary operand. DeepBugs achieves high precision on the most confident reports (i.e., low number of reports). It is unclear how effective DeepBugs is in a realistic scenario where a higher number of reports is desired.

Instead of detecting bugs, other works learn probabilistic models for predicting or suggesting identifier names. Probabilistic graphical models are employed in [15, 27, 40] for predicting identifier names for minified Javascript program, obfuscated Android applications and stripped binaries, respectively. Naturalize [8] suggests identifier names by learning statistical language models representing coding conventions. Convolutional attention networks are used in [10] for code summarization, i.e., method name suggestion. Other notable works [11, 13] use paths between AST nodes as input features to a neural network trained to summarize code and predict descriptive method names.

6.2 Machine Learning and Program Reasoning

In the following, we discuss research at the intersection of machine learning and program reasoning.

Static analysis. Bayesian optimization [36] and reinforcement learning [29, 43] have been used for balancing the tradeoff between precision and performance for numerical program analysis. API specifications are a key ingredient in modern static analysis. Atlas [14] leverages active learning to obtain points-to specifications. *Uspec* [22] learns API aliasing specifications with unsupervised learning. Seldon [20] infers taint specifications by solving linear optimization. Statistical methods have been used to rank and classify alerts from static analyzers [25, 31].

Testing. Machine learning has been adopted to improve test input generation. Skyfire [46] learns from an input corpus and input grammar for generating well-distributed seed inputs for fuzzing. AFLFast [17] learns a markov chain for modeling program branching behavior to guide input generation. The work of [42] learns feed-forward networks from already generated fuzzing inputs to guide future mutation. ILF [26] learns gate recurrent units from input traces generated by symbolic execution. The work of [23] learns recurrent neural networks from an existing test corpus.

7 Conclusion

We proposed a novel approach for finding and fixing naming issues. Our method combines interpretable name patterns mined from Big Code to represent different kinds of naming idioms together with a machine learning classifier that filters false positives, trained on expressive features. Importantly, making use of Big Code does not require labeling the programs (which would be practically infeasible), while we only require a small amount of labeled data to train the classifier.

We implemented our approach in a system called Namer which supports both Python and Java, and evaluated it extensively on a massive dataset of open source repositories. Our experimental results showed that Namer is able to find real world naming issues and suggest correct fixes with high precision (~70%). We also showed that Namer is practically more effective than state-of-the-art deep neural network approaches (that use synthetic data).

We believe this work presents new insights in building a practical bug finding tool with machine learning. It also reassures the importance of program abstraction and analyses in finding defects and other issues in real world code.

References

[1] 2020. American fuzzy lop. https://lcamtuf.coredump.cx/afl/

[2] 2020. Error (Java SE 14 & JDK 14). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Error.html

[3] 2020. GitHub. https://github.com

[4] 2020. ICLR20-Great. https://github.com/VHellendoorn/ICLR20-Great

[5] 2020. tf-gnn-samples. https://github.com/microsoft/tf-gnn-samples

[6] 2020. unittest — Unit testing framework. https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertTrue

[7] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *FSE 2015*. https://doi.org/10.1145/2786805.2786849

[8] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning Natural Coding Conventions. In *FSE 2014*. https://doi.org/10.1145/2635868.2635883

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR 2018*. https://openreview.net/forum?id=BJOFETxR-

[10] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML 2016*. http://proceedings.mlr.press/v48/allamanis16.html

[11] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR 2019*. https://openreview.net/forum?id=H1gKYo09tX

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *PLDI 2018*. https://doi.org/10.1145/3192366.3192412

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *PLDI 2018*. https://doi.org/10.1145/3192366.3192383

[15] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *CCS 2016*. https://doi.org/10.1145/2976749.2978422

[16] Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. In *ICML*. http://proceedings.mlr.press/v119/bielik20a.html

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS 2016*. https://doi.org/10.1145/2976749.2978428

[18] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *CSMR 2010*. https://doi.org/10.1109/CSMR.2010.27

[19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI 2008*. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[20] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable Taint Specification Inference with Big Code. In *PLDI 2019*. https://doi.org/10.1145/3314221.3314648

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*. https://doi.org/10.1145/512950.512973

[22] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised Learning of API Aliasing Specifications. In *PLDI 2019*. https://doi.org/10.1145/3314221.3314640

[23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE 2017*. https://doi.org/10.1109/ASE.2017.8115618

[24] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD 2000*. https://doi.org/10.1145/342009.335372

[25] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. In *MSR 2014*. https://doi.org/10.1145/2597073.2597100

[26] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *CCS 2019*. https://doi.org/10.1145/3319535.3363230

[27] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *CCS 2018*. https://doi.org/10.1145/3243734.3243866

[28] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *ICLR 2020*. OpenReview.net. https://openreview.net/forum?id=B1lnbRNtwr

[29] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware Program Analysis via Online Abstraction Coarsening. In *ICSE 2019*. https://doi.org/10.1109/ICSE.2019.00027

[30] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *ECOOP 2009*. https://doi.org/10.1007/978-3-642-03013-0_14

[31] Ted Kremenek and Dawson R. Engler. 2003. Z-Ranking: using statistical analysis to counter the impact of static analysis approximations. In *SAS 2003*. https://doi.org/10.1007/3-540-44898-5_16

[32] Carson Kai-Sang Leung, Laks V. S. Lakshmanan, and Raymond T. Ng. 2002. Exploiting Succinct Constraints using FP-trees. *SIGKDD Explorations* 4, 1 (2002), 40–49. https://doi.org/10.1145/568574.568581

[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. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[34] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: Exploring and Exploiting Similarities between Argument and Parameter Names. In *ICSE 2016*. https://doi.org/10.1145/2884781.2884841

[35] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 84:1–84:28. https://doi.org/10.1145/3133908

[36] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. In *OOPSLA 2015*. https://doi.org/10.1145/2814270.2814309

[37] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring crypto API rules from code changes. In *PLDI 2018*. https://doi.org/10.1145/3192366.3192403

[38] Michael Pradel and Thomas R. Gross. 2011. Detecting Anomalies in the Order of Equally-typed Method Arguments. In *ISSTA 2011*. https://doi.org/10.1145/2001420.2001448

[39] Michael Pradel and Koushik Sen. 2018. DeepBugs: a Learning Approach to Name-based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[40] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *POPL 2015*. https://doi.org/10.1145/2676726.2677009

[41] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 104:1–104:22. https://doi.org/10.1145/3133928

[42] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *S&P 2019*. https://doi.org/10.1109/SP.2019.00052

[43] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *CAV 2018*. https://doi.org/10.1007/978-3-319-96145-3_12

[44] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[45] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *ICLR 2019*. https://openreview.net/forum?id=ByloJ20qtm

[46] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *S&P 2017*. https://doi.org/10.1109/SP.2017.23

[47] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27. https://doi.org/10.1145/3428205