ANALYZING CODE CORPORA TO IMPROVE THE CORRECTNESS AND RELIABILITY OF PROGRAMS

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Jibesh Patra

aus Bankura, Indien

Hauptberichter: Prof. Dr. Michael Pradel
Mitberichter: Prof. Dr. Earl T. Barr
Tag der mündlichen Prüfung: 18 May, 2021

Institut für Software Engineering der Universität Stuttgart
2021

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Stuttgart, Deutschland, März 2021

Jibesh Patra

ACKNOWLEDGMENTS

My Ph.D. days will remain one of the most memorable and cherished times in my life. I feel very fortunate to come across incredibly smart and down-to-earth people who taught me many things on the way.

First and foremost, I want to thank my adviser Michael Pradel for his constant patience with me and for guiding me at every step. Michael gave me ample opportunities to learn and grow for which I will be forever grateful.

Special thanks to the members of my Ph.D. committee Andrés Bruhn, Earl Barr & Steffen Staab for their insightful questions, comments, and feedback.

I also want to thank Rahul Sharma, Alexander Gaunt, Miltos Allamanis & Marc Brockschmidt for providing me the opportunity to intern at Microsoft Research in Bangalore and Cambridge. Both of these internships introduced me to new research directions and helped me grow as a researcher.

My Ph.D. journey would not have been so enjoyable without the lab environment I received. This was made possible by my labmates Cristian-Alexandru Staicu, Marija Selakovic, Marina Billes, Andrew Habib, Daniel Lehmann, Luca Di Grazia, Moiz Rauf, Aryaz Eghbali, & Matteo Paltenghi. We had fun discussions and debates over lunch which made the entire journey particularly interesting. I want to thank my collaborators Rabee Sohail Malik, Satia Herfert, Junjie Chen, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang for the successful collaborations and insightful discussions during meetings. I also want to thank our secretary Andrea Püchner who made my initial days in Germany very smooth and Katharina Plett who helped me among others to translate the abstract of this thesis into German.

My biggest support came from my family, in particular, my wife Supriti who would always encourage me no matter how down I felt and my daughter Ahana who has the superpower of making everything right with her smile and her kiss. I want to thank my parents Tarasankar Patra, Rekha Patra, parents-in-law Mrinal Kanti Sinhamahapatra, Arati Sinhamahapatra for constantly encouraging me and my cousin Chinmoy Patra for inspiring me to pursue a Ph.D.

CONTENTS

ABSTRACT

Bugs in software are commonplace, challenging, and expensive to deal with. One widely used direction is to use program analyses and reason about software to detect bugs in them. In recent years, the growth of areas like web application development and data analysis has produced large amounts of publicly available source code corpora, primarily written in dynamically typed languages, such as Python and JavaScript. It is challenging to reason about programs written in such languages because of the presence of dynamic features and the lack of statically declared types.

This dissertation argues that, to build software developer tools for detecting and understanding bugs, it is worthwhile to analyze code corpora, which can uncover code idioms, runtime information, and natural language constructs such as comments. The dissertation is divided into three corpus-based approaches that support our argument. In the first part, we present static analyses over code corpora to generate new programs, to perform mutations on existing programs, and to generate data for effective training of neural models. We provide empirical evidence that the static analyses can scale to thousands of files and the trained models are useful in finding bugs in code. The second part of this dissertation presents dynamic analyses over code corpora. Our evaluations show that the analyses are effective in uncovering unexpected behaviors when multiple JavaScript libraries are included together and to generate data for training bug-finding neural models. Finally, we show that a corpus-based analysis can be useful for input reduction, which can help developers to find a smaller subset of an input that still triggers the required behavior.

We envision that the current dissertation motivates future endeavors in corpus-based analysis to alleviate some of the challenges faced while ensuring the reliability and correctness of software. One direction is to combine data obtained by static and dynamic analyses over code corpora for training. Another direction is to use meta-learning approaches, where a model is trained using data extracted from the code corpora of one language and used for another language.

ZUSAMMENFASSUNG

Softwarefehler sind alltäglich, herausfordernd und teuer zu beheben. Es ist weit verbreitet Programmanalysen anzuwenden und die Software genau auszuwerten, um diese Fehler zu entdecken. In den letzten Jahren hat das Wachstum von Bereichen wie Web-Anwendungsent-wicklung und Datenanalyse große Mengen an öffentlich verfügbaren Quellcode-Korpora hervorgebracht, die hauptsächlich in dynamisch typisierten Sprachen wie Python und JavaScript geschrieben sind. Aufgrund des Vorhandenseins dynamischer Eigenschaften und des Fehlens statisch deklarierter Typen ist es eine Herausforderung solche Programme auszuwerten, die in solchen Sprachen geschrieben wurden.

In dieser Dissertation wird argumentiert, dass es für die Erstellung von Programmentwicklungswerkzeugen zum Erkennen und Verstehen von Fehlern sinnvoll ist, Code-Korpora zu analysieren, die Code-Idiome, Laufzeitinformationen und natürlichsprachliche Konstrukte wie Kommentare aufdecken können. Die Dissertation gliedert sich in drei korpusbasierte Ansätze, die unsere Argumentation unterstützen. Im ersten Teil stellen wir statische Analysen von Code-Korpora vor, um neue Programme zu generieren, Mutationen an bestehenden Programmen durchzuführen und Daten für ein effektives Training von neuronalen Modellen zu erzeugen. Wir liefern empirische Beweise dafür, dass die statischen Analysen auf Tausende von Dateien skalieren können und die trainierten Modelle beim Auffinden von Fehlern im Code nützlich sind. Der zweite Teil dieser Dissertation stellt dynamische Analysen von Code-Korpora vor.

Unsere Auswertungen zeigen, dass die Analysen effektiv sind, um unerwartete Verhaltensweisen aufzudecken, wenn mehrere JavaScript-Bibliotheken zusammen eingebunden sind, und um Daten für das Trainieren von neuronalen Modellen zur Fehlerfindung zu erzeugen. Schließlich zeigen wir, dass eine korpusbasierte Analyse für die Inputreduktion nützlich sein kann, was Entwicklern helfen kann, eine kleinere Teilmenge an Input zu finden, der das gewünschte Verhalten abruft.

Wir hoffen, dass die vorliegende Dissertation zukünftige Bestrebungen im Bereich der korpusbasierten Analyse motiviert, um den Herausforderungen entgegen zu wirken und die Zuverlässigkeit und Fehlerfreiheit von Software zu gewährleisten. Eine Möglichkeit ist Daten zu kombinieren, die aus statischen und dynamischen Analysen von Code-Korpora für das Training gewonnen wurden. Eine andere Möglichkeit ist die Verwendung von Meta-Learning-Ansätzen, bei denen ein Modell mit Daten trainiert wird, die aus den Code-Korpora einer Sprache extrahiert und für eine andere Sprache verwendet werden.

1

INTRODUCTION

Bugs in software are expensive [14], unavoidable, and present a key challenge in software development. It has been estimated that software developers spend almost half of their time debugging programs which equates to $312 billion per year [92]. As a result, to aid developers, there has been a large body of research [178] on the development of automated tools for fast and precise detection of software bugs. Languages such as JavaScript and Python have become popular[1] in recent years because of their ease of use and the availability of large number of libraries. For example, JavaScript has become the de facto language for web applications, and Python for data analysis and machine learning-based applications. This calls for increased attention towards designing approaches and developing tools that ensure the correctness and reliability of software written in such languages.

Program analysis is a widely used approach to find bugs in software. Such bug detecting approaches for languages such as Python and JavaScript either check for commonly made mistakes based on pre-defined rules or perform sophisticated analyses of programs. These approaches can be broadly classified into two groups:

- **Static analysis-based approaches:** Such approaches reason about programs without actually executing them. As a result, static analysis approximates the runtime behavior and can produce false positives. For JavaScript, static analysis tools that help with code quality and report commonly made mistakes are JSHint [59], ESLint [268], etc. and for Python there are tools such as Pyre [269], flake8 [270].
- **Dynamic analysis-based approaches:** Such approaches analyze programs during execution and reason about program behavior

1 https://www.tiobe.com/tiobe-index/

based on runtime values. In practice, only a subset of program code gets executed at runtime, which can cause dynamic analysis to miss many legitimate cases. For JavaScript, approaches such as DLint [132] and TypeDevil [142] use dynamic analysis to report warnings about code quality and for Python there is an approach by Xu et al. [172] that finds bugs using program traces.

In addition to the above two broad categories, some approaches like JSNose [93] for JavaScript, also combine static and dynamic analysis into a hybrid analysis to reason about program behavior.

Because of the popularity of JavaScript and Python, large code corpora written in such dynamically typed languages are available and instead of relying on pre-defined rules to uncover mistakes, such corpora may be leveraged to extract code idioms, which can aid in finding bugs. This dissertation argues that:

Analyzing large code corpora of dynamically typed languages provides opportunities to uncover repeated patterns that aid in training learning-based approaches and solve software engineering problems such as detecting inconsistencies, bugs in code, and reduction of test inputs.

In this dissertation, we show why dynamically typed languages such as JavaScript and Python are particularly prone to errors and present six approaches to support our thesis, that analyzing code corpora aids in solving software engineering problems. The goal of this dissertation is to use program analysis and novel learning-based techniques to alleviate some of the challenges faced for ensuring the correctness and reliability of programs. In particular, we address the following three key challenges.

- (C-I) *Generating a large number of valid and realistic source code examples*: Automatically generated source code examples provide a way to improve the quality and ensure the correctness of software. The generated examples can be useful in two key ways. First, the examples can be used to test programs that consume such source code as input. Second, the generated examples can be useful for training neural models, where the downstream tasks can be bug finding, code smell detection, etc.

  For both of the use cases mentioned above, the generated source code examples need to be valid and realistic. For the first use

case to test other programs, the generated examples need to be correct and exercise the program to find bugs. It is challenging because the generation needs some form of reasoning about the target program under test. For example, if the target is a JavaScript engine that accepts JavaScript programs, generating random strings as input to the engine would likely not exercise it enough to find bugs. A better approach is to generate at least syntactically valid JavaScript programs or generate programs that do not crash during execution, which is challenging. Similarly, for the second use case to train an effective neural classifier for specific downstream tasks using generated source code examples also need to be correct and realistic which again need some form of understanding about the task.

- (C-II) *Reasoning about programs written in dynamically typed languages*: Reasoning about programs is a way to mitigate mistakes in programs. The two widely used program reasoning approaches are static and dynamic analysis. Reasoning about programs is particularly challenging for languages such JavaScript or Python because of the lack of statically declared types, which causes making assumptions about behavior of a program at a given point difficult.

- (C-III) *Reducing test inputs*: The third challenge is related to the first in the sense that once a test input has triggered an unexpected behavior, it is useful to find a subset of the input that still triggers the same behavior. This is helpful for developers who want to understand and fix the bug. For example, suppose we have an input JavaScript program that triggers a fault in the JavaScript engine. A challenge now is to find a smaller or reduced JavaScript program that still triggers the same fault thereby help in debugging of the engine. It is difficult to reduce a test input because the behavior gets triggered at runtime and finding a subset of the program that gets executed and triggers the bug is not always straightforward.

1.1 OUTLINE AND CONTRIBUTIONS

This section provides an outline of the dissertation and summarizes our main contributions. The dissertation is divided into three parts: 1) Corpus-based static analysis to detect software bugs, 2) Corpus-based

dynamic analysis to detect software bugs, 3) Corpus-based input reduction. Each of the three parts is divided into chapters that each represent an article. Figure 1.1 provides an overview of the outline.

1.1.1 *Corpus-based Static Analysis to Find Software Bugs*

Statically analyzing source code can uncover many interesting features about programs such as frequent code idioms, function signatures. We use static analysis to generate new programs, to seed bugs in programs, and to obtain data for training neural models.

TREEFUZZ In Chapter 2, we present TreeFuzz, a language-independ ent, blackbox fuzz testing approach that generates tree-structured data, in our case JavaScript programs or HTML documents. The core idea is to statically analyze a corpus of example programs to infer a set of probabilistic, generative models, which then create new programs that has properties similar to the corpus.

Primary Contributions: In TreeFuzz, we address the challenge (C-I) of generating large numbers of valid and realistic source code examples. Our evaluation shows that 96% of the TreeFuzz-generated programs are syntactically correct. Additionally, TreeFuzz is the first language-independent, blackbox fuzz testing approach that enables testing a variety of programs that expect structured input data. We specifically evaluate on JavaScript engines and uncover various inconsistencies among browsers, including browser bugs and unimplemented language features.

SEMSEED Chapter 3 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.

Primary Contributions: Similar to TreeFuzz, SemSeed also addresses the challenge (C-I) of generating large numbers of valid and realistic source code examples. We find that in 97% of the cases, a SemSeed-induced mutation results in a syntactically correct program. SemSeed is the first to use learned token embeddings for mutating programs and seed bugs. The bug-seeded programs, along with the original

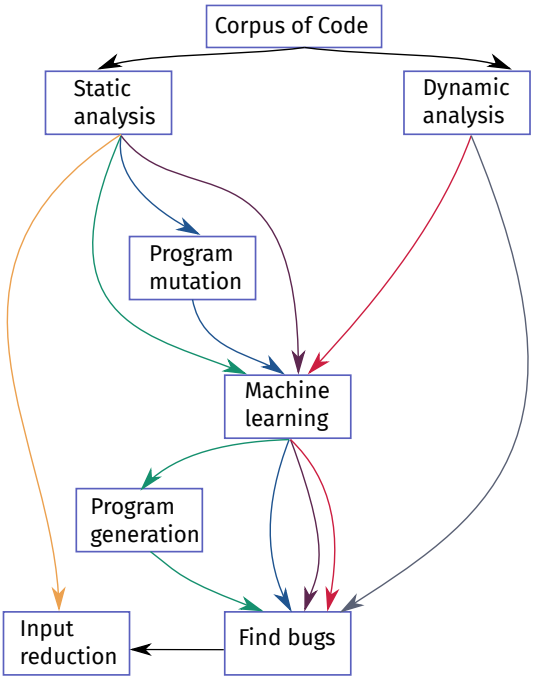


Figure 1.1: Overview of the contributions and the connections between them.

correct programs, are useful for training more effective neural bug detection models.

NL2TYPE In Chapter 4 we provide details of our approach called NL2Type that addresses the lack of type annotations in dynamically typed languages. We analyze a corpus of JavaScript code to extract information, such as comments, function names, and parameter names. We use the extracted information to train a neural model that helps annotating not yet annotated JavaScript code by suggesting types to the developer.

Primary Contributions: NL2Type addresses the second challenge (C-II) of reasoning about programs specifically for languages that are dynamically typed. In particular, we provide empirical evidence that natural language information, such as comments, are also valuable in reasoning about programs. The trained model based on the information extracted from a JavaScript corpus is able to identify inconsistencies in existing type annotations. The main contribution of the author of this dissertation in this project has been to co-design the approach and to implement the static analysis that extracts the relevant information for training the classifier.

1.1.2 *Corpus-based Dynamic Analysis to Find Software Bugs*

While static analysis can provide many interesting properties about programs, it is still an approximation of the actual behavior. More precise information is obtained by analyzing the runtime behavior of programs also known as dynamic analysis. We use dynamic analysis to uncover bugs and obtain data for training neural classifiers.

CONFLICTJS Chapter 5 presents ConflictJS, an approach to analyze a corpus of code, in this case a collection of JavaScript libraries for conflicts. Due to the lack of namespaces in JavaScript, libraries when included together all share the same global namespace. As a result, one library may inadvertently modify or even delete the APIs of another library, causing unexpected behavior of library clients that we call as conflicts. ConflictJS finds and validates conflicts between JavaScript libraries in two steps. At first, a dynamic analysis of individual libraries identifies pairs of potentially conflicting libraries. Then, targeted test synthesis validates potential conflicts by creating a client application.

Primary Contributions: With ConflictJS, we address the challenge (C-II) of reasoning about programs written in dynamically typed languages using runtime values. We find that dynamic analysis is useful in detecting bugs and inconsistencies between JavaScript libraries. Additionally, ConflictJS is the first to address the problem of conflicts among libraries in a language such as JavaScript without explicit namespaces.

NALIN Chapter 6 presents Nalin, a technique to automatically detect name-value inconsistencies. The approach dynamically analyzes a corpus of Python programs to track assignments of values to names and then trains a neural machine learning model that predicts whether a name and a value fit together.

Primary Contributions: Like ConflictJS, Nalin also addresses the challenge (C-II) of reasoning about programs written in dynamically typed languages using runtime values. We find the dynamic analysis to be useful in training neural models that can detect bugs in real-world code. Nalin is also the first approach to find coding issues through machine learning on runtime behavior. We provide empirical evidence of the effectiveness of the approach that finds name-value inconsistencies in real-world code with a reasonable precision.

1.1.3 *Corpus-based Input Reduction*

GTR Chapter 7 shows that in addition to bug finding, large corpora of code may be leveraged for other tasks such as reducing test inputs. We present an effective technique called Generalized Tree Reduction algorithm (GTR), to reduce arbitrary test inputs that can be represented as a tree, such as program code, PDF files, and XML documents. The efficiency of input reduction is increased by learning transformations from a corpus of example data.

Primary Contributions: With GTR, we address the challenge (C-III) of reducing test inputs. Our evaluation suggest that the presented approach is significantly more effective and efficient than two state-of-the-art techniques. The main contribution of the author of this dissertation is to co-design the approach and the implementation of the experiment where we successfully demonstrate GTR's effectiveness on JavaScript programs.

1.2 LIST OF ARTICLES

The current dissertation is based on the following articles that are either published or under consideration at some venue. In the following, we list each chapter and the corresponding article form where it has been adapted:

- *Chapter 2*: Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data [165], Technical Report, Distinguished Poster Award at the European Conference on Object-Oriented Programming (ECOOP), 2016
- *Chapter 3*: Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs, Distinguished Paper Award, European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021
- *Chapter 4*: NL2Type: Inferring JavaScript Function Types from Natural Language Information [240], International Conference on Software Engineering (ICSE), 2019
- *Chapter 5*: ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries [224], International Conference on Software Engineering (ICSE), 2018
- *Chapter 6*: Learning from Runtime Behavior to Find Name-Value Inconsistencies, Under Submission
- *Chapter 7*: Automatically Reducing Tree-Structured Test Inputs [190], International Conference on Automated Software Engineering (ASE), 2017
- *Not part of this dissertation*: A Survey of Compiler Testing [251], ACM Computing Surveys (CSUR), 2020

We also make the implementations of our most recent approaches public. Table 1.1 shows the mapping from each chapter to the publicly hosted implementation.

Table 1.1: Mapping between chapters and the corresponding implementation.

Article	Implementation
Chapter 3	https://github.com/sola-st/SemSeed
Chapter 4	https://github.com/sola-da/NL2Type
Chapter 5	https://github.com/sola-da/ConflictJS
Chapter 6	https://github.com/sola-st/Nalin
Chapter 7	https://github.com/sherfert/GTR

Part I

CORPUS-BASED STATIC ANALYSIS TO FIND SOFTWARE BUGS

Statically analyzing source code can uncover many interesting features about programs. We use static analysis to generate new programs, to seed bugs in programs and to obtain data for training neural models.

2

TREEFUZZ: LEARNING PROBABILISTIC MODELS OF INPUT DATA FOR FUZZ TESTING

Generating test inputs is one of the key ways to find bugs in software. To find bugs in complex programs, complex input data is required that is not only valid but also realistic. The approach of generating random input data to test programs is also known as fuzz testing. Fuzz testing has been successfully applied, e.g., to compilers [81], runtime engines [39, 84], refactoring engines [33], office applications [40], and web applications [68]. In Chapter 1, we discuss the challenge (C-I) of generating test inputs, i. e., it is difficult to generate a large number of valid and realistic source code examples. The current chapter presents a corpus-based approach to address this challenge. Our approach, called TreeFuzz, learns probabilistic models by statistically analyzing a corpus of tree-structured input data such as JavaScript programs (represented as ASTs), HTML documents. After learning, TreeFuzz generates a large number of valid and realistic tree-structured data which are useful for finding bugs.

2.1 MOTIVATION

Existing approaches that try to alleviate the challenge (see C-I in Chapter 1) of generating valid programs or that comply with specific input format roughly fall into three categories. First, format-specific approaches, such as Csmith [81] and FLAX [68], target software that processes one particular input format and rely on built-in knowledge of this format. Manually creating a format-specific fuzzer is a time-consuming and strongly heuristic effort that cannot be easily adapted to other formats and even newer versions of the same format. Yang et al., who created the popular Csmith compiler testing tool, report that it took "substantial manual tuning of the 80 probabilities that

govern Csmith's random choices" to "make the generated programs look right" [81]. Second, whitebox approaches analyze the program under test to generate input that triggers particular paths, e.g., based on symbolic execution. SAGE [40] and BuzzFuzz [50] are examples of whitebox fuzzing approaches. Unfortunately, the assumption made by these approaches that the tested program is available at input generation time is not always given, e.g., when creating inputs for differential testing across multiple supposedly equivalent programs [12] or when fuzz testing remote web applications. Moreover, whitebox techniques often suffer from scalability issues because they reason about an exponential number of execution paths. Third, grammar inference-based approaches, such as Glade [180], first learn a context-free grammar and then randomly sample the grammar to create new input. While effective at testing the parsing component of a program, these approaches are limited by the expressiveness of context-free grammars and typically produce data that violates constraints of the input format that cannot be captured by a context-free grammar.

A stream of research investigates probabilistic models of data, in particular source code, and how to learn such models from a corpus of examples. For example, n-gram-based models [83, 104], graph-based models [140], and models based on probabilistic higher-order grammars [149, 167] have been explored. These models have been shown to be useful for code completion but none of them has been used for generating new data from scratch, as required for fuzz testing. Work by Godefroid et al. [187], applies recurrent neural networks to learn a model of input data for fuzz testing. However, their approach focuses on fuzzing the parser component of a program under test, not on generating syntactically correct data that reaches deeper into the program.

This chapter merges two streams of research, fuzz testing and learning probabilistic models of structured data, into a novel approach for learning how to test complex programs by learning from examples of input data. We focus on input data that can be represented as a labeled, ordered tree. Our approach, called *TreeFuzz*, learns models of such input data by traversing each tree once while accumulating information. For each node and edge in the tree, TreeFuzz gathers facts that explain why the node or edge has a particular label and appears at a particular position in the tree. After having traversed all data, the approach summarizes the gathered information into

probabilistic models. Finally, the learned models guide TreeFuzz while generating new trees in a depth-first manner.

Our approach provides several benefits:

- *Format-independent*. By considering tree-shaped data, we abstract away details of specific input formats and support a wide range of data formats that can be transformed into trees. For example, TreeFuzz can be applied to source code (represented as an AST), documents (PDF, ODF, HTML), and images (SVG, JPG). To support a specific format, our implementation relies on a parser that transforms data into a tree and a pretty printer that transforms fuzz-generated trees into data. Parsers and pretty-printers are available for many input formats and otherwise could be created based on existing grammar-inference techniques [158, 180].

- *Fully automatic*. In contrast to format-specific approaches, TreeFuzz relies neither on built-in knowledge about the input format nor on any manual intervention while learning from examples and generating new data.

- *Extensible set of models*. Instead of focusing on a single probabilistic model, we design TreeFuzz as a framework that supports an extensible set of models. Each model describes a particular aspect of the input format. We describe six models in this chapter. For example, one of these models suggests child nodes based on parent nodes, similar to a probabilistic context-free grammar. Another model suggests node labels in a way that enforces definition-use-like relationships between subtrees of a generated tree. During generation, the approach reconciles models by chaining them and by letting one model refine the suggestions of previous models. The main benefits of this multi-model approach are that TreeFuzz considers different aspects of the input format and that extending TreeFuzz with additional models is straightforward.

- *Expressiveness*. The models of TreeFuzz can express constraints of input formats that cannot be expressed in a context-free grammar. For example, models can express non-local properties, such as definition-use-like relationships between subtrees of a generated tree. We find such non-local properties to be crucial

for getting past basic checks in the program under test, enabling fuzz-generated inputs to reach deeply into the program.

- *Scalable*. The models supported by TreeFuzz are "single-traversal models", i.e., they are extracted during a single traversal of each tree, and they generate new trees in a single pass. The advantage of such models is that they bound the time of learning and generation, leading to linear time complexity w.r.t. the number of examples to learn from and w.r.t. the number of newly generated trees.

As two examples of input formats that TreeFuzz is useful for, we apply the approach to a programming language, JavaScript, and to a markup language, HTML. Given examples of data in these formats, the approach generates new data with similar properties as the provided examples. As an application of TreeFuzz-generated data, we use generated JavaScript programs for differential testing of web browsers.

Our evaluation assesses the ability of TreeFuzz to generate valid input data, its performance and scalability, as well as its effectiveness for fuzz testing. The results show that the approach generates input data that mostly complies with the expected input format. Specifically, given a corpus of less than 100 HTML documents, the approach creates HTML documents that have only 2.06 validation errors per generated kilobyte of HTML.[1] Given a corpus of JavaScript programs, 96.3% of the created programs are syntactically valid and 16.1% of them execute without any runtime errors. Experiments with different corpus sizes show that the time required for learning models and generating new data increases linearly with the number of examples to learn from. As a result, TreeFuzz scales well to large sets of examples, such as 100,000 JavaScript files. Finally, we find that TreeFuzz is effective for fuzz testing web browsers. Using the TreeFuzz-generated JavaScript programs to fuzz test eight versions of two popular browsers has revealed various inconsistencies, including browser bugs, unimplemented language features, and browser-specific behaviors that developers should be aware of.

In summary, this chapter contributes the following:

- A new technique for learning probabilistic, generative models of tree-shaped data.

1 The example documents are not perfect either: they contain 0.59 errors per kilobyte.

- An application of the technique, fuzz testing, that shows the power of inferred probabilistic models beyond already studied applications, such as code completion.
- An extensible framework that easily supports additional models beyond the six models described in this chapter.
- Empirical evidence shows that the approach is effective at generating valid data that is useful for fuzz testing, while scaling to large amounts of data.

2.2 OVERVIEW AND EXAMPLE

TreeFuzz consists of three phases. First, during the *learning* phase, the approach infers from a corpus of examples a set of probabilistic, generative models that encode properties of the input format. Second, during the *generation* phase, TreeFuzz creates new data based on the inferred models. Finally, the generated data serves as input for the *fuzz testing* phase.

As a running example, consider applying TreeFuzz to JavaScript programs and suppose that the corpus of examples consists only of the program in Figure 2.1(a). For the evaluation (Section 2.6), we apply the approach to significantly larger corpuses. TreeFuzz represents data as a tree with labeled nodes and edges. Figure 2.1(b) shows a tree representation of the example program, which is the abstract syntax tree.

2.2.1 *Learning*

The learning phase of TreeFuzz traverses the tree of the example while inferring probabilistic, generative models of the input format. The models capture structural properties of the tree, which represent syntactic and semantic properties of the input format, i.e. for our running example, the JavaScript language. For example, the approach infers that nodes labeled *Program* have outgoing *body* edges and that these edges may lead to nodes labeled *VarDeclaration* and *IfStmt*. Furthermore, the approach infers the probability of particular destination nodes. For example, for nodes labeled *BlockStmt*, an outgoing edge *body* leads to an *ExprStmt* three out of five times. TreeFuzz infers similar properties for the rest of the tree, providing a basic model of the syntactic properties of the target language, similar to

(a) Example data from corpus:

```
var valid = true, val = 0;
if (valid) {
  function foo(num) {
    num = num + 1;
    valid = false;
    return;
  }
  foo(val);
}
```

(b) Tree representation of example data:

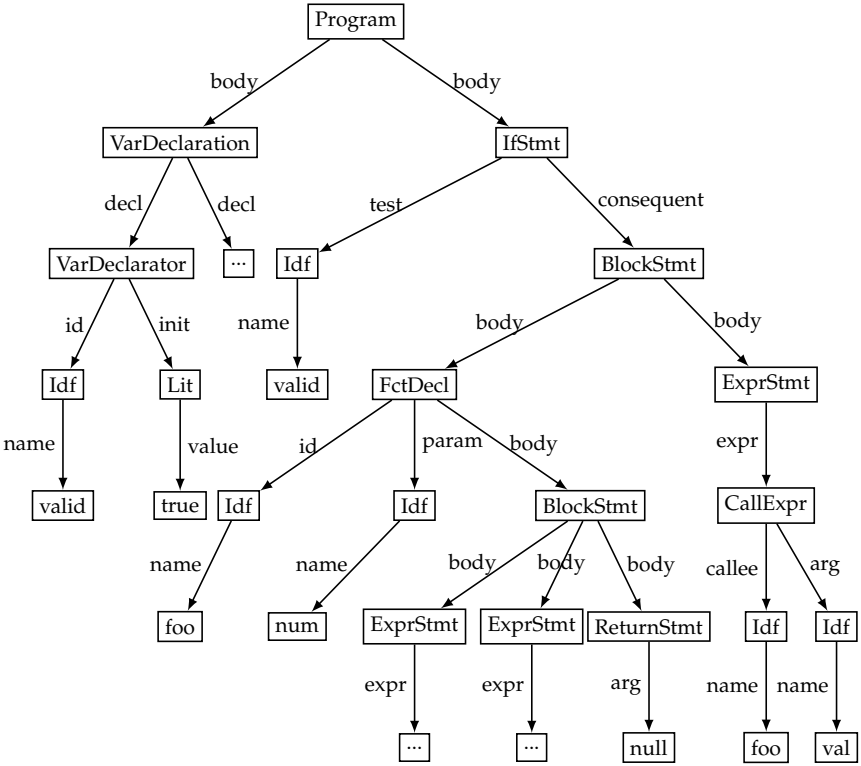


Figure 2.1: Corpus with a single example and its abstract syntax tree. Parts of the abstract syntax tree have been abstracted for the sake of conciseness. *Idf* and *Lit* denote Identifier and Literal, respectively.

a PCFG. Existing format-specific approaches have built-in knowl-

edge of such properties. In contrast, TreeFuzz and existing grammar inference-based approaches learn these properties automatically.

In addition to the PCFG-like properties described above, TreeFuzz infers more complex properties. For example, TreeFuzz considers the ancestors of nodes to find constraints about the context in which a particular node may occur. From the AST in Figure 2.1(b), the approach infers that nodes labeled *ReturnStmt* always occur as descendants of a node *FunctionDecl*, i.e., the approach infers that return statements occur inside functions.

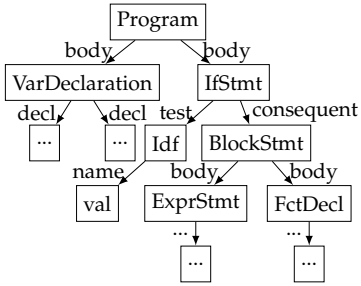
Another inferred property considers repeatedly occurring subtrees. For example, the approach finds that the *id* edge of node *FunctionDecl* and the *callee* edge of node *CallExpr* lead to identical subtrees $Idf \xrightarrow{name} foo$. If such a pattern occurs repeatedly in the corpus, TreeFuzz infers that *FunctionDecl* and *CallExpr* have a definition-use-like relation. Such non-local, semantic properties are crucial to produce input data that are not only syntactically correct but that also reach code deep inside the program under test. For example, we observed that without respecting definition-use-like relations between nodes, most generated JavaScript programs crash after a few lines because of undefined references. TreeFuzz mitigates this problem by learning that uses typically have a matching definition. It is important to note that this and similar properties are inferred without any a priori knowledge about the input format, except for examples of trees in this format.

2.2.2 *Generation*

Based on the inferred models, TreeFuzz creates new trees. Figures 2.2 and 2.3 show four examples of generated trees and their pretty-printed, textual representations as JavaScript programs. The generated trees are based on the example in Figure 2.1. Tree generation starts in a top-down manner and nodes are iteratively expanded guided by the inferred models. For the example, an inferred model specifies that the root node of any tree is labeled *Program*, that *Program* nodes have two outgoing edges, and that the children may be labeled *VarDeclaration* or *IfStmt*. For this reason, all four generated programs contain two statements, which are variable declarations or if statements. Generated programs have the same identifiers and literals as in the corpus because TreeFuzz infers the corresponding nodes.

(a) Generated program 1:

```
var val = true, valid = true;
if (val) {
  foo(val);
  function foo(num) {
    return;
    return;
    val = num + 1;
  }
}
```



(b) Generated program 2:

```
if (valid) {
  function foo(num) {
    return;
    valid = false;
    num = false;
  }
  foo(val);
}
var valid = 0, valid = 0;
```

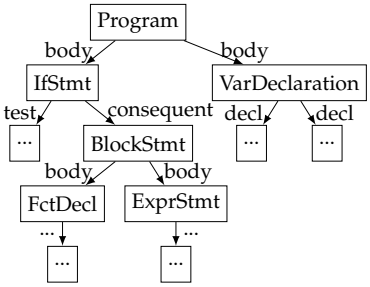
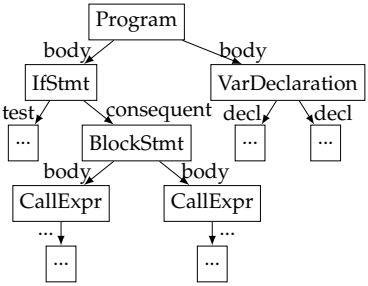


Figure 2.2: Programs generated by TreeFuzz based on the example in Figure 2.1. The left-hand side shows the generated trees; the right-hand side shows the pretty-printed representations of the programs. (Two more examples in Figure 2.3)

To enforce the inferred constraint that return statements must appear within a function declaration, TreeFuzz only creates a *ReturnStmt* node when the currently expanded node is a descendant of a *FunctionDecl* node. As a result, the return statements in the two programs of Figure 2.2 are within a function. Enforcing such constraints avoids syntax errors that TreeFuzz-generated programs would have otherwise. As an illustration of using complex properties encoded in the inferred model, recall the definition-use-like relation between *FuncDecl* and *CallExpr* that TreeFuzz infers. Suppose the approach generates the *Idf* subtree of a *CallExpr* node. To select a label for the destination node of an edge *name*, the approach checks whether there already exists a *FunctionDecl* node with a matching subtree, and if so, reuses the label of this subtree. As a result, most generated function calls in Figure 2.2 have a corresponding function declaration, and vice versa. Creating such relations avoids runtime errors during

(c) Generated program 3:

```
if (valid) {
  foo(val);
  foo(val);
}
var valid = true, val = 0;
```



(d) Generated program 4:

```
var valid = 0, valid = 0;
var valid = true, val = 0;
```

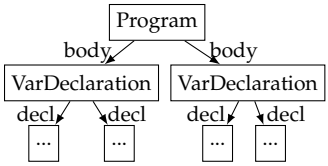


Figure 2.3: Programs generated by TreeFuzz based on the example in Figure 2.1. The left-hand side shows the generated trees; the right-hand side shows the pretty-printed representations of the programs. (Two more examples in Figure 2.2)

fuzz testing, e.g., due to undefined functions, that TreeFuzz-generated programs would have otherwise.

The models that TreeFuzz infers from a single example obviously overfit the example, and consequently, the generated programs do not use all features of the JavaScript language. The hypothesis of this work is that, given a large enough corpus of examples ("big code"), the approach learns a model that is general enough to create a variety of other valid examples that go beyond the corpus.

2.2.3 *Fuzz Testing*

Finally, the data generated by TreeFuzz is given as input to programs under test. For the running example, consider executing the generated programs in multiple browsers to compare their behavior. Executing the program in Figure 2.2(a) exposes an inconsistency between Firefox 45 and Chrome 50. A bug in Firefox[2] causes the program to crash because the function `foo` declared in the `if` block does not get hoisted

2 Mozilla bug #585536

to the top of the block, which leads to a `ReferenceError` when calling it. This bug and other problems detected by TreeFuzz are bugs in the implementation of the semantics of the JavaScript language, i.e., problems that go beyond parser component of the program under test.

2.3 LEARNING AND GENERATION

This section describes the first two phases of TreeFuzz: learning and generation. An important goal of TreeFuzz is to support different kinds of structured data, including programs written in arbitrary programming languages and structured file formats. A common format to represent such data is a labeled, ordered tree, and we use this representation in TreeFuzz.

Definition 2.1. *A* labeled, ordered tree $t = (N, E)$ *consists of a set* N *of nodes and a set* E *of edges. Each node* $n \in N$ *and each edge* $e \in E$ *has a label. The function* $outgoing : N \to E \times ... \times E$ *maps each node to a tuple of outgoing edges. The function* $dest : E \to N$ *maps each edge to its destination node.*

For example, a labeled, ordered tree can represent the AST of a program, the DOM tree of a web page, a JSON file, an XML file, or a CSS file. Section 2.4 shows how to apply TreeFuzz to some of these formats. In the remainder of the chapter, we simply use the term "tree" instead of "labeled, ordered tree". To ease the presentation, we do not explicitly distinguish between a node and its label, or an edge and its label, if the meaning is clear from the context.

2.3.1 *Extensible Learning and Generation Framework*

To enable learning from a corpus of trees and generating new trees, TreeFuzz provides a generic framework that gets instantiated with an extensible set of techniques to infer probabilistic, generative models. We call these techniques *model extractors*. Each model extractor infers a particular kind of property from the given corpus and uses the inferred model to steer the generation of new trees. We currently have implemented six such model extractors (Section 2.3.2).

2.3.1.1 *Hooks*

The TreeFuzz framework provides a set of hooks for implementing model extractors. The hooks are designed to support single-traversal models, i.e., the hooks are called during a single traversal of each example in the learning phase and during a single pass that creates new data during the generation phase. During the learning phase, TreeFuzz calls two hooks:

- $visitNode(node, context)$, which enables model extractors to visit each node of each tree in the corpus once, and
- $finalizeLearning()$, which enables model extractors to summarize knowledge extracted while visiting nodes.

During the generation phase, TreeFuzz calls four hooks:

- $startTree()$, which notifies model extractors that a new tree is going to be generated, enabling them to reset any tree-specific state,
- $pickNodeLabel(node, context, candidates)$, which asks model extractors to recommend a label for a newly created node,
- $pickEdgeLabel(node, context, candidates)$, which asks model extractors to recommend a label for the edge that is going to be generated next, and
- $havePickedNodeLabel(node, context)$, which notifies model extractors that a particular node label has been selected.

The *context* is the path of nodes and edges that lead from the tree's root node to the current node.

One important insight of this chapter is that this simple API is sufficient to infer probabilistic models that enable generating trees suitable for effective fuzz testing.

2.3.1.2 *Learning*

To infer probabilistic, generative models that describe properties of the given set of trees, TreeFuzz traverses all trees while calling the hooks implemented by the model extractors. Algorithm 2.1 summarizes the learning phase. The algorithm traverses each tree in a top-down, depth-first manner and calls the *visitNode* hook for each

Algorithm 2.1 Learning phase.

Input: Set $\mathcal{T}$ of trees.
Output: Probabilistic, generative models.

```
1:  for each t ∈ T do
2:      n ← root(t)
3:      c ← initialize context with n
4:      visitNode(n, c)
5:      while c is not empty do
6:          if visited all e ∈ outgoing(n) then
7:              remove n from c
8:          else
9:              e ← next not yet visited edge ∈ outgoing(n)
10:             n ← dest(e)
11:             expand c with e and n
12:             visitNode(n, c)
13: finalizeLearning()
```

node. During the traversal, the algorithm maintains the context of the currently visited node. After visiting all trees, the algorithm calls *finalizeLearning* to let model extractors summarize and store their extracted knowledge. Section 2.3.2 describes the model extractors in detail.

2.3.1.3 *Generation*

Based on the inferred models, which probabilistically describe properties of the trees in the corpus, TreeFuzz generates new trees that comply with these inferred properties. Algorithm 2.2 summarizes the generation phase of TreeFuzz. Trees are created in a top-down, depth-first manner while querying models about the labels a node should have, how many outgoing edges a node should have, and how to label these edges. The algorithm maintains a work list of nodes that need to be expanded. For each such node, the algorithm calls the *pickNodeLabel* function of all models and repeatedly calls the *pickEdgeLabel* function to determine the outgoing edges of the node. For each newly created outgoing edge, the algorithm creates an empty destination node and adds it to the work list. The algorithm has completed a tree when the work list becomes empty. Once a tree is completed, the algorithm adds it to the set $\mathcal{G}$ of generated trees.

Algorithm 2.2 Generation phase.

Input: Probabilistic, generative models.
Output: Set $\mathcal{G}$ of generated trees.

```
1:  while |G| < maxTrees do
2:      startTree()
3:      n_root ← new node
4:      c ← initialize context with n_root
5:      N ← empty stack                    ▷ work list of nodes to expand
6:      N.push([n_root, c])
7:      while |N| > 0 do
8:          [n, c] ← N.pop()
9:          pickNodeLabel(n, c)
10:         l_e ← pickEdgeLabel(n, c)
11:         while l_e ≠ undefined do
12:             add new edge with label l_e to outgoing(n)
13:             l_e ← pickEdgeLabel(n, c)
14:         for each e ∈ outgoing(n) do
15:             n_dest ← new node
16:             dest(e) ← n_dest
17:             c_dest ← expand c with e and n_dest
18:             insert [n_dest, c_dest] into N
19:         if |reachableNodes(n_root)| > θ then
20:             discard tree and continue with main loop
21:     G ← G ∪ {n_root}
```

Because models may continuously recommend to create additional outgoing edges, generating a tree may not terminate. To address this problem and to bound the size of generated trees, the algorithm checks (line 19) whether the current tree's total number of nodes exceeds a configurable threshold θ (default: 1,000) and discards the tree in this case.

The approach described so far provides a generic framework for inferring properties from a corpus of trees and for generating new trees based on these properties. The following section fills this generic framework with life by presenting a set of model extractors that are applicable across different kinds of data formats, such as JavaScript programs and HTML documents.

2.3.2 *Model Extractors*

To support a wide range of properties of data formats, TreeFuzz uses an extensible set of model extractors. Each model extractor implements the hooks from Section 2.3.1.1 to learn a model from the corpus and to make recommendations for generating new trees. The following explains six model extractors. They are sorted roughly by increasing conceptual complexity, starting from simple model extractors that learn PCFG-like properties and ending with model extractors that encode properties out of reach for PCFGs. Section 2.3.3 explains how TreeFuzz reconciles the recommendations made by different model extractors.

2.3.2.1 *Parent-based Selection of Child Nodes*

Each generated node needs a label. During learning, the following model extractor reads the incoming edge and the parent node from the context provided to *pickNodeLabel* and keeps track of how often a node n is observed for a particular edge-parent pair. This information is then summarized using the $finalizeLearning$ hook into a map $\mathcal{M}_{child}$ that assigns a probability mass function f_{child} to each edge-parent pair.

For the example in Figure 2.1(b), the model extractor infers the following probability mass function for the edge-parent pair $(body, BlockStmt)$:

$$f_{child}(n) = \begin{cases} 0.6 & \text{if } n = ExprStmt \\ 0.2 & \text{if } n = FunctionDecl \\ 0.2 & \text{if } n = ReturnStmt \\ 0 & \text{otherwise} \end{cases}$$

During generation, the approach uses the inferred probabilities to suggest a label for a node based on the incoming edge and the parent of the node. For this purpose, the approach picks a node label according to the probability distribution described by f_{child}.

2.3.2.2 *Determining Outgoing Edges*

The following model extractor infers the set of edges that a particular node label n should have, and uses this knowledge to suggest edge

labels during the generation of trees. To this end, the approach maintains two maps. The map $\mathcal{M}_{edgeExists}$ assigns to an edge label e the probability that n has at least one outgoing edge e. The map $\mathcal{M}_{edgeNb}$ assigns to an edge label e a probability mass function that describes how many outgoing edges e the node n typically has.

LEARNING To construct these two maps, the model extractor implements the *visitNode* hook and stores, for each visited node, the label of the node and the label of its outgoing edges, as well as how many outgoing edges with a particular label the node has. After all trees have been visited, the model extractor uses the $finalizeLearning$ hook to summarize the extracted facts into the maps $\mathcal{M}_{edgeExists}$ and $\mathcal{M}_{edgeNb}$.

For the example in Figure 2.1(b), the model extractor infers the following maps for node *BlockStmt*:

- $\mathcal{M}_{edgeExists} = \{body \mapsto 1.0\}$ because each *BlockStmt* has at least one outgoing edge labeled "body".
- $\mathcal{M}_{edgeNb}$ maps *body* to the following probability mass function because 50% of all block statements have two outgoing *body* edges and the other 50% have three outgoing *body* edges:

$$f_{edgeNb}(k) = \begin{cases} 0.5 & \text{for } k = 2 \\ 0.5 & \text{for } k = 3 \\ 0 & \text{otherwise} \end{cases}$$

GENERATION The inferred maps $\mathcal{M}_{edgeExists}$ and $\mathcal{M}_{edgeNb}$ are used by the *pickEdgeLabel* hook to steer the generation of edges. At the first invocation of *pickEdgeLabel* for a particular node, a list of outgoing edges are pre-computed based on the probabilities stored in these maps. At the first and all subsequent invocations of *pickEdgeLabel* for a particular node, the model returns edge labels from this pre-computed list until each such label has been returned once. Afterwards, the model returns $undefined$ to indicate that no more edges should be created.

For the running example, suppose that the generation algorithm has created a node *BlockStmt*. When it calls *pickEdgeLabel*, the model will decide based on $\mathcal{M}_{edgeExists}$ that there needs to be at least one *body* edge. Furthermore, suppose that the model decides based on

$\mathcal{M}_{edgeNb}$ to create two such edges. As a result, it will return *body* for the first two invocations of *pickEdgeLabel* and *undefined* afterwards.

2.3.2.3 *Determining the Root Node*

Every generated tree needs a root node. This model extractor infers from the corpus which label root nodes typically have. During learning, the extractor builds a map $\mathcal{M}_{root}$ that assigns a label to the number of occurrences of the label in a root node. During generation, the model is used to recommend a label for the root node: When Algorithm 2.2 calls *pickNodeLabel* with a context that only contains the current node (i.e., a root node), the model picks a label from the domain $dom(\mathcal{M}_{root})$ of the map, where the probability to pick a particular label n is proportional to $\mathcal{M}_{root}(n)$.

For the example in Figure 2.1(b), $\mathcal{M}_{root} = \{Program \mapsto 1\}$. When generating a new tree, the approach will recommend the label *Program* for every root node.

The properties learned by the previous three model extractors are similar to those encoded in a PCFG. Existing format-specific approaches hard code the knowledge that these model extractors infers. For example, the grammar used by Csmith [81] encodes which outgoing edges a particular kind of node may have, as well as a set of manually tuned probabilities that specify how many statements a typical function body has, how many arguments a typical function call passes, and what kinds of statements typically occur within a block statement. Instead of hard-coding this knowledge for a specific input format, TreeFuzz infers this knowledge from a corpus.

2.3.2.4 *Ancestor-based Selection of Child Nodes*

Section 2.3.2.1 that infers a node label based on the immediate ancestor of the default indicator for which node to context. The model extractor in Section 2.3.2.1 infers the probability of a node label based on the immediate ancestor of the current node. While the immediate ancestor is a good default indicator for which node to create next, it may not provide enough context. For example, consider determining the destination node of the *value* edge of a *Lit* node. Based on the parent only, the generator would choose among all literals observed in the corpus, ignoring the context in which a literal has been observed, such as whether it is part of a logical expression or an arithmetic expression.

To exploit such knowledge, we generalize the idea presented in Section 2.3.2.1 of increasing the amount of context to consider the k closest ancestor nodes and their connecting edges. We call the sequence of labels of these edges and nodes the *ancestor sequence*. For each such ancestor sequence, the model extractor infers a probability mass function, as described in Section 2.3.2.1, and uses this function during generation to suggest labels for newly created nodes. In addition to the model extractor from Section 2.3.2.1, which is equivalent to $k = 1$, we also use a model extractor that considers the parent and grand-parent of the current node, i.e., $k = 2$. Supporting larger values of k is straightforward, but we have not found any need for a value of $k > 2$.

Since a grammar only encodes the immediate context of each node, existing grammar-based approaches cannot express such ancestor-based constraints. To avoid creating syntactically incorrect programs, the existing Csmith approach [81] uses built-in filters that encode syntactic constraints not obvious from a grammar. Instead, TreeFuzz infers these constraints from a corpus of examples.

2.3.2.5 *Constraints on the Selection of Child Nodes*

Tree structures often impose constraints on where in a tree a particular node may appear. For example, consider an AST node that represents a return statement. In the AST of a syntactically valid program, such a node appears only as a descendant of a node that represents a function. Enforcing such constraints while generating trees is challenging yet important to reduce the probability to generate invalid trees.

To address this challenge, this model extractor infers constraints of the following form:

Definition 2.2. *An* ancestor constraint $(n, \mathcal{N})$ *states that a node labeled* n *must have at least one ancestor from set* $\mathcal{N} = \{n_{A1}, \ldots, n_{Ak}\}$.

Ancestor constraints are inferred in two steps. First, in the *visitNode* hook, the approach stores for each node the set of labels of all ancestors of the node, as provided by the node's context. Second, in the *finalizeLearning* hook, the approach iterates over all observed node labels and checks for each node label n whether all occurrences of n have at least one ancestor from a set $\mathcal{N}$ of node labels. If such a set $\mathcal{N}$ exists, then the approach infers a corresponding ancestor constraint. Otherwise, the approach adds n to the set $\mathcal{N}_{unconstr}$ of unconstrained node labels.

During generation, the approach uses the *pickNodeLabel* hook to suggest a set of nodes that are valid in the current context. This set is the union of two sets. First, the set of all unconstrained nodes $\mathcal{N}_{unconstr}$, because these nodes are always valid. Second, the set of all nodes n that have an ancestor constraint $(n, \mathcal{N})$ where $\mathcal{N}$ has a non-empty intersection with the set of node labels in the current context.

2.3.2.6 *Enforcing Repeated Subtrees (Definition-use-like Relationships)*

Complex trees sometimes contain repeated subtrees that refer to the same concept. For example, consider an AST that contains a function call and its matching function declaration, such as the two subtrees ending with *foo* in Figure 2.1(b). The *Idf* nodes of the call and the declaration have an identical subtree that specifies the name of the function. Such definition-use-like relationships between nodes in the tree occur in various input formats. We find that enforcing such relationships is important to create inputs that reach deep into the program under test. For the example of JavaScript, programs that use undefined program elements often crash immediately and therefore cannot test behavior triggered by long-running programs, e.g., code related to just-in-time optimization.

The following model extractor infers rules that specify which nodes of a tree are likely to share an identical subtree.

Definition 2.3. *An* identical subtree rule *states that if there exists a subtree* $n_A \xrightarrow{e_A} n_B \xrightarrow{e_b} x$ *in the tree, then there also exists a subtree* $n_D \xrightarrow{e_d} n_B \xrightarrow{e_b} y$ *in the same tree so that* $x = y$.

The notation $n \xrightarrow{e} n'$ denotes that a node labeled n has an outgoing edge labeled e whose destination is a node labeled n'. For each rule, the approach infers the support, i.e., how many instances of this rule have been observed, and the confidence, i.e., how likely the right-hand side of the rule holds given that the left-hand side of the rule holds.

For example, given the corpus of JavaScript programs that we use in the evaluation, TreeFuzz infers that

$$CallExpr \xrightarrow{callee} Idf \xrightarrow{name} x$$

implies

$$FunctionDecl \xrightarrow{id} Idf \xrightarrow{name} y$$

so that $x = y$ with support 59,146 and confidence 61.7%. This rule expresses that function calls are likely to have a corresponding function declaration with the same function name. The reasons for confidence being lower than 100% are that functions can also be declared through a function expression and that functions may be defined in other files.

LEARNING To infer identical subtree rules, the model extractors uses the *visitNode* and *finalizeLearning* hooks. When visiting a node n with context ... $\rightarrow n_A \xrightarrow{e_A} n_B \xrightarrow{e_B} n$, the approach stores the information that the suffix $n_B \xrightarrow{e_B} n$ has been observed with the prefix ... $\rightarrow n_A \xrightarrow{e_A}$. After visiting all trees, the *finalizeLearning* hook summarizes the stored information into identical subtree rules by considering all suffixes that have been observed with more than one prefix. The approach increments the support of a rule for each node n for which the rule holds. To compute the confidence of a rule, the approach divides rule's support by the number of times the left-hand side of the rule has been observed.

GENERATION During generation, the approach uses the inferred identical subtree rules to suggest labels for nodes that are at positions x and y (as in Definition 2.3) of a rule. To this end, the approach maintains two maps. First, the map $\mathcal{M}_{pathToLabels}$ associates to a subtree $n_D \xrightarrow{e_D} n_B \xrightarrow{e_B}$ the set of labels x that have already been used to label the destination node of e_B. Second, the map $\mathcal{M}_{pathToLabelTodos}$ associates with a subtree $n_D \xrightarrow{e_D} n_B \xrightarrow{e_B}$ the set of labels that the generator still needs to assign to a destination node of e_B to comply with a identical subtree rule. Whenever the *havePickedNodeLabel* hook is called, the approach checks if the current context matches any of the inferred rules. If the current node matches the left-hand side of a rule, then the approach decides with a probability equal to the rule's confidence that the right-hand side of the rule should also be true. If $\mathcal{M}_{pathToLabels}$ indicates that the right-hand side is not yet fulfilled, then the approach adds an entry to $\mathcal{M}_{pathToLabelTodos}$. Whenever the *pickNodelLabel* hook is called, the approach checks whether the current context matches an entry in $\mathcal{M}_{pathToLabelTodos}$. If it does, the approach fulfills the rule by suggesting the required label.

The last three model extractors show that single-traversal models can express rather complex rules and constraints that go beyond grammar-based approaches. Existing format-specific approaches, such as Csmith [81], hard code such constraints into their approach. For example, if Csmith generates a function call, it checks whether there is any matching function definition, and otherwise, generates such a function definition. Existing grammar-inference-based approaches, such as Glade [180], ignore properties of the input format that go beyond the expressiveness of grammars, and as a result, are most useful for testing the input parsing component of a program under test.

TreeFuzz provides a general framework that allows for implementing a wide range of models beyond the six that we describe here.

2.3.3 *Combining Multiple Model Extractors*

When Algorithms 2.1 and 2.2 call a hook function, they call the function for each available model extractor. In particular, this means that multiple model extractors may propose different labels during the generation of trees. For example, suppose that while generating a tree, the generation algorithm must decide on the label of a newly created node. One model extractor, e.g., the one from Section 2.3.2.5, may restrict the set of available node labels to a subset of all nodes, and another model extractor, e.g., the one from Section 2.3.2.1 may pick one of the labels in the subset. Furthermore, when multiple model extractors provide contradicting suggestions, then the generation algorithm must decide on a single label.

To reconcile the suggestions by different model extractors, TreeFuzz relies on an order of precedence for querying the model extractors during generation. Based on such an order, each model extractor obtains the set of label candidates from the already queried extractors and returns another set of candidates. The set of candidates returned by a model extractor must be a subset of its input set, i.e., a model extractor can only select from the set of already pre-selected candidates. If, after querying all model extractors, the set of label candidates is non-empty, the generator randomly picks one of the candidates. If the set of candidates is empty, the generator falls back on a random default strategy, which sets node labels to the empty string and suggests to create another edge with an empty label with a configurable probability (default: 10%). During our evaluation, when using all

model extractors described in this section, the set of candidates is practically never empty.

For the evaluation, we use the model extractors in the following order of precedence (high to low): Constraints on the selection child nodes, determining the root node, enforcing repeated subtrees, determining outgoing edges, ancestor-based selection of child nodes, and parent-based selection of child nodes.

2.4 FUZZ TESTING

This section presents how to use TreeFuzz-generated data as inputs for fuzz testing. We consider two data formats: programs in the JavaScript programming language (Section 2.4.1) and documents in the web markup language HTML (Section 2.4.2).

2.4.1 *JavaScript Programs*

TreeFuzz generates ASTs of JavaScript programs by learning from the ASTs of a set of example programs. Generated programs may serve as test input for program analyses, refactoring tools, compilers, and other tools that process programs [81, 84]. We here use TreeFuzz-generated JavaScript programs for differential testing across multiple browsers, where the same program is executed in multiple browsers to detect inconsistencies among the browsers.

Our differential testing technique classifies programs into three categories: First, the behavior is *consistent* if there is no observable difference across browsers, which may be because the program either crashes in all browsers or does not crash all browsers. Second, the behavior is *inconsistent* if we observe a difference across browsers. This may be either because the program raises an exception in at least one browser but does not crash in another browser, or because the program crashes in all browsers but with different types of error, such as *TypeError* and *ReferenceError*. To compare errors with each other, we use the type of the thrown runtime error, as specified in the language specification. Finally, some programs are classified as *non-deterministic* because the behavior of different executions in a single browser differs, which we check by executing each program twice.

2.4.2 *HTML Documents*

As another input format, we apply TreeFuzz to the hypertext markup language HTML. Due to the popularity of HTML documents, there are various tools that require HTML documents as their input, such as browsers, text editors, and HTML processing tools. TreeFuzz generates inputs for these tools based on a corpus of example HTML documents, without requiring any explicitly given knowledge about the structure and content of HTML documents.

Since an HTML document consists of nested tags, there is a natural translation from such documents to labeled, ordered trees. We represent each tag as a node, where the label represents the tag name, such as *body* and *a*. We represent nested tags through an edge between the parent and the child. The label of this edge is *childNode* concatenated with the label of the destination node. The reason for copying the destination's label into the edge label is that otherwise, most edges would have the generic label *childNode*, which is not helpful in inferring the tree's structure. We represent attributes of tags, such as `id='foo'`, through child nodes with label *attribute*. These nodes have two outgoing edges, which point to the name and the value of the attribute, e.g., *id* and *foo*.

2.5 IMPLEMENTATION

We implement the approach into a framework with an extensible set of model extractors. The implementation can be easily instantiated for different input formats because most of the implementation of the framework and the model extractors is independent of the format. The JavaScript instantiation builds upon an existing parser [191] and code generator [274] and adds less than 300 lines of JavaScript code to the framework. The HTML instantiation builds upon an existing toolkit to parse and generate HTML documents [282] and adds less than 200 lines of JavaScript code to the framework. We implement differential testing as an HTTP server that sends JavaScript programs to client code running in different browsers, and that receives a summary of the programs' runtime behavior from these clients.

2.6 EVALUATION

This section describes an experimental evaluation of TreeFuzz.

		minimum	median	maximum
HTML	file size (bytes)	39	77,604	703,327
	number of nodes	11	4,858	41,626
JS	file size (bytes)	3	2,438	7,241,063
	number of nodes	0	262	1,045,978

Table 2.1: HTML and JS corpuses used for learning.

2.6.1 *Experimental Setup*

CORPUS We use a corpus of 100,000 JavaScript files collected from GitHub [278]. For HTML, we visit the top 100 web sites (according to the Alexa ranking) and store the HTML files of their start page. Some sites appear multiple times in the top 100 list, e.g., `google.com` and `google.co.in`. We remove all but one instance of such duplicates and obtain a corpus of 79 unique HTML files. Table 2.1 summarizes the file size and the number of nodes in the tree representations of the corpuses.

DIFFERENTIAL CROSS-BROWSER TESTING We instantiate the differential testing technique (Section 2.4.1) with eight versions of the popular Firefox and Chrome browsers released over a period of four years: Firefox 17, 25.0.1, 33.1, 44, and Chrome 23, 31, 39, and 48.

All performance-related experiments are carried out on an Intel Core i7-4790 CPU (3.60GHz) machine with 32GB of memory running Ubuntu 14.04. We use Node.js 6.5 and provide it with 11GB of memory.

2.6.2 *Syntactic Validity of Generated Trees*

TreeFuzz generates trees intended to comply with an input format without any a priori knowledge about this format beyond a set of example trees. To assess the effectiveness in achieving this goal, we measure the percentage of generated trees that pass language-specific checks of syntactic validity.

JAVASCRIPT To measure whether a generated JavaScript program is valid, we pretty print it and parse it again. If the pretty printer rejects the tree or if the parser rejects the generated program, then

we consider the program as syntactically invalid. 96.3% of 100,000 generated trees represent syntactically valid JavaScript programs.

HTML To measure the validity of generated HTML documents, we use the W3C markup validator [281]. In practice, most HTML pages are not fully compatible with W3C standards and therefore cause validation errors. As a measure of how valid an HTML document is, we compute the number of errors per kilobyte of HTML.

The generated HTML documents have 2.06 validation errors per kilobyte. As a point of reference, the corpus documents contain 0.59 validation errors per kilobyte. That is, the generated documents have a slightly higher number of errors, but overall, represent mostly valid HTML. We conclude that TreeFuzz effectively generates HTML documents that mostly comply with W3C standards, without any a priori knowledge of HTML.

To the best of our knowledge, there is no existing approach based on a learned, probabilistic language model that generates entire programs with so few mistakes.

2.6.3 *Semantic Validity of Generated Trees*

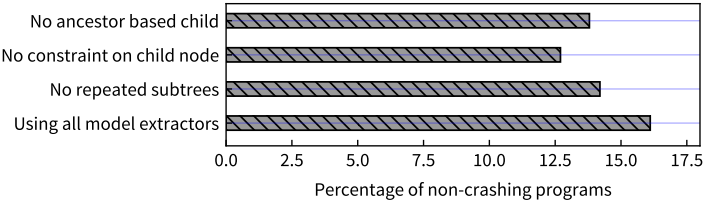


Figure 2.4: Percentage of non-crashing JavaScript programs generated by variants of the TreeFuzz approach.

Some of the model extractors of TreeFuzz, in particular those presented in Sections 2.3.2.4 to 2.3.2.6, address properties of the input format that go beyond the expressiveness of context-free grammars. One motivation for including such models is to prevent generated data to be quickly discarded by the program under test, so that the testing can reach deeply into the program. For generated JavaScript program, we observed that a common reason for quickly discarding

programs is that the program crashes due to a violation of some semantic property that human-written programs typically respect. For example, many programs crash because of undefined references, a problem addressed by the model extractor in Section 2.3.2.6.

To validate whether adding these model extractors leads to generated input data with fewer semantic errors, we compare different variants of TreeFuzz. Specifically, we compare the approach without the model extractors in Sections 2.3.2.4 to 2.3.2.6 with the full approach that includes all model extractors. With each variant, we generate 1,000 JavaScript programs, execute these programs, and count how many programs cause a runtime crash, i.e., how many programs do not terminate normally.

Figure 2.4 shows the results of the comparison. With the full TreeFuzz approach, 16.1% of all generated programs terminate without crashing. In contrast, the variants of TreeFuzz that leave out one of the model extractors all lead to a lower percentages of non-crashing programs. We draw two conclusion from these results. First, adding model extractors that go beyond context-free grammars helps to enforce semantic properties of the input format. Second, there is potential for further model extractors that address additional semantic properties. Since TreeFuzz is designed as an extensible framework, adding more model extractors is straightforward.

2.6.4 *Influence of Corpus Size on Validity and Performance*

INFLUENCE OF CORPUS SIZE ON VALIDITY To be effective, statistical learning approaches often need large amounts of training data. We evaluate the influence of the corpus size on the validity of TreeFuzz-generated programs. We measure the percentage of syntactically correct generated JavaScript programs while learning from a varying corpus size ranging from 10 to 100,000. We observe that the percentages vary between 96% and 98%, i.e., most generated programs are syntactically correct independent of the corpus size. We conclude from the results that the size of the corpus does not have a significant influence on the validity of the generated trees, suggesting that TreeFuzz is useful even when few examples are available.

PERFORMANCE AND SCALABILITY To enable TreeFuzz to learn from many examples and to generate large amounts of new data, the performance and scalability of the approach is crucial. Figure 2.5

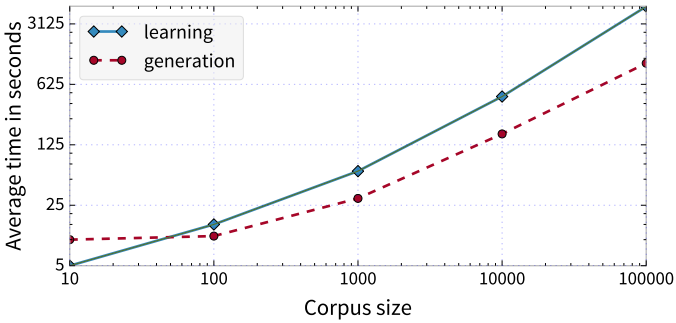


Figure 2.5: Learning and generation time based on varying corpus sizes. Both axes are log-scaled.

shows how long the approach takes to learn depending on the size of the corpus, and how long it takes to generate 100 trees. The presented results are averages over three repetitions to account for performance variations. We observe that both learning and generation scale linearly with the size of the corpus. The main reason for obtaining linear scalability is that the approach focuses on single-traversal models.

2.6.5 *Effectiveness for Differential Testing*

As an application of TreeFuzz-generated JavaScript programs, we evaluate the effectiveness for differential testing (Section 2.4.1) in two ways. First, we quantitatively assess to what extent the generated trees reveal inconsistencies. Second, we present a set of inconsistencies that we discovered during our experiments and discuss some of them in detail.

QUANTITATIVE EVALUATION OF DIFFERENTIAL TESTING The behavior of most programs (97.2%) is consistent across all engines unsurprisingly because consistency is the intended behavior. For 0.22% of all programs, the behavior is non-deterministic, i.e., two executions in the same browser have different behaviors. Each of the remaining 2.5% of programs expose an inconsistency, i.e., achieves the ultimate goal of differential testing. Given the little time required to generate programs (Section 2.6.4), we conclude that TreeFuzz is effective at generating programs suitable for differential testing.

ID	Inconsistent browsers	Description	Root cause
1	Firefox vs. Chrome	Mozilla bug #585536: Function declared in block statement should get hoisted to top of block.	Browser bug
2	Firefox 17 and 25 vs. others	Mozilla bug #597887: Calling `setTimeout` with an illegal argument causes runtime error.	Browser bug
3	Firefox 44 vs. others	Mozilla bug #1231139: `TypeError` is thrown even though it should be `SyntaxError`.	Browser bug
4	Firefox 17 and 25 vs. others	Mozilla bug #409444: The type of `window.constructor` is "object" in some browsers and "function" in others.	Browser bug
5	Firefox vs. Chrome	Only Firefox provides `window.content` property.	Browser-specific behavior
6	Firefox 44, Chrome 23, and Chrome 31 vs. others	Some browsers throw an exception when calling `scrollBy` without arguments.	Browser-specific behavior
7	Firefox vs. Chrome	`event` is a global variable in Chrome but not in Firefox.	Browser-specific behavior
8	Chrome 23 vs. others	Some browsers throw an exception when calling `setTimeout` without arguments.	Browser-specific behavior
9	Firefox 25–44 vs. others	Some browsers throw an exception when redirecting to a malformed URI.	Browser-specific behavior
10	Firefox 17–33 vs. others	Call of `Int8Array()` without mandatory `new` keyword, as required by ECMAScript 6.	Evolving specification

Table 2.2: Examples of inconsistencies found through differential testing with TreeFuzz-generated programs.

QUALITATIVE EVALUATION OF DIFFERENTIAL TESTING To better understand the detected inconsistencies, we manually inspect a subset of all inconsistencies. Table 2.2 lists ten representative inconsistencies and associates them with three kinds of root causes. First, *browser bugs* are inconsistencies caused by a particular browser that does not implement the specified behavior. Second, *browser-specific behavior* are inconsistencies due to unspecified or non-standard features that some but not all browsers provide, or because the standards allow multiple different behaviors. Third, *evolving specification* refers to inconsistencies due to features of not yet implemented revised specifications, such as ECMAScript 6 and DOM4. The examples listed in Table 2.2 show that TreeFuzz-generated JavaScript programs are effective at revealing different kinds of inconsistencies among browsers.

2.6.6 *Comparison with Corpus and Other Approaches*

We compare TreeFuzz to three alternatives approaches:

- A PCFG-based approach that creates JavaScript programs based on a probabilistic grammar of abstract syntax trees [279]. The approach generates programs by starting from the top-level AST node *Program* and by iteratively expanding nodes according to the grammar. If a node has multiple possible expansions, we decide on which expansion to use based on a probability distribution, which we extracted by traversing the AST representation of the 100,000 corpus files introduced in Section 6.1.

  During our experiments, a naive implementation of the PCFG-based approach often fails to terminate after a fixed timeout because expanding a grammar rule often leads to another non-terminal. For example, expanding a block statement leads to a sequence of statements, which may again include block statements, etc. To address this problem, we manually modify the probability distribution of some grammar rules. Specifically, we bias the distribution in favor of nodes that lead to terminal symbols without introducing recursion. For the example of statements, we increase the probability to choose either an identifier or a literal, as these AST nodes do not lead to further statements. With this manual biasing of the inferring probabilities, the PCFG-based approach generates trees in a reasonable

amount of time. We call the approach *PCFG_term*, where "term" stands for "termination".

- *LangFuzz* [84] is a state of the art fuzzing approach. Similar to TreeFuzz, it supports multiple languages and exploits a corpus of examples. In contrast to our work, LangFuzz requires built-in knowledge of the target language, such as which AST nodes represent identifiers and which built-in variables and keywords exist. For example, LangFuzz uses knowledge to adapt program fragments by modifying its identifier names.

  During our experiments, LangFuzz suffered from severe scalability problems when providing it with the full corpus of 100,000 programs. One reason is that the implementation keeps all programs in memory. Because of these problems, we provide only 10,000 programs to LangFuzz. These programs are a randomly sampled subset of all corpus programs.

  The root cause of these memory issues is that LangFuzz combines fragments of existing programs with each other. Our approach avoids such problems by learning a probabilistic model of JavaScript code, instead of storing concrete code fragments.

- The *corpus-only* approach uses the 100,000 corpus programs as an input, i.e., no new programs get generated.

2.6.6.1 *Comparison Based on Syntactical Differences*

At first, we compare the approaches by syntactically comparing the programs that they provide. For this experiment, we format all programs consistently and remove all comments. We compare the programs generated by TreeFuzz and by LangFuzz with the programs in the corpus to assess whether any generated programs are syntactically equal to a corpus program. 241 of the 100,000 programs generated by LangFuzz are such duplicates, whereas only one of 100,000 TreeFuzz-generated programs is also present in the corpus. We conclude that TreeFuzz is effective at creating a large number of syntactically diverse programs.

The effectiveness of generated programs for fuzz testing partly depends on whether the programs are syntactically correct. The reason is that syntactically incorrect programs are typically rejected by an early phase of the JavaScript engine and therefore cannot reach any code beyond that phase. Figure 2.6 shows for each of the four

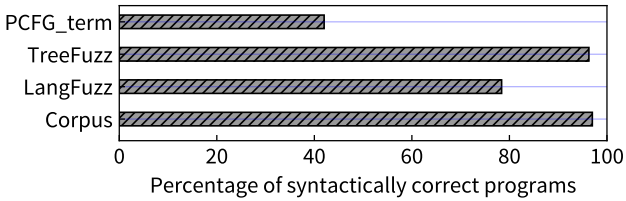


Figure 2.6: TreeFuzz compared to corpus, LangFuzz and PCFG_term.

approaches the percentage of syntactically correct programs among all generated programs. For TreeFuzz and the corpus programs, the percentage is 96.3% and 97.0%, respectively. The fact that both values are similar confirms that TreeFuzz effectively learns from the given corpus. In contrast, only 78.4% of the programs generated by LangFuzz are syntactically correct. The percentage of syntactically correct programs generated by PCFG_term is lowest at 42%.

2.6.6.2 *Comparison Based on Differential Testing*

To compare the programs generated by the different approaches beyond their syntax, we compare what kinds of inconsistencies the programs find when being used for differential testing. Since syntactically incorrect programs are typically rejected by an early phase of the JavaScript engine and therefore cannot reach any code beyond that phase, for this comparison, we use only the best three approaches containing syntactically correct programs i.e., TreeFuzz, LangFuzz and the Corpus.

Since inspecting thousands of inconsistencies manually is practically infeasible, we assign inconsistencies to equivalence classes based on how an inconsistency manifests. These equivalence classes are an approximation of the actual root cause that triggers an inconsistency.

To compute the equivalence class of a program, we summarize the behavior in a particular browser into a single string, such as "okay" for a non-crashing program and "ReferenceError" or "TypeError" for a crashing program. Based on these summaries, we compute a tuple $(b_1, \ldots, b_n)$ of strings for each program, where each b_i is the summary from a particular browser. Two inconsistencies belong to the same equivalence class if and only if they share the same tuple. For example, two programs that both throw a "TypeError" in all

versions of Chrome but do not crash in any version of Firefox belong to the same equivalence class.

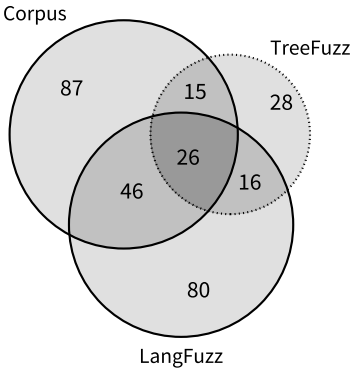


Figure 2.7: Equivalence classes of inconsistencies found by the three approaches.

Figure 2.7 summarizes the results of the comparison. The figure shows for each approach how many equivalence classes of inconsistencies the approach detects, and how many equivalence classes are shared by multiple approaches. The results show that the three approaches are complementary to each other. Even though there is an overlap of 26 equivalence classes found by all three approaches, each individual approach contributes a set of otherwise missed inconsistencies. In particular, TreeFuzz detects 28 otherwise missed classes of inconsistencies.

2.7 CONCLUSION

In this chapter, we present TreeFuzz, a format-independent, blackbox fuzz testing approach that generates tree-structured data. The core idea is to infer from a corpus of example data a set of probabilistic, generative models, which then create new data that have properties similar to the corpus. Beyond the set of example trees, the approach does not require any a priori knowledge of the format of the generated data, but instead infers syntactic and semantic properties of the format. TreeFuzz supports an extensible set of single-pass models, enabling it to learn a wide range of properties of the data format. We

apply the approach to two different data formats, a programming language and a markup language, and show that TreeFuzz generates data that is mostly valid and effective for detecting bugs through fuzz testing.

3

SEMANTIC BUG SEEDING: A LEARNING-BASED APPROACH FOR CREATING REALISTIC BUGS

Chapter 2 presents TreeFuzz, one approach to address the challenge (C-I, Chapter 1) of generating valid and realistic programs. Our focus with TreeFuzz is to generate complete programs from scratch. The current chapter presents an alternative approach to address the same challenge but instead of generating complete programs from scratch, we mutate existing programs. One of the benefits of mutating existing programs is that the mutations can increase the diversity of existing programs, which is useful for training neural models. The most common way to mutate programs is by using pre-defined mutation operations. In this chapter, we present an alternative approach, called SemSeed, where we do not mutate using pre-defined operations but instead learn such operations from a corpus. SemSeed is the first to use learned token embeddings for mutating programs that generates realistic programs.

3.1 MOTIVATION

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 [151, 213, 217], and bugs created via mutations are useful for evaluating the effectiveness of test suites [76]. 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 [118, 230].

Another purpose of known bugs is to help build a bug-related technique. For example, learning-based bug detectors [219, 226, 238], defect prediction models [171], and repair tools [234, 259] 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 [226], on noisy datasets that, e.g., approximate buggy code as any code changed in the next version of a program [171], on manually curated bug datasets [118, 230], or on code changes that are heuristically linked with bug reports [238].

This chapter 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 [182, 271], 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 3.1 summarizes and contrasts SemSeed with other work on automatically seeding bugs. First, mutation testing [102, 117] seeds bugs based on pre-defined code transformations. However, mutation operators cover only a small set of the syntactic transformations

Table 3.1: Comparison with other bug seeding techniques.

Approach	Kinds of bugs	Target locations (C1)	Adaptation to target location (C2)	Unbound tokens (C3)
Mutation operators [102, 117]	Few, manually defined	Everywhere	Syntactic	Not supported
Inferred mutat. operators [183]	Many, inferred	Everywhere	Syntactic	Not supported
Neural machine translation [246]	Many, inferred	Implicit by model	Implicit by model	Not supported
Bug synthesis [151, 227]	Memory updates	Hard to trigger paths	N/A	N/A
This work	Many, inferred	Based on semantic fit	Semantic	Supported

that occur in the wild and only sometimes represent real-world bugs [113]. Second, some work infers mutation operators from past bug fixes [183]. 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 [246]. 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 [151, 227] 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 [102], 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 [226] significantly improves the bug detection ability compared to the state of the art.

In summary, this chapter 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, comple-

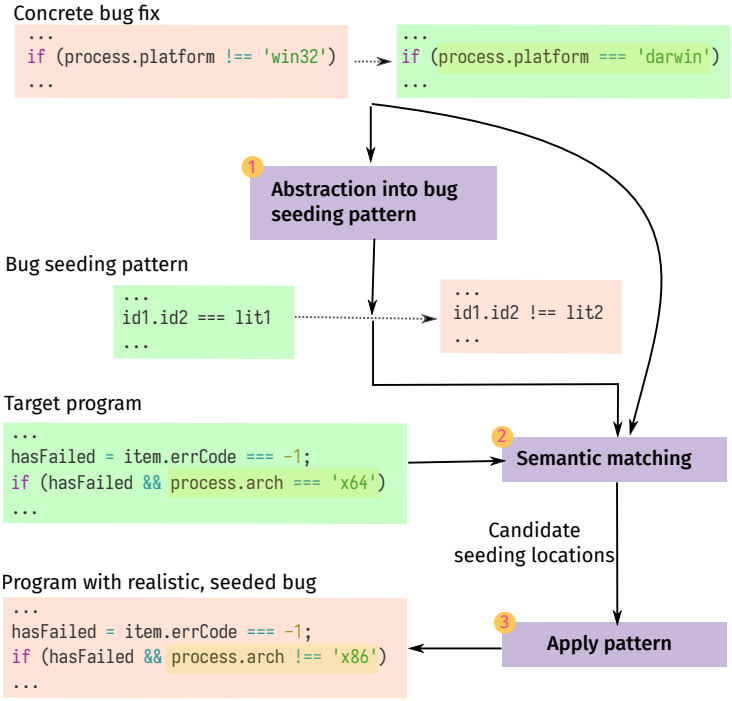


Figure 3.1: Overview of the approach and running example.

ments traditional mutation operators, and yields bugs useful for training more effective bug detection models.

3.2 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 3.1 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 3.1, the "wrong comparison with wrong literal" pattern could be applied at every binary expression that compares some `id1.id2` with some `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 3.1, 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 [195, 210, 226, 240], 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.

3.3 APPROACH

This section presents the three steps of our SemSeed approach in detail. At first, Section 3.3.1 describes how to extract bug seeding patterns from concrete bug fixes in version histories. Then, Section 3.3.2 presents how our approach semantically matches these bug seeding patterns against previously unseen code to find candidate locations for seeding new bugs. Finally, Section 3.3.3 describes how to apply the patterns to a given code location by semantically adapting them.

3.3.1 *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.

3.3.1.1 *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 chapter. 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 [198, 236, 253].

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 [246] of commits mined with a similar approach has shown 97% of their commits to be bug-fixing.

3.3.1.2 *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 3.1. *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 3.1. 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.

3.3.1.3 *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 3.2. *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 [244]. 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 3.1 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.

3.3.2 *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.

3.3.2.1 *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 3.3.1.3 to get the abstracted token sequence C_{abstr}.

For example, consider the target program in Figure 3.1. 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`].

3.3.2.2 *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 [183].

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 3.3.1. Hence, both binary expressions in the target program are retained as candidate seeding locations.

3.3.2.3 *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.

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 [101] or FastText [182]. 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 [182], which we choose for two reasons. First, FastText does not suffer from the out-of-vocabulary problem [256], because it reasons about the n-grams in a token in-

Algorithm 3.1 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

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

stead 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 [271].

Algorithm 3.1 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 [120, 179]. 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 3.1 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'`.

3.3.3 *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.

3.3.3.1 *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 3.1, 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 [183, 246] 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

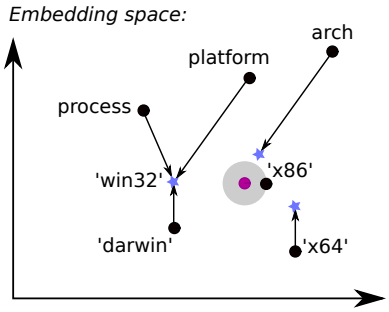


Figure 3.2: Example of analogy queries to bind unbound tokens.

the bug seeding location, e.g., because it uses a variable of a wrong type.

3.3.3.2 *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"* [101]. 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 3.2 illustrates our analogy-based technique for binding unbound tokens using the example in Figure 3.1. 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

Algorithm 3.2 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
    $V_{tgt} \leftarrow \emptyset$ ▷ 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}

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 3.2. The nearest neighbor in our example is the literal token `'x86'`, and hence SemSeed seeds a bug using this token.

3.3.3.3 *Algorithm*

After providing an intuition of the approach, Algorithm 3.2 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 3.3.2, 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 3.2) 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 3.2.

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 3.2, 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.

3.4 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 [182] on token sequences extracted from a corpus of 150K JavaScript [166] files.

3.5 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?

3.5.1 *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 3.3.1.1, 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 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.

3.5.2 *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 [183]. 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 3.2 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.

Table 3.2: 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 3.3: 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

3.5.3 *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. 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 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 3.3 lists some bug seeding patterns that SemSeed finds, along

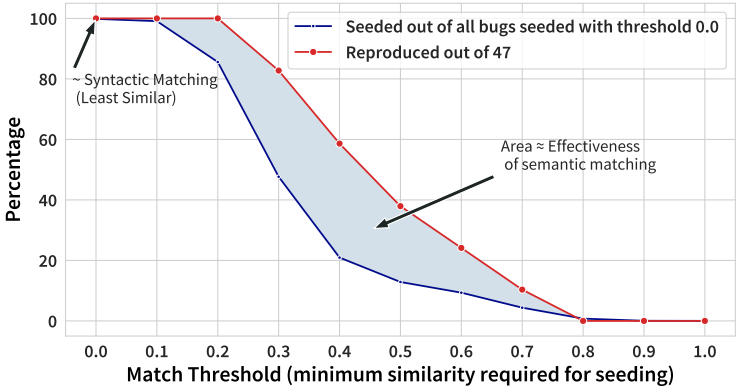


Figure 3.3: Influence of matching threshold m on seeded bugs.

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 [183, 246], 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.

3.5.4 *RQ3: Impact of Configuration Parameters*

3.5.4.1 *Matching Threshold m*

The matching threshold m determines in Algorithm 3.1 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 3.3 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.

3.5.4.2 *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 3.3.3.3). 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.
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 3.2, and we evaluate values of k ranging from 1 to 10.

Figure 3.4 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.

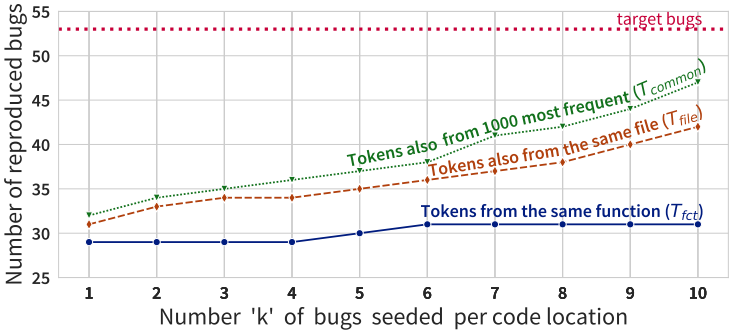


Figure 3.4: Reproduced real-world bugs depending on token set T and number k if bugs to seed.

3.5.5 *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 [226], 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 [226], 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 targeted by DeepBugs. We apply both configurations to the same code corpus:

a de-duplicated version [207] of a JavaScript corpus [166], 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 3.3.1.1; and (iii) Bugs from the JavaScript variant of an existing dataset of single-statement bugs [236]. 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 3.7 and Figure 3.10 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)
```

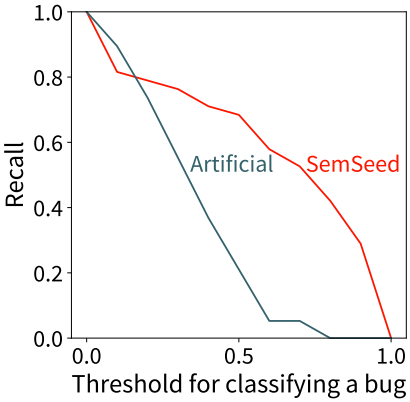


Figure 3.5: Recall wrong assignments.

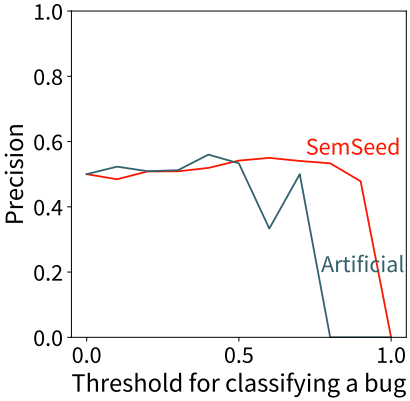


Figure 3.6: Precision wrong assignments.

Figure 3.7: Precision and recall of DeepBugs with artificially seeded and SemSeed-seeded bugs.

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.

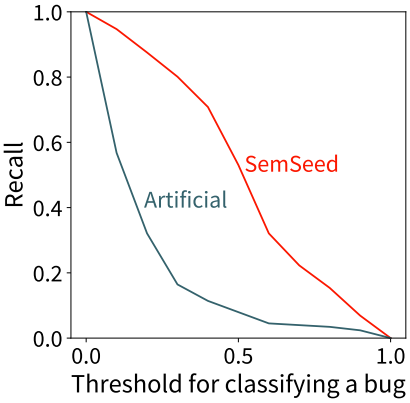


Figure 3.8: Recall wrong binary operand.

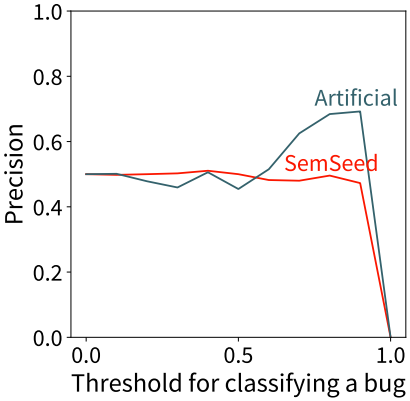


Figure 3.9: Precision wrong binary operand.

Figure 3.10: Precision and recall of DeepBugs with artificially seeded and SemSeed-seeded bugs.

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

3.5.6 *RQ5: Comparison with Traditional Mutation Operators*

Existing code mutation approaches, such as Mutandis [102] for JavaScript and Major [117] 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.[1]

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.

3.5.7 *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 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

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

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.

3.6 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 [198, 236, 253]. For example, Karampatsis and Sutton [236] 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 code 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.

3.7 CONCLUSION

This chapter presents SemSeed, an approach for seeding bugs in a semantics-aware way. Given a possibly small set of example bugs obtained from a corpus, 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.

4

NL2TYPE: INFERRING JAVASCRIPT FUNCTION TYPES FROM NATURAL LANGUAGE INFORMATION

In the previous two chapters (Chapter 2, Chapter 3), we use static analysis to either generate programs or mutate existing programs. In both cases, the static analysis ignores code constructs such as comments, thereby missing some rich source of information for reasoning about programs. This chapter presents NL2Type where we perform corpus-based static analysis to train a neural model that is able uncover inconsistencies in source code. One of the challenges of reasoning about programs written in dynamically typed languages (C-II, Chapter 1) is the lack of statically declared types. The current chapter addresses this challenge by leveraging natural language information such as comments present in the source code.

4.1 MOTIVATION

JavaScript has become one of the most popular programming languages. It is widely used not only for client-side web applications but, e.g., also for server-side applications running on Node.js [232], desktop applications running on Electron, and mobile applications running in a web view. However, unlike many other popular languages, such as Java and C++, JavaScript is dynamically typed and does not require developers to specify types in their code.

While the lack of type annotations allows for fast prototyping, it has significant drawbacks once a project grows and matures. One drawback is that modern IDEs for other languages heavily rely on types to make helpful suggestions for completing partial code. For example, when accessing the field of an object in a Java IDE, code completion suggests suitable field names based on the object's type.

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangle.
  * @returns {number} The area of the rectangle in meters.
  *                   May also be used for squares.
  */
getArea: function(length, breadth) {
  return length * breadth;
}
```

Figure 4.1: Function with JSDoc annotations. The annotations include comments, parameter types, and the return type.

In contrast, JavaScript IDEs often fail to make accurate suggestions because the types of the code elements are unknown. Another drawback is that APIs become unnecessarily hard to understand, sometimes forcing developers to guess what types of values a function expects or returns. Finally, type errors that would be detected at compile time in other languages may remain unnoticed in JavaScript, which causes unexpected runtime behavior.

To mitigate the lack of types in JavaScript, several solutions have been proposed. In particular, gradual type systems, such as Flow [275] developed by Facebook and TypeScript [280] developed by Microsoft, use a combination of developer-provided type annotations and type inference to statically detect type errors. A popular format to express types in JavaScript are JSDoc annotations. Figure 4.1 shows an example of such annotations for a simple JavaScript function. The main bottleneck of these existing solutions is that they rely on developers to provide type annotations, which remains a manual and time-consuming [222] task.

Previous work has addressed the type inference problem through static analyses of code [49, 53, 69, 98]. Unfortunately, the highly dynamic nature of languages like JavaScript prevent these approaches from being accurate enough in practice. In particular, analyses that aim for sound type inference yield various spurious warnings.

This chapter addresses the type inference problem from a new angle by exploiting a valuable source of knowledge that is often overlooked by program analyses: the natural language information embedded in source code. We present NL2Type, a learning-based approach that uses the names of functions and formal parameters, as well as comments associated with them, to predict a likely type signature of a function. Type signature here means the types of function parameters and the return type of the function, e.g., expressed

via `@param` and `@return` in Figure 4.1. We formulate the type inference task as a classification problem and show how to use an LSTM-based recurrent neural network to address it effectively and efficiently. The approach trains the machine learning model based on a corpus of type-annotated functions, and then predicts types for previously unseen code.

There are four reasons why NL2Type works well in practice. First, developers use identifier names and comments to communicate the semantics of code. As a result, most human-written code contains meaningful natural language elements, which provide a rich source of knowledge. Second, source code has been found to be repetitive and predictable, even across different developers and projects [83]. Third, probabilistic models, such as the deep learning model used by NL2Type, are a great fit to handle the inherently fuzzy natural language information [72]. Finally, our work benefits from the fact that some developers annotate their JavaScript code with types, giving NL2Type sufficient data to learn from.

We are aware of two existing approaches, JSNice [144] and DeepTyper [218], that also use machine learning to predict types in JavaScript. JSNice analyzes the structure of code, in particular relationships between program elements, to infer types. Instead, we consider natural language information, which allows NL2Type to make predictions even for functions with very little code. Moreover, our approach is language-independent, as it does not depend on a language-specific analysis to extract relations between program elements. DeepTyper uses a sequence-to-sequence neural network to predict a sequence of types from a sequence of tokens. Similar to us, they also consider some natural language elements of the code. However, their approach considers only identifier names, not comments, missing a valuable source of type hints, and they frame the problem as sequence-to-sequence translation, while we frame it as a classification problem.

We envision NL2Type to be valuable in several usage scenarios. For code that does not yet have formal type annotations, the approach serves as an assistance tool that suggests types to reduce the manual annotation effort. For code that already has type annotations, NL2Type checks for inconsistencies between these annotations and natural language information, which exposes incorrect annotations, misleading identifier names, and confusing comments. Another usage scenario is improving type-related IDE features, such as code completion or refactoring, for code that does not have any type annotations.

We evaluate NL2Type with 162,673 JavaScript files from open-source projects. After learning from a subset of these files, the approach predicts types in the remaining files with a precision of 84.1% and a recall of 78.9%, giving an F1-score of 81.4%. When considering the top-5 suggested types, precision and recall even increase to 95.5% and 89.6%, respectively. Comparing our approach to JSNice [144] and DeepTyper [218], we find that NL2Type significantly outperforms both approaches. When combining NL2Type with JSNice, we find that 27.8% of all correctly predicted types are found exclusively by NL2Type, showing that our approach not only improves upon, but also complements existing work. Beyond predicting likely types for code where annotations are missing, we use NL2Type to check for inconsistencies in existing type annotations. We rank the reported inconsistency warnings by the confidence of the prediction and manually inspect the top 50. 39 out of 50 warnings are valuable, in the sense that developers should fix an incorrect type annotation or improve a misleading natural language element in the code. Finally, the approach is efficient enough for practical use. Training takes 93 minutes in total, and predicting types for a function takes 72ms, on average.

In summary, this chapter contributes the following:

- The insight that natural language information is a valuable, yet currently underused source of information for inferring types in a dynamically typed language.
- A neural network-based machine learning model that exploits this insight to predict type annotations for JavaScript functions.
- Empirical evidence that the approach is highly effective at suggesting types and that it clearly outperforms state-of-the-art approaches.
- Empirical evidence that the approach is effective at finding inconsistencies between type annotations and natural language elements, a problem not considered before.

4.2 LEARNING A MODEL TO PREDICT TYPES

This section describes NL2Type, our learning-based approach for predicting the type signatures of functions from natural language information embedded in code. Figure 4.2 gives an overview of the

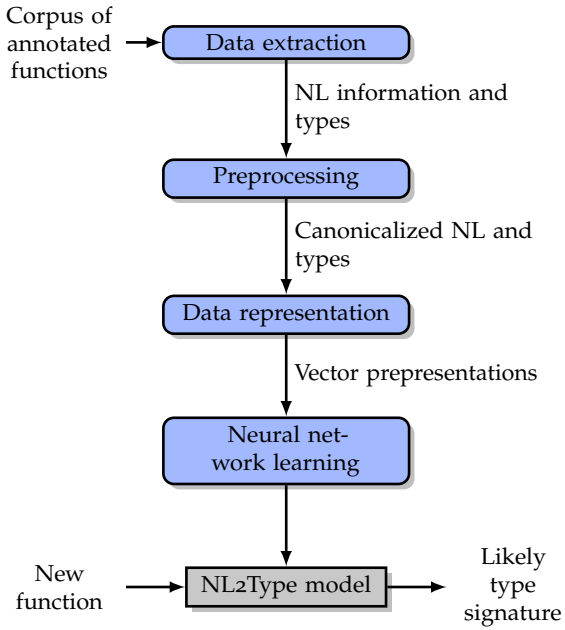


Figure 4.2: Overview of the approach.

approach, which consists of two phases: a *learning phase*, shown in blue in the top part of the figure, which learns a neural model from a corpus of code with type annotations, and a *prediction phase*, shown in gray in the bottom part of the figure, which uses the learned model to predict types for previously unseen code. To prepare the given code for learning, a lightweight static analysis extracts natural language and type data (Section 4.2.1) and preprocesses these data using natural language processing techniques (Section 4.2.2). Section 4.2.3 describes how NL2Type transforms the data into a representation that captures the semantic relations between words, which is then fed into a neural network that learns to predict type signatures (Section 4.2.4). Once the model is trained, querying it with natural language information extracted from a previously unseen function yields a likely type signature for the function (Section 4.2.5).

Extracted function data:

n_f	c_f	c_r	t_r
getArea	Calculates the area of a rectangle.	The area of the rectangle in meters. May also be used for squares.	number

Preprocessed function data:

n_f	c_f	c_r	t_r
get area	calculate area rectangle	area rectangle meter may also use square	number

Figure 4.3: Example of data extraction and preprocessing.

4.2.1 *Data Extraction*

The goal of the data extraction step is to gather natural language information and type signatures associated with functions. To this end, a lightweight static analysis visits each function in the given corpus of code. We focus on functions with JSDoc annotations, an annotation format that is widely used to specify comments and types. For each JavaScript function, the analysis extracts the following:

Definition 4.1 (Function data)**.** *For a given function f, the extracted function data is a tuple (n_f, c_f, c_r, t_r, P) where*

$$
\begin{aligned}
n_f &= \textit{name of the function } f \\
c_f &= \textit{comment associated with } f \\
c_r &= \textit{comment associated with return type of } f \\
t_r &= \textit{return type of } f \\
P &= \textit{sequence of parameter data}
\end{aligned}
$$

The sequence P of parameter data is a sequence of tuples (n_p, c_p, t_p) where

$$
\begin{aligned}
n_p &= \textit{name of the formal parameter } p \\
c_p &= \textit{comment associated with } p \\
t_p &= \textit{type of } p
\end{aligned}
$$

For example, the upper table in Figure 4.3 shows the function data extracted from the JavaScript code in Figure 4.1. We omit the parameter data for space reasons.

4.2.2 *Preprocessing*

To prepare the natural language information extracted in the previous step for effective learning, NL2Type automatically preprocesses the function data using natural language processing techniques. The goal of this step is to canonicalize natural language words and to remove uninformative words.

At first, we tokenize all natural language data into words. The approach tokenizes the extracted comments, c_f, c_r, and c_p, on the space character. For the extracted names of functions and parameters, n_f and n_p, we tokenize each name based on the camel-case convention, which is the recommended naming convention in JavaScript. For example, the name "getRectangleArea" is tokenized into three words: "get", "Rectangle", and "Area". Beyond camel-case, other tokenization techniques for identifier names [71] could be plugged into NL2Type.

After tokenization, the approach removes all punctuation, except for periods, and converts all characters to lowercase. By converting to lowercase, we reduce the vocabulary size without losing much semantic information. The approach also removes stopwords, i.e., words that appear in various contexts and therefore do not add much information, such as "the" and "a". Finally, the approach lemmatizes all words, i.e., it reduces the inflicted forms of a word, e.g., "running", "runs", "ran", to its base form, e.g., "run".

For our running example in Figure 4.1, the lower table in Figure 4.3 shows the function data after preprocessing.

4.2.3 *Data Representation*

To feed the extracted data into a machine learning model, we need to represent it as vectors. The following describes our vector representations of natural language words and of types.

4.2.3.1 *Representing Natural Language Information*

To enable NL2Type to reason about the meaning of natural language words, we build upon word embeddings, a popular technique to

map words into a continuous vector space. The key property of embeddings is to preserve semantic similarities by mapping words that have a similar meaning to similar vectors. For example, assuming we map words into a 3-dimensional space, then "nation" and "country" may have vectors $[0.5, 0.9, -0.6]$ and $[0.5, 0.8, -0.7]$. In practice, embeddings map words into larger spaces; we use vectors of length 100 for our evaluation.

More formally, a word embedding is a map $E : V \to \mathbb{R}^k$ that assigns to each word $w \in V$ in the vocabulary a k-dimensional vector of real numbers. To learn word embeddings, NL2Type builds upon Word2Vec [100], which takes a set S of sentences composed of words in V and learns the embedding of a word w from the contexts in which w occurs. Context here means the words preceding and following w, where the number of context words to consider is a configurable parameter (ten in our evaluation).

NL2Type learns two word embeddings: an embedding E_c for words that occur in comments and an embeddings E_n for words that occur in identifier names. The rationale for having two instead of just one embedding is that identifier names tend to contain more source code-specific jargon and abbreviations than comments. To learn E_c, the set of sentences S consists of all sequences of words in the preprocessed comments c_f, c_r, and c_p. For example, for the word "rectangle" in the lower table in Figure 4.3, the comments c_f and c_r give two sequences of words in which "rectangle" occurs. For a larger corpus of code, many more such sequences are available. Similarly, to learn E_n, the set of sentences S consists of the sequences of words in the preprocessed identifier names n_f and n_p. For both embeddings, we consider only words that occur at least five times in the training data, to prevent the embedding from overfitting to few contexts.

A possible alternative to learning word embeddings from data extracted from a code corpus would be to use publicly available, pre-trained embeddings, e.g., the Google News word embeddings.[1] However, such pre-trained embeddings are typically trained on sentences that use a different vocabulary than that found in real-world JavaScript code or on sentences where some words have a different meaning than in source code. For example, words like "push" or "float" may have a different meaning in a programming context than in common usage, while other words, e.g., "int", occur often in a programming context but not at all in common usage.

1 https://code.google.com/archive/p/word2vec/

4.2.3.2 *Representing Types*

In addition to the natural language information, which is the input to NL2Type, we must also represent the to-be-predicted types as vectors. Given the set T_{all} of all types that occur either as a function return type t_f or as a parameter type t_p in the training corpus, the approach focuses on a subset $T \subseteq T_{all}$ of frequently occurring types. The reason for bounding the size of T is that types have a long-tail distribution, i.e., a few types occur very frequently while many other types occur only rarely (Section 4.4.6). Predicting more frequent types covers a large percentage of all type occurrences, whereas predicting less frequent types is more difficult, as there is less data to learn from. For a specific size $|T|$, we select the $|T| - 1$ most frequent types from T_{all} and add an artificial type "other" that represent all other types and that indicates that NL2Type cannot predict the type. The size of T is a configuration parameter and we evaluate its influence in Section 4.4.6. For the evaluation, we consider the 1,000 most common types, including the built-in types of the JavaScript language, e.g., `boolean` and `number`, and custom types, e.g., `Graphics` and `Point3d`.

Given the set T, we represent a type $t \in T$ using a one-hot vector, i.e., a vector of length $|T|$, where all elements are zero except for one specific element set to one for each word. For example, the type `boolean` may be represented by a vector $[0, 0, 1, , 0, .., 0]$ that consists of 999 zeros and a single one.

4.2.4 *Training the Model*

Based on the vector representations of natural language information and types, NL2Type learns to predict the latter from the former. We use a neural network-based machine learning model for this purpose because neural networks have been shown to be highly effective at reasoning about natural language information. Specifically, we adopt a recurrent neural network based on long short-term memory (LSTM) units. Recurrent neural networks are well suited for ordered input data, such as sequences of natural language words. LSTMs are effective for data with both long-term and short-term dependencies. They have been successfully applied to a number of problems in natural language processing that are similar to our classification problem, such as sentiment analysis, which classifies texts into different cate-

gories [145, 161, 162]. The following describes the data points used for training the model and the architecture of the neural network.

4.2.4.1 *Data Points*

We transform the extracted and preprocessed function data into a set of data points. Each data point represents a single type and the natural language information associated with it. We distinguish two kinds of data points, one for return types and another for parameter types.

Definition 4.2 (Data points)**.** *A data point is a pair (N, t) of natural language information N and a type t. Given the function data (n_f, c_f, c_r, t_r, P) of a function, where P is a sequence $P = [(n_p^1, c_p^1, t_p^1), .., (n_p^{|P|}, c_p^{|P|}, t_p^{|P|})]$ of parameter data, we have two kinds of data points:*

1. *One data point for the return type with:*
   $N = (n_f, c_f, c_r, n_p^1, .., n_p^{|P|})$ *and* $t = t_r$.
2. *$|P|$ data points for the parameter types with:*
   $N = (n_p^i, c_p^i)$ *and* $t = t_p^i$.

For example, for the function in Figure 4.1, there are three data points:

1. For the return type:

   N = (area, calculate area rectangle, area rectangle meter may also use square, length, breadth)

   t = number

2. For the first parameter:

   N = (length, length rectangle)

   t = number

   For the second parameter:

   N = (breadth, breadth rectangle)

   t = number

Given a set of data points (N, t), the task solved by the neural network is to predict t from N. We train a single model for both

return types and parameter types because both tasks are similar and it enables the model to learn from all available data. To feed data points into the neural network, we transform each data point into a sequence of input vectors and an output vector, using the vector representations from Section 4.2.3. Intuitively, the input is the sequence of embeddings of words in the natural language information N, and the output vector is the vector representation of the type t.

To formally define the input vectors, consider a helper function $E^* : w_1, .., w_l \to \mathbb{R}^{l \times k}$ that takes a sequence of l words, maps each word to a vector representation using the embedding function $E : w \to \mathbb{R}^k$, and then yields the sequence of these vectors. The embedding E refers to E_n and E_c for names and comments, respectively, as described in Section 4.2.3. To ensure that all input vectors have the same length $l \times k$, no matter how many natural language words the static analysis could extract from the source code, the helper function E^* truncates word sequences to a maximum length and pads word sequences that are too short with zeros. We discuss and evaluate the length limits in Section 4.4.6.

Based on this helper function, the input for a data point that represents a return type is the following sequence of vectors (where $\circ$ chains vectors into a sequence):

$$K_{ret} \circ E^*(c_f) \circ E^*(n_p^1) \circ ... \circ E^*(n_p^{|P|}) \circ E^*(n_f) \circ E^*(c_r)$$

Likewise, the input for a data point that represents a parameter type is the following sequence of vectors:

$$K_{param} \circ E^*(c_p) \circ Z \circ ... \circ Z \circ E^*(n_p) \circ E^*(c_r)$$

The vectors K_{ret} and K_{param} are special marker vectors that indicate to the network what kind of type to predict, i.e., whether the type is a return type or a parameter type. Making the kind of type explicit enables the network to distinguish between both kinds if necessary. The Z vectors are padding vectors of zeros that we use to ensure that the input sequences of return types and parameter types have the same length. In addition to concatenating vectors, each $\circ$ also inserts a vector of ones into the sequence, as a delimiter between the different natural language elements, which helps the network understand the structure of the data.

For instance, recall the three examples of data points given above. The natural language part N of each of them is transformed into

a sequence of real-valued vectors based on the embeddings of the natural language words in N. Due to the padding, all three sequences have the same length.

4.2.4.2 *Neural Network Architecture*

Given the data points described above, NL2Type learns a function $m : \mathbb{R}^{x \times k} \mapsto \mathbb{R}^{|T|}$ where x is the total number of word embeddings in an input sequence and $|T|$ is the number of types we are trying to predict. Using the length limits as set in our evaluation, the network maps a sequence of $x = 43$ vectors of length $k = 100$ to a vector of length $|T| = 1,000$.

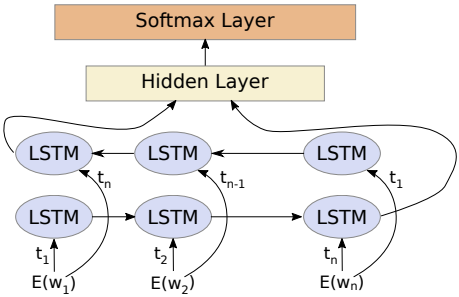


Figure 4.4: Architecture of neural network used in NL2Type.

To learn the function m, we use a bi-directional LSTM-based recurrent neural network, as illustrated in Figure 4.4. The network takes a sequence of $\mathbb{R}^k$ vectors, at each step consumes one vector, and updates its internal state (represented by the "LSTM" nodes). After consuming all the vectors for a single data point, the network feeds the internal state through a hidden layer to the output layer. The output layer uses the softmax function, which yields a vector of real-valued numbers in $[0, 1]$ so that the sum of all numbers is equal to one. That is, the output can be interpreted as a probability distribution. During training, the backpropagation algorithm adapts the weights of the network to minimize the error between the predicted and the expected type.

4.2.5 *Prediction*

Once the model is sufficiently trained, it can predict the types of previously unseen functions. To query the model with a new function, we extract and preprocess all natural language information associated with the function, and create one sequence of input vectors for each type associated with the function (i.e., one sequence for the return type and one sequence for each parameter type). Then, each such input sequence is given to the network, which yields a type vector in $\mathbb{R}^{|T|}$. The type vector can be interpreted as a probability distribution over the types in T. For example, suppose that $T = \{number, boolean, function, other\}$ and that the predicted type vector is $[0.6, 0.2, 0.1, 0.1]$. We interpret this prediction as a 60% probability that the type is "number", 20% that the type is "boolean", 10% that the type is "function", and 10% that the type is any other type. If the most likely type is "other", the network essentially says that it cannot predict a suitable type for the given natural language information.

4.3 APPLICATIONS

The previous section describes a general model to predict the return type and the parameter types of functions from natural language information. This model has several applications, which we present in the following. All these applications query NL2Type as described in Section 4.2.5.

4.3.1 *Suggesting Type Annotations*

The perhaps most obvious application of NL2Type is to support developers in the process of annotating code with types by suggesting type annotations. Adding type annotations to functions enables an effective use of type systems for JavaScript, such as Flow and TypeScript, and it provides useful API documentation. For the large number of functions in legacy JavaScript code without type annotations, NL2Type can suggest types during the annotation process. To this end, the developer queries the model for each type and uses the predicted type vector as a ranked list of type suggestions.

4.3.2 *Improving Type-based IDE Features*

IDEs use type information for making suggestions to developers, such as how to complete partial code. For example, consider a developer that implements the body of a function and wants to access a property of a parameter of this function. Without type information, the IDE cannot make any accurate suggestions about the property name. For example, the popular WebStorm IDE will simply suggest an alphabetically ordered list of all identifier names used in the current file. NL2Type can improve these suggestions by probabilistically predicting the parameter type of the function, which the IDE can then use to prioritize the suggested property names.

4.3.3 *Detecting Inconsistencies*

In addition to predicting types for functions that are not yet type-annotated, NL2Type can check existing type annotations for inconsistencies. In this scenario, the approach checks whether the natural language information associated with a type matches the annotated type. Finding mismatches is useful for fixing broken type annotations, for changing misleading identifier names, and for improving confusing comments.

Given an annotated function type, we query the NL2Type model with the natural language information associated with the type and compare the type predicted as the most likely with the actual type. To avoid overwhelming developers with spurious inconsistencies, the approach ranks all inconsistencies by how certain the model is in its prediction. One possible ranking approach would be to consider the predicted type vectors and to rank inconsistencies by the highest probability in each vector. For example, suppose the type vector is $[0.9, 0.025, 0.025, 0.05]$ but the type represented by the first element does not match the annotated type. Based on the type vector, the model appears to be very certain of its prediction and we would rank this inconsistency high. Unfortunately, this naive ranking approach does not work well in practice because neural networks tend to be too confident in their predictions. The underlying reason, as shown by Guo et al. [188], is that for a softmax function over more than two classes, the output of the softmax function is not a true probability distribution.

Instead of ranking inconsistencies by the highest value in the type vector, we compute a more reliable estimate of the network's confidence [152]. The key idea is to use dropout, i.e., to purposefully deactivate some neurons, during prediction and to measure how much it influences the outcome of the prediction. For every sequence of input vectors, we query the model multiple times, each time deactivating some probabilistically selected neurons, and record the predicted type vectors. We then measure the variance of the type vectors and consider a prediction with lower variance to be more confident. Finally, we rank all potential inconsistencies by their confidence and report the ranked list to the developer.

4.4 EVALUATION

Our evaluation on real-world JavaScript code focuses on the following research questions:
RQ1: How effective is NL2Type at predicting function type signatures from natural language information?
RQ2: How does the approach compare to existing type prediction techniques [144, 218]?
RQ3: How useful is NL2Type for detecting inconsistencies in existing type annotations?
RQ4: What is the influence of hyperparameters, such as the number $|T|$ of considered types, on the effectiveness of NL2Type?
RQ5: Is NL2Type efficient enough to be applied in practice?

Our implementation and data to reproduce our results are available at `https://github.com/sola-da/NL2Type`.

4.4.1 *Implementation*

We implement NL2Type in Python based on several existing tools and libraries. For the data extraction, the implementation parses every JavaScript file using the JSDoc tool [276], which extracts the comments, the function name, and the parameter names of a function. The preprocessing, including removing stopwords and lemmatization, is implemented based on the Python NLTK library [44]. To convert natural language words into embeddings, we use gensim's Word2Vec module [67]. The neural network that predicts types from a sequence

of embeddings is implemented on top of Keras, a high-level deep learning library, using TensorFlow as a backend [128].

4.4.2 *Experimental Setup*

We evaluate NL2Type on a corpus of 162,673 JavaScript files composed of a corpus from prior work [166] and popular JavaScript libraries downloaded from a content-delivery service [272]. Following common practice in large-scale machine learning, including on software [144, 176, 218, 219, 226], we divide these files into disjoint sets of training files (80%) and testing files (20%). A fixed split into training data and validation data, instead of k-fold cross-validation, reduces computational cost, yet gives accurate results due to the large amount of available data. For all files, we extract data points as described in Section 4.2, which gives a total of 618,990 data points. 31.1% and 68.9% of them are for function return types and parameter types, respectively. Not all data points contain all pieces of natural language information. In particular, 20.3% of all data points do not contain a comment c_f or c_p. Given the data extracted from the training files, we train the embeddings and our model, and then use the data extracted from the testing files to evaluate the trained model. All experiments are run on an Ubuntu 16.04 computer with an Intel Xeon E5-2650 processor with 48 cores, 64GB of memory, and an NVIDIA Tesla P100 GPU with 16GB of memory.

4.4.3 *RQ1: Effectiveness at Predicting Types*

4.4.3.1 *Metrics*

To evaluate the effectiveness of NL2Type in predicting types, we measure precision, recall, and F1-score. Intuitively, precision is the percentage of correct predictions among all predictions, and recall is the percentage of correct predictions among all data points. The F1-score is the harmonic mean of precision and recall. Similar to previous work [218], we report these evaluation metrics for the top-k predicted types, assuming that a user of NL2Type inspects up to k suggested types. We also report the top-1 results, which means that the user considers only the single most likely predicted type.

We define top-k precision as $precision = \frac{pred_{corr}}{pred_{all}}$ where $pred_{corr}$ is the number of predictions where the actual type is in the top-k and

Table 4.1: Precision, recall, and F1-score as percentages of NL2Type, with and without considering comments, and of a naive baseline.

Approach	Top-1			Top-3			Top-5		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5
NL2Type w/o comments	72.3	68.3	70.3	86.6	81.8	84.1	91.4	86.3	88.8
Naive baseline	18.5	17.3	17.9	49.0	46.0	47.4	66.3	62.3	64.2

```
/** Get the appropriate anchor and focus node/offset
  * pairs for IE.
  * @param {DOMElement} node
  * @return {object}
  */
function getIEOffsets(node) {
  ...
}
```

Figure 4.5: Function with correctly predicted type signature.

$pred_{all}$ is the number of data points for which the model makes a prediction at all. If the model suggests "other" as the most likely type, it indicates that it cannot make a good prediction, and we count it neither in $pred_{all}$ nor in $pred_{corr}$. The top-k recall is defined as $recall = \frac{pred_{corr}}{dps}$ where dps is the number of all data points.

4.4.3.2 *Results*

Table 4.1 shows the precision, recall, and F1-score of the type predictions. The first row shows the default approach, as described in Section 4.2. When considering the first suggested type only, the approach achieves 84.1% precision with a recall of 78.9%. When considering the top-5 suggested types, the precision and recall increase to 95.5% and 89.6%, respectively. The results for parameter types and for return types are similar to each other, showing that NL2Type is effective for both kinds of types. For example, Figure 4.5 shows a function for which NL2Type correctly predicts the parameter type and the return type. Note that the parameter type, `DOMElement`, is not a built-in JavaScript type, but nevertheless predicted correctly, presumably from the words "node" and "IE". Overall, these results show that the approach is highly effective at making accurate type suggestions for the majority of JavaScript functions.

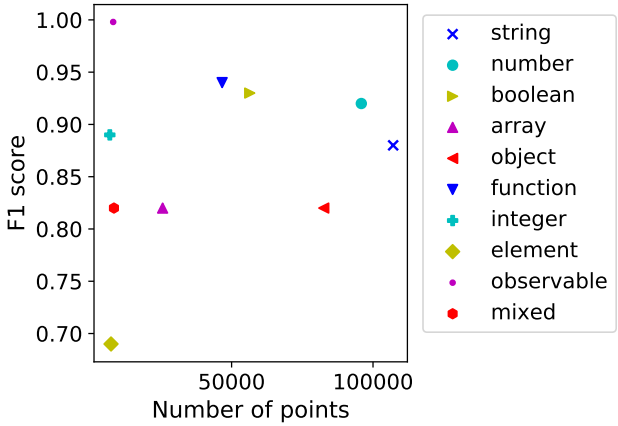


Figure 4.6: Relation between F1-score and amount of data available for a type.

To better understand whether the effectiveness depends on the amount of training data, Figure 4.6 shows for the ten most common types the F1-score along with the number of data points for the type. We find little correlation between the amount of available data and the prediction's F1-score, suggesting that the data we train NL2Type on is sufficient for commonly used types in JavaScript. Interestingly, the F1 scores differ between types, presumably because some types are more likely than others to have a comment or name that reveals the type. For example, functions with return type `boolean` often have a name that begins with "is" or "has", while for "object", inferring the type is less straightforward.

Because not all functions come with comments, but all functions and their parameters have a name, we also evaluate a variant of NL2Type that does not consider any comments. Instead, the input given to the neural network consist only of the function name and parameter names. The second row in Table 4.1 shows the results for this variant of the approach. As expected, the precision, recall, and F1-score are lower than for the full approach, because some valuable parts of the input are omitted. However, the approach still makes accurate suggestions that are likely to be useful in practice. We conclude from these results that using comments as part of the input considered by NL2Type is beneficial, but that comments are not essential to the effectiveness of the approach.

We also compare NL2Type to a naive baseline that simply predicts the k most common types every time it is queried. In particular, when asked for the top-1 type suggestion, the baseline always suggests `string` because this is the most common type. The third row in Table 4.1 shows the effectiveness of this baseline. NL2Type is clearly better than the baseline, e.g., improving the F1-score for the top-1 suggestion by a factor of 4.5x.

4.4.4 *RQ2: Comparison with Prior Work*

The two closest existing approaches are JSNice [144] and DeepTyper [218]. Both use the implementation of a function to infer the function's type signature, whereas our approach ignores the function implementation and instead focuses on natural language information associated with the function. JSNice uses structured prediction on a graph of dependencies that express structural code properties, such as what kind of statement a variable occurs in. Similar to our work, they train their model with existing type-annotated JavaScript code. DeepTyper is similar to our work in the sense that they also use a neural network model. However, they train the model with an aligned code corpus, i.e., pairs of TypeScript and JavaScript programs, which are generated from existing TypeScript code.

4.4.4.1 *Comparison with JSNice*

To compare with JSNice, we download their publicly available artifact[277] and train a model with the same training data as for NL2Type, using the command line arguments given in the artifact's README file. We run the tool with a time limit of two minutes per file and remove any files that exceed that limit from the training corpus of both JSNice and NL2Type. In total, 7,025 files are removed for this reason. Once trained, we evaluate JSNice on our testing set. Because JSNice tries to predict types only for minified files, we minify the testing files using a script provided in the JSNice artifact. All results reported for JSNice are for the top-1 suggestion only, because the JSNice artifact reports only the most likely type suggestion. Beside types, JSNice also predicts other code properties, e.g., identifier names; we consider only the predicted parameter types and return types for our comparison.

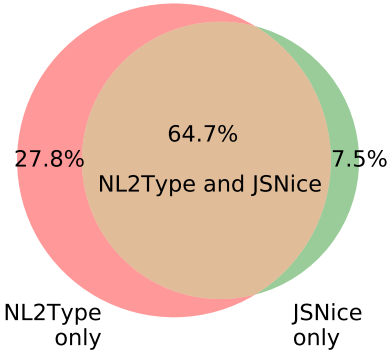


Figure 4.7: Venn diagram showing the overlap of data points correctly predicted by NL2Type and JSNice.

The precision achieved by JSNice is 62.5% with a recall of 45.0%, which gives an F1-score of 52.3%.[2] Comparing these results to those in Table 4.1 shows that NL2Type clearly outperforms the state-of-the-art approach. In particular, the F1-score of NL2Type is 29.1% higher than that of JSNice, which is a significant improvement. One reason why NL2Type outperforms JSNice is that it successfully predicts types for functions independent of the amount of code in the function body, whereas JSNice relies on type hints provided by the function body. To evaluate to what extent NL2Type and JSNice complement each other, Figure 4.7 shows how many of the correctly predicted types overlap. The figure considers the top-1 predictions only. Of all data points that are predicted correctly by either NL2Type or JSNice, 27.8% are predicted only by NL2Type, while 7.5% are predicted only by JSNice. Overall, these results show that our approach of considering natural language information complements and improves upon prior work that focuses on the implementation of a function.

4.4.4.2 *Comparison with DeepTyper*

We compare with DeepTyper [218] based on their publicly available artifact [273]. As we do for JSNice, we compare the top-1 predictions of DeepTyper and compute our precision and recall metrics. For a fair comparison, we implement a TypeScript frontend for NL2Type and

2 Note that our definition of recall is different from the one used in [144], which defines recall as the percentage of data points for which any prediction is made, either correct or incorrect.

```
/** Utility function to ensure that object properties are
  * copied by value, and not by reference
  * @private
  * @param {Object} target Target object to copy
  *                        properties into
  * @param {Object} source Source object for the
  *                        properties to copy
  * @param {string} propertyObj Object containing
  *                             properties names we
  *                             want to loop over
  */
function deepCopyProperties(target, source, propertyObj) {
  ...
}
```

Figure 4.8: Incorrect type annotation found by NL2Type: Our model correctly predicts the third parameter to be `object`.

then use the TypeScript data set used in [218]. NL2Type achieves a precision of 77.5% and a recall of 44.6%, compared to 68.6% precision and 44.0% recall by [218].[3] That is, when using the same data set for both approaches, our model significantly improves precision while slightly improving recall. The results of NL2Type are less strong than when applying it to the JavaScript data set because the TypeScript data set is smaller and because its types have a longer-tail distribution.

4.4.5 *RQ3: Usefulness for Detecting Inconsistencies*

An application of NL2Type that goes beyond predicting types in code without type annotations is as a tool to detect inconsistencies in existing type annotations. To evaluate the usefulness of NL2Type for this task, we get a ranked list of potential inconsistencies, as described in Section 4.3.3, and manually inspect the top 50 of this list. We classify each potential inconsistency into one of three categories.

1) Inconsistency. We classify a warning as an *inconsistency* if the source code, the comments, and the type annotations are inconsistent with each other, because at least two of these three are contradictory. Developers should fix these inconsistencies by adapting either the type annotations, the comments, or the code. Figure 4.8 shows an example of an inconsistency due to an incorrect type annotation. Our

3 The results differ from those reported in [218] for two reasons: (i) We use a different definition of recall ($\frac{pred_{corr}}{dps}$ and not $\frac{pred_{all}}{dps}$). (ii) We do not apply any confidence threshold when using DeepTyper, whereas their best precision/recall results are with a threshold optimized after-the-fact.

```
/** Tests to see if a point (x, y) is within a range of
  * current Point
  * @param {Numeric} x - the x coordinate of tested point
  * @param {Numeric} y - the x coordinate of tested point
  * @param {Numeric} radius - the radius of the vicinity
  **/
near: function(x, y, radius) {
  var distance = Math.sqrt(Math.pow(this.x - x, 2)
      + Math.pow(this.y - y, 2));
  return (distance <= radius);
}
```

Figure 4.9: Non-standard type annotation detected by NL2Type: Our model predicts the parameters to have type `number`, but the code annotates them as `Numeric`, which is not a legal JavaScript type.

```
/** Calculate the average of two 3d points
  * @param {Point3d} a
  * @param {Point3d} b
  * @return {Point3d} The average, (a+b)/2
  */
Point3d.avg = function(a, b) {
  return new Point3d((a.x + b.x) / 2, (a.y + b.y) / 2,
      (a.z + b.z) / 2);
}
```

Figure 4.10: Misclassification: NL2Type predicts a `number` return value, but the code indeed returns an object of type `Point3d`.

model correctly predicts that the type of the `propertyObj` should be `object`, but the code instead annotates it as `string`.

2) Non-standard type annotation. We classify a warning as *non-standard type annotation* if the type annotation refers to a "type" that is not a legal JavaScript type, but may nevertheless convey the intended type to a human developer. For example, Figure 4.9 shows a function where the parameters are annotated as `Numeric`. However, this type is not a legal JavaScript type, and the developer intended the types to be `number`, which NL2Type correctly predicts. Because NL2Type learns conventions from a large corpus of code, it tends to predict the standard type instead of the non-standard type. To benefit from one of the type checkers built on top of JavaScript [275, 280] and from improved IDE support, developers should replace non-standard types with the corresponding standard type.

3) Misclassification. We call a warning a *misclassification* if the type predicted by NL2Type is incorrect and the code need not be changed in any way. For example, the function in Figure 4.10 returns an object that represents a point in the 3-dimensional space, as specified in the `@return` annotation. However, the function name and the comment of

Table 4.2: Classification of potential inconsistencies reported by NL2Type.

Category	Total	Percentage
All inspected warnings	50	100%
Inconsistencies	25	50%
Non-standard type annotations	14	28%
Misclassifications	11	22%

Table 4.3: Length limits for inputs processed by the neural network.

	Avg. in data set	Maximum considered	Fully covered data points
Words in function or parameter name	1.6	6	99.9%
Words in function comment	5.9	12	89.9%
Words in parameter or return comment	0.5	10	99.8%
Number of parameters	1.1	10	98.5%

the function mislead NL2Type to predict `number`. Misclassifications can result because NL2Type has not seen enough data similar to the given natural language information during training or because the code, comments, or identifier names are unusual w.r.t. the training corpus.

Table 4.2 shows how the 50 manually inspected warnings reported by NL2Type distribute across the above categories. Most warnings point to code that deserves action by the developer: fixing a type annotation, improving a comment, or changing the code. The percentage of actionable warnings is 78%. We conclude that NL2Type provides a useful tool for checking type annotations for inconsistencies. To the best of our knowledge, our work is the first to show probabilistic type inference to be effective for this task.

4.4.6 *RQ4: Parameter Selection*

Table 4.4: Impact of the number of considered types on the number of covered unique types and data points.

Number of types	Unique types covered	Data points covered
5	0.04%	61.9%
50	0.44%	81.6%
500	4.37%	91.7%
1,000	8.73%	94.1%
5,000	43.65%	98.6%
10,000	87.30%	99.9%

4.4.6.1 *Parameters for Input Representation*

As discussed in Section 4.2.4, each part of the input sequence has a fixed length, and data that are too short or too long are padded with zeros or truncated, respectively. Table 4.3 shows the length limits we use and how many of all data points these limits cover without any truncation. For example, we consider up to six words as part of a function or parameter name, which covers 99.9% of all names in our data set. The parameters are selected to cover the large majority of the available natural language data.

4.4.6.2 *Parameters for Output Representation*

The output of the neural network is a type vector of length $|T|$, which determines how many different types the model can predict. The set T_{all} of all types in our data set contains 11,454 types. Because classification problems become harder when the number of classes increases, and because the frequency of types follows a long-tail distribution, we focus on a subset $|T| \subseteq T_{all}$. Table 4.4 shows how the size of $|T|$ influences the percentage of all data points covered by the considered types. For example, $|T| = 1,000$ covers 94.1% of all data points.

The trade-off in choosing $|T|$ is between precision and recall. Choosing a larger $|T|$ has the potential to increase recall because the model can predict the types of more data points. However, this potential increase of recall comes at the cost of lower precision because the model must choose from more possible types and because the amount of training data quickly decreases for less frequent types. To pick

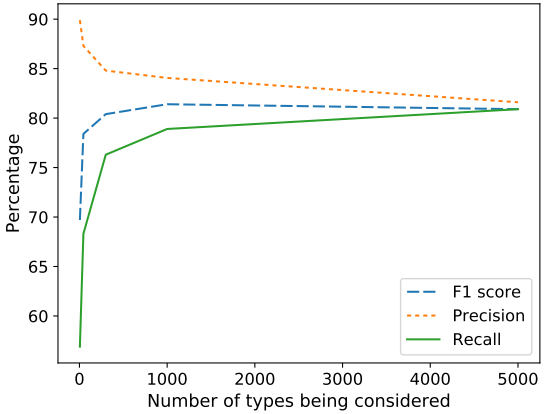


Figure 4.11: Effectiveness of NL2Type depending on the number $|T|$ of types.

$|T|$, we train and evaluate models for $5 \leq |T| \leq 5,000$ and measure precision, recall, and F1-score for the top-1 prediction. The results in Figure 4.11 show the tradeoff between precision and recall. The approach reaches the maximum F1-score at $|T| = 1,000$, which is the value we select for the evaluation.

4.4.6.3 *Parameters for Learning*

Table 4.5 summarizes the values of parameters related to the learning parts of NL2Type. The hyperparameters of the neural networks are selected based on values suggested by previous work and by our initial experiments. We stop training after twelve epochs because it is sufficient to saturate the accuracy.

4.4.7 *RQ5: Efficiency*

The total time taken by NL2Type is the sum of the time for five subtasks. First, data extraction takes 44ms per function, on average, most of which is spent in the JSDoc tool while parsing JavaScript code. Second, data pre-processing takes 23ms per function, on average. Third, learning both the word embeddings takes about 2 minutes in total. Fourth, the one-time effort of training the model takes about 93 minutes. This time is relatively little, compared to some other neural

Table 4.5: Parameters and their default values.

Parameter	Value
Neural network to predict word embeddings:	
Word embedding size	100
Context size	5
Minimum occurrences of a word	5
Neural network to predict types:	
Hidden layer size	256
Batch size	256
Number of epochs used for training	12
Dropout of model	20%
Loss function for model	Categorical cross entropy
Optimizer	Adam

networks, because of the small number of units in the hidden layer. Finally, predicting types for a new function takes the time to extract and pre-process data from the function plus 5ms per function, on average, to query the model. We conclude that NL2Type is efficient enough to apply to real-world JavaScript code and to quickly give feedback to developers.

4.5 CONCLUSION

This chapter addresses the lack of types in dynamically typed languages (Challenge C-II, Chapter 1). In contrast to traditional techniques, which infer types from the program source code, we tackle the problem by analyzing natural language information embedded in the code. We present NL2Type, a new learning-based approach that extracts identifier names and comments from a corpus and feeds into a recurrent neural network to predict function signatures. The approach yields a neural model that helps annotating not yet annotated JavaScript code by suggesting types to the developer. Our experiments show that NL2Type predicts types with an F1-score of 81.4% for the top-most prediction and of 92.5% for the top-5 predictions, which clearly outperforms existing work on learning to predict types.

In addition to predicting missing types, we show how to use the model to identify inconsistencies in existing type annotations. By inspecting 50 warnings about such inconsistencies, we find 39 problems that require developer attention, e.g., because type annotations are incorrect or because they do not match the comments associated with a function. The broader impact of our work is to show that natural language information in code is a currently underused resource that is useful for predicting program properties.

Part II

CORPUS-BASED DYNAMIC ANALYSIS TO FIND SOFTWARE BUGS

While static analysis can provide many interesting properties about programs, it is still an approximation of the actual behavior. More precise information is obtained by analyzing the runtime behavior of programs, also known as dynamic analysis. We use dynamic analysis to uncover bugs and obtain data for training neural classifiers.

5

CONFLICTJS: FINDING AND UNDERSTANDING CONFLICTS BETWEEN JAVASCRIPT LIBRARIES

In the previous part (Chapter 2, Chapter 3, Chapter 4) of this dissertation, we provide details on the corpus-based static analysis approaches. While static analysis is useful, one of its drawbacks are the over-approximations made about program behavior which can lead to occasional false positives. This is particularly significant for dynamically typed languages such as JavaScript and Python. As a result, in this and the next chapter, we present corpus-based dynamic analysis approaches. The current chapter dynamically analyses a collection of JavaScript libraries to check if including pairs of such libraries would lead to unexpected behaviors such as crashes.

5.1 MOTIVATION

The popularity of JavaScript has lead to the development of numerous JavaScript libraries. For example, a popular content delivery network that hosts JavaScript libraries provides over 3,000 different libraries.[1]. Libraries are ubiquitous and many applications use multiple libraries. One estimate is that 75% of the top 10 million websites use at least one of the top 18 libraries.[2] A recent study on the top 75,000 Alexa websites [193] reports that the number of externally hosted scripts that a website includes has a median of 9 and a maximum of 202.

Unfortunately, using multiple independently developed libraries together may cause unexpected behavior. The reason is that JavaScript does not have namespaces but instead, all libraries share a single global namespace. As a result, a value or a function "exported" by one library may be easily overwritten, modified, deleted, or accidentally

1 https://cdnjs.com/libraries
2 https://w3techs.com/technologies/overview/javascript_library/all

```
// Strophe.js
window.Base64 = {
  encode: function(b) {
    /* code */
  }
  decode: function(b) {
   /* code */
  }
};
```

```
// JSEncrypt.js
window.Base64 = {
  unarmor: function(t) {
    /* code */
  }
  decode: function(i) {
    /* code */
  }
};
```

```
// Library client
jsEncrypt = new JSEncrypt();
jsEncrypt.setKey(...);

// Returns false instead of
// decrypted data when
// Strophe.js is loaded
// after JSEncrypt.js.
jsEncrypt.encrypt(...);
```

Figure 5.1: Example of two conflicting libraries and a client that will experience unexpected behavior when loading both libraries.

used by another library. Moreover, libraries may overwrite built-in APIs, sometimes called "monkey patching", and multiple libraries may try to overwrite the same API in different ways. In practice, the problem is compounded by the loose typing in JavaScript, which allows one library to overwrite another library's API even with a type-incompatible value.

As a real-world example of a library conflict found by our approach, consider Figure 5.1. The left side of the figure shows an excerpt of *Strophe.js*, a library that implements the XMPP middleware protocol. The center part of the figure shows *JSEncrypt.js*, a library that provides OpenSSL RSA encryption. Both libraries write to the global variable `Base64`.[3] When included together, the library that is included last will overwrite the `Base64` object of the library that was included first. Such overwriting may cause unexpected behavior in a client of either of these libraries. For example, the right side of the figure shows a client that tries to encrypt some data using *JSEncrypt.js*. When executing this client after loading only *JSEncrypt.js*, the last call returns the encrypted data. However, when executing the client after loading *JSEncrypt.js* and then *Strophe.js*, the last call simply returns `false`. The fact that including an apparently unrelated library breaks the core feature of the encryption library will surprise users and is unintended by the developers of both libraries.

Problems caused by conflicting libraries may occur whenever a developer loads two libraries, which is common practice. Even if a developer explicitly loads only one library, other libraries may be implicitly loaded. Due to the highly dynamic nature of JavaScript, where some code may dynamically load other code, an application

3 The `window` variable is the global object in client-side JavaScript.

developer may implicitly load libraries without even noticing it. For example, websites built on top of content management systems often use plugins, each of which implicitly loads some libraries.[4] Other common ways of implicitly loading libraries are third-party ads, social media services, and news feeds. When a conflict between libraries exists, JavaScript often follows a "no crash" philosophy, i.e., misbehavior may not lead to an exception. As a result, conflicts easily remain unnoticed at library load time or even later, until a user triggers the unexpected behavior, as illustrated in the motivating example.

In principle, there is a sane way for libraries to share the global namespace. Ideally, library developers all follow a "single API object" pattern, where the entire API of the library is encapsulated into a single object. The library then writes this object to a single global variable, e.g., named like the library itself, to minimize the potential for conflicts. In practice, not all libraries follow this pattern, and some global variables, such as `$` and `_`, are particularly popular. Our empirical results show that 71% of all libraries do not follow the "single API object" pattern.

Library conflicts are challenging to detect for a program analysis and difficult to avoid for library developers. One reason is that unintended effects of conflicts typically manifest only at runtime. A purely static analysis can either soundly overapproximate potential conflicts and their effects, which is likely to produce a large number of false positives, in particular for JavaScript, or unsoundly underapproximate them, which may miss conflicts. Another challenge for detecting conflicts is the large number of JavaScript libraries. With thousands of libraries available, and new libraries being added and updated every day, analyzing all possible combinations of libraries leads to a combinatorial explosion that is prohibitive in practice. Currently, there exists no technique for library developers to check whether their library conflicts with another and for library clients to check which combinations of libraries to avoid. Furthermore, it is currently unknown to what extent the problem of library conflicts matters in practice.

This chapter presents ConflictJS, the first automated and scalable technique that analyzes JavaScript libraries for conflicts. We address the huge search space of possible conflicts and the difficulties of

4 For a real-world example, see http://simple-press.com/documentation/codex/faq/troubleshooting/what-is-this-jquery-conflict/.

statically analyzing JavaScript through a two-phase approach that combines dynamic analysis and test synthesis. In the first phase, ConflictJS dynamically analyzes individual libraries to detect writes to the global namespace while loading a library. An offline comparison of these global writes yields a set of potential conflicts between libraries. In the second phase, ConflictJS synthesizes and dynamically analyzes library clients to check if potential conflicts indeed lead to unexpected behavior. The second phase, and therefore also the overall approach, is precise in the sense that every validated conflict certainly occurs in the synthesized client and leads to different behavior depending on the loaded libraries.

We use ConflictJS to analyze and study 951 libraries. The results show that 268 (28%) libraries are potentially conflicting and that 166 (17%) libraries are certainly conflicting with at least one other library. The conflicts may lead to crashes, unexpected behavior, and globally reachable state with unexpected values and types. A manual analysis of conflicting libraries reveals several recurring patterns of root causes for conflicts, which are instructive for library developers, API designers, and language designers. We reported seven of the detected conflicts to the respective library developers, of which four already have been acknowledged and confirmed as problematic. Of the four conflicts, two have been fixed by the developers of the respective libraries.

Compared to existing work on analyzing JavaScript [178], our work is the first to address conflicts among libraries. Existing static analyses focus on type checks of single libraries [112] or assume the presence of library clients [99]. Existing dynamic analyses that target type inconsistencies [142] and other coding problems [132] assume to have inputs to exercise the program, whereas our work synthesizes library clients automatically. JSNose [93] identifies excessive uses of global variables, but focuses on single libraries. Finally, our empirical results relate to existing large-scale studies of JavaScript libraries and their usage [85, 193]. Our work is the first to study library conflicts.

We envision ConflictJS to be useful for developers of libraries and library clients alike. Library developers may use ConflictJS to check whether their library conflicts with others, allowing them to avoid the conflicts by adapting the library. Developers of library clients may use ConflictJS to check which libraries conflict, allowing them to avoid including them together.

In summary, this chapter contributes the following:

- We are the first to address the problem of conflicts among libraries in a language without explicit namespaces.
- We present ConflictJS, an automated and scalable technique to precisely detect conflicts in JavaScript libraries through a combination of dynamic analysis and test synthesis.
- We provide empirical evidence that the approach scales to 951 libraries, where it effectively detects and validates 1,840 conflicts among them.
- We provide our implementation as open-source.[5]

5.2 PROBLEM STATEMENT

This section provides some background, motivates the problem of conflicts among libraries with examples from real-world libraries, and formulates the problem addressed in this chapter.

5.2.1 *Background*

The JavaScript version that is fully supported by most modern browsers is ECMAScript 5 [73]. It does not provide any kind of namespaces or modules at the language level. Instead, library developers rely on several ad-hoc mechanisms to encapsulate code and to export APIs. First, some libraries follow a "single API object" pattern, where the library initializes itself in a local scope and provides its API as properties of a single global object. The most obvious choice for naming this global API object is the name of the library, which typically is unique. For example, *react.js* follows this pattern by exporting its APIs into the global `React` object. The popular *jQuery* library furthermore enables developers to avoid conflicts by specifying the global variable where to provide the library or to even export the library into an existing, non-global object.[6] Second, some libraries build upon the asynchronous module specification (AMD), a module system targeted at client-side JavaScript and implemented as a library, e.g., RequireJS[7]. Third, some libraries use CommonJS, a module system

5 https://github.com/sola-da/ConflictJS
6 https://api.jQuery.com/jQuery.noconflict
7 http://requirejs.org/

targeted at non-client-side JavaScript and implemented as the default module system on the Node.js platform. Unfortunately, these options are neither compatible with each other nor available on all JavaScript platforms. ECMAScript 6 [129] unifies ideas from CommonJS and AMD into language-level module support, and popular JavaScript platforms have started to adopt it. However, since widely used libraries cannot rely on recently added language features, they typically ensure backward compatibility by relying on other ways to export their APIs. In summary, the lack of namespace and modules in currently deployed versions of JavaScript creates a non-trivial problem for library developers.

5.2.2 *Motivating Examples and Classification of Conflicts*

The following section motivates the problem of conflicts between libraries with real-world examples (Table 5.1, 5.2) found using our approach. Furthermore, we use these examples to define four classes of conflicts, based on how the conflicts manifest to a library client. For each example, we show code from two conflicting libraries and a client application that observes different behavior depending on which of the libraries are included and on the order of inclusion.

INCLUSION CONFLICTS This kind of conflict raises an exception when including multiple libraries, without any further interaction between the client and the libraries. The example in the first column illustrates the problem with the *curl* and *dojo* libraries. Loading the second library after loading the first library causes an exception. For a library user, finding such conflicts is non-trivial because the exception depends on the order of including the libraries: Only if a client loads *dojo* before loading *curl* the exception occurs. The documentation of neither of the libraries provides any reference to the other library, presumably because the respective developers are not aware of each other.

TYPE CONFLICTS Type conflicts occur when multiple libraries write type-incompatible values to the same globally reachable location. Table 5.1 presents an example of two libraries, *ocanvas.js* and *aframe.js*, that write to `window.logs` an array and a function, respectively. A client using one of these libraries may rely on the type of the conflicting value and will be surprised if including another library or

Table 5.1: Examples of conflicts between real-world libraries. The first row shows a code example, the second row a client that exposes the conflict and the last row a description about the conflict.

Inclusion conflict	Type conflict

`/* curl.js */`
`window.define`
`  = function K()`
`  { ... };`
 	
`/* dojo.js */`
`var def = function()`
`  { ... };`
`var req = function()`
`  { ... };`
`if (window.define) {`
`  ...`
`} else {`
`  window.define = def;`
`  window.require = req;`
`}`
 	
`// exception`
`window.require();`
	`/* ocanvas.js */`
	`(function(a, b, c)`
	`{`
	`  a.logs = [];`
	`}`
	`(window, document));`
	
	`/* aframe.js */`
	`c = function(e)`
	`{`
	`  ...`
	`};`
	`window.logs = c`

`// client that includes first`
`// curl.js and then dojo.js`
 	
`// exception because require`
`// is undefined`
	`// try to add to`
	`// the 'logs' array`
	`logs.push("log");`
	
	`// exception because`
	`// logs is a function`

Both libraries write to the global variable `define`. To avoid overwriting an already defined variable, e.g, when the same library is included multiple times, *dojo* checks whether `define` is already defined. Unfortunately, the code incorrectly assumes that `require` is always defined together with `define`, causing an exception when trying to call this function. The problem is triggered by any client that includes first *curl* and then *dojo*.	Both libraries write to a global variable `logs`. The type of `logs` is array in *ocanvas* but function in *aframe*.

Table 5.2: Examples of conflicts between real-world libraries. The first row shows a code example, the second row a client that exposes the conflict and the last row a description about the conflict.

Value conflict	Behavior conflict

`/* pako */`
`var pako = {`
`  Deflate: function() { ... },`
`  Inflate: function() { ... },`
`  ...`
`}`
 	
`/* 3Dmol */`
`var pako = {`
`  inflate: function() { ... },`
`  inflateRaw: function() { ... },`
`  ...`
`}`
	`/* jsface */`
	`function O(t, o) {`
	`  ...`
	`}`
	`window.Class = O;`
	
	`/* matreshka */`
	`window.Class`
	`  = function(a, b) { ... }`

`Object.keys(pako);`
 	
`// returns 35 with pako`
`// but 4 with 3Dmol`
	`var v1 = null;`
	`var v2 = "";`
	`v0 = window.Class(v1, v2);`
	
	`// TypeError with matreshka`
	`// but no errors with jsface`

Both libraries overwrite the global variable `pako`. The size of the global variable is different in both cases. This overwriting happens because 3Dmol ships a variant of the pako library that misses some features.	Both libraries write to the same global variable `Class`. The implementation of both differ as illustrated by the client.

changing the order of library inclusion breaks the type assumption. This and the following kinds of conflict are more subtle than inclusion conflicts because they do not lead to an obvious error when simply including the libraries.

VALUE CONFLICTS Similar to type conflicts, this kind of conflict is caused by multiple libraries writing different values to the same globally reachable location. We classify a conflict as value conflict if the values are type-compatible but different. Table 5.2 provides an example where two libraries, *pako* and *3Dmol*, write different values to the same variable `pako`. The root cause of this conflict is that *3Dmol* contains an outdated version of *pako*.

BEHAVIOR CONFLICTS A behavior conflict occurs when multiple libraries store functions at the same globally reachable location, but these functions do not provide the same behavior. Table 5.2 presents an example where two libraries, *jsface* and *matreshka*, overwrite the same variable `Class`. As illustrated by the client code, the two functions provide different behaviors, which may surprise a client that is not aware of the fact that both libraries provide dissimilar implementations of the same global function.

5.2.3 *Problem Statement*

Based on these four types of conflicts, we now formulate the problem addressed in this chapter. The input to our approach is a set $\mathcal{L}$ of libraries. We assume that each $l \in \mathcal{L}$ is supposed to be usable without including any other library in $\mathcal{L}$. In particular, this assumption excludes libraries that extend another library, e.g., libraries that extend the popular *jQuery* library with additional features.

Libraries are used by clients that interact with the APIs of a library. Client here means any sequence of statements that is executed after loading one or more libraries. We denote a client c that executes after loading libraries $l1, .., lk$ as $c_{l1,..,lk}$. We call the sequence $l1, .., lk$ of libraries loaded before executing a client the *library configuration*. The "client" row of Table 5.1 and Table 5.2 show examples of clients.

We target conflicts due to libraries that write to the same globally accessible memory location. In JavaScript, such memory locations are properties of an object. Properties are accessed using either dot notation, e.g., `x.p`, or bracket notation, e.g., `x["p"]`. In either case, the

name of a property is represented by an identifier of string type. For properties of nested objects, the property accessors consist of multiple identifiers, e.g., `window.foo.bar`. We call all property accessors, using either single or multiple identifiers, *access paths*. If the first segment of an access path is globally reachable, we call it a *global access path*. For example, `window.foo.bar` and `window.baz` are global access paths. Since the `window`-prefix is optional in JavaScript, we omit it in the remainder of the chapter, unless needed.

Based on these definitions, we can now define conflicts between pairs of libraries:

Definition 5.1 (Conflict). *Let $l_1, l_2 \in \mathcal{L}$ be two libraries that both write to the same global access path p. These libraries are conflicting with each other if there exists a client so that any of the following is true:*

1. c_{l1} *behaves differently than* c_{l2}
2. c_{l1} *behaves differently than* $c_{l1,l2}$
3. c_{l1} *behaves differently than* $c_{l2,l1}$
4. c_{l2} *behaves differently than* $c_{l1,l2}$
5. c_{l2} *behaves differently than* $c_{l2,l1}$
6. $c_{l1,l2}$ *behaves differently than* $c_{l2,l1}$

The first case means that the same client behaves differently depending on which library is loaded. Such a conflict is relevant for the developers of the libraries because these libraries write different data or functions to the same globally accessible memory location. Cases 2 to 5 mean that a client that includes a single library will change its behavior simply because another library is also included. Such a conflict is relevant for developers of clients who may be surprised that simply including another library causes new behavior. The last case means that a client's behavior changes when swapping the order in which two libraries are included. Again, this case is relevant for client developers because such a change in behavior is surprising.

Based on Definition 5.1, we say that a library is *conflicting* if there exists another library so that both are conflicting with each other. The problem addressed in this chapter is how to find conflicting libraries in a precise way, i.e., without false positives.

5.2.4 *Challenges*

Due to the increasing popularity of JavaScript, there exist thousands of libraries. Only few of them come with representative clients that could serve as test cases. Our work aims at detecting conflicts in an automated and scalable way. Automated here means that the approach requires no input except for a set of libraries. Scalable here means that this set may contains thousands of libraries.

To find conflicts in an automated and scalable way, we must address several challenges. First, the sheer number of JavaScript libraries makes it practically impossible for an analysis to compare all combinations or even all pairs of libraries. For example, given 1,000 libraries, there are about 500,000 pairs of libraries. We address this challenge by identifying potential conflicts during an analysis of individual libraries (Section 5.3.1), which significantly reduces the number of combinations to analyze further. Second, the approach cannot rely on any a-priori available library clients. We address this challenge by synthesizing library clients, guided by the potential conflicts (Section 5.3.2). Third, to validate whether a potential conflict is indeed a conflict, we need to check whether the behavior of clients differs depending on the library configuration. We address this challenge by comparing the runtime behavior of synthesized clients executed with different library configurations (Section 5.3.2).

5.2.5 *Scope and Limitations*

Some challenges are out of the scope of this work. One of them is detecting all library conflicts. While our approach is precise, it is not sound, i.e., it may miss some conflicts. For most interesting program analysis tasks, providing a sound and precise answer is impossible, and we opt for precision in this work. Another out-of-scope question is how many real-world clients suffer from a detected conflict. Instead of addressing this question, our approach shows the existence of a client by synthesizing the client, so that library developers could anticipate conflicts that any possible client may run into and prevent conflicts before they occur. Finally, we focus on pairwise library conflicts and ignore conflicts that arise only if three or more libraries interact with each other.

5.3 APPROACH

This section presents ConflictJS, a scalable and automated approach to find conflicts between libraries. Given a set of libraries, the approach consists of two main steps:

1. *Detection of potential conflicts*. At first, ConflictJS dynamically analyzes individual libraries to identify which globally reachable memory locations they write to. Based on the global writes of each library, the first step then reports a potential conflict for each pair of libraries that write to the same location.

2. *Validation of conflicts*. This step validates whether two libraries that write to the same globally reachable location can indeed cause a client to behave differently depending on the library configuration. To this end, ConflictJS synthesizes clients and compares their behavior across different library configurations. If and only if the approach finds a client with diverging behavior, it reports a conflict.

The remainder of this section explains these two steps in more detail.

5.3.1 *Detection of Potential Conflicts*

To find potential conflicts between libraries, ConflictJS analyzes the global access paths written to by a library. To this end, we dynamically analyze the loading of each library to keep track of the writes made to the global namespace:

Definition 5.2 (Global Writes of a Library). *The global writes of a library l is a set $\mathcal{G}_l = \{p_1, .., p_k\}$ of global access paths to which l writes while loading l.*

For example, if the global object is called `window` and a library writes to it using `window.obj = {prop1:1, prop2:2}`, then the set of global writes is `{obj, obj.prop1, obj.prop2}`.

To compute the global writes of a library, ConflictJS generates a trivial client that simply loads the library and dynamically analyzes the execution. The dynamic analysis updates the set $\mathcal{G}$ when specific runtime events occur, as summarized in Table 5.3. The analysis is guaranteed to observe all global writes that occur while loading the library. In particular, the analysis handles writes to aliases of globally

reachable objects, as illustrated by the example involving `window.Array` in Table 5.3. The access paths of all reachable values, i.e., $paths(v)$ mentioned in Table 5.3 are computed by recursively traversing the properties of the object v. The information whether a variable is global is provided by Jalangi [108] on top of which we implement the analysis.

After extracting the global writes of each library, ConflictJS compares the global writes of all libraries with each other to check for writes to the same global access path. If two libraries share a global write, we classify them as potentially conflicting:

Definition 5.3 (Potentially Conflicting Libraries). *Two libraries $l_1, l_2 \in \mathcal{L}$ are potentially conflicting if $\mathcal{G}_{l_1} \cap \mathcal{G}_{l_2} \neq \emptyset$, i.e., if the two libraries share at least one access path in their global writes.*

The first phase of ConflictJS reduces the search space of potential conflicts to be considered by the second phase of the approach. The first phase scales well to a large number of libraries because each library is analyzed in isolation. Comparing the global writes across libraries requires computing pairwise intersections of sets, which easily scales to a large number of sets. As mentioned in Section 5.2.5, the analysis might miss potential conflicts, e.g., because a library might perform a global write after the library has been loaded. A manual inspection of a subset of libraries suggests this limitation to be negligible in practice, because libraries tend to initialize their APIs at load time.

5.3.2 *Precise Validation of Conflicts*

The second step of ConflictJS is to validate potential conflicts identified in the first step. At first, we motivate the need for this second step with an example. Then, we explain the details of the validation.

5.3.2.1 *Motivation for Validation*

Potentially conflicting libraries write to the same globally accessible memory location. This situation may or may not cause a client to suffer from a conflict as defined in Definition 5.1. For example, consider Figure 5.2, which shows code snippets from two potentially conflicting libraries, *JSLite.js* and *ext-core.js*. The global access path to which both libraries write is `Array.prototype.remove`. Both libraries

Table 5.3: Actions performed by the global-writes analysis.

Runtime event	Action	Example

Variable write $w = v$	If w is a global variable:
	• Add w to $\mathcal{G}$. Let $paths(v)$ be the access paths of all values reachable from v. For each $p_v \in paths(v)$, add p_v to $\mathcal{G}$.
		`(function() {`
		`  var x = {a: 23};`
		`  window.foo = x;`
		`})();`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{foo, foo.a\}$

Property write $x.p = v$	Let $paths(window)$ be the access paths of all globally reachable values. For each $p_w \in paths(window)$:
	• If p_w points to x:
	– Add $concat(p_w, p)$ to $\mathcal{G}$. Let $paths(v)$ be the access paths of all values reachable from v. For each $p_v \in paths(v)$, add $concat(p_w, p_v)$ to $\mathcal{G}$.
		`(function() {`
		`  var x = window.Array;`
		`  x.p = {b: 42};`
		`  var y = {};`
		`  y.q = 5;`
		`})();`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{Array.p, Array.p.b\}$

Declaration of function f	If the global variable f points to the declared function (i.e., the function is globally declared), add f to $\mathcal{G}$.	`(function() {`
		`  function foo() {}`
		`})();`
		`function bar() {}`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{bar\}$

```
/* JSLite.js */
Array.prototype.remove = function(t) {
  var n = this.indexOf(t);
  return n > -1 && this.splice(n, 1),
    this
}
```

```
/* ext-core.js */
Array.prototype.remove = function(e) {
  var t = this.indexOf(e);
  return -1 != t && this.splice(t, 1),
    this
}
```

Figure 5.2: Example to show the need for validating potential conflicts.

extend the built-in `Array` object by adding a new method `remove`, which can be called with one argument. Even though the two methods are syntactically different, close inspection shows that both pieces of code are functionally equivalent. This example illustrates that reporting all potential conflicts would cause false positives because for some potential conflicts, all clients are guaranteed to observe the same behavior, irrespective of the library configuration.

5.3.2.2 *Synthesizing Clients and Comparing their Behavior*

To check whether a potential conflict between two libraries is indeed a conflict, ConflictJS synthesizes library clients and checks whether their runtime behavior differs depending on the library configuration. The basic idea is to consider each of the six scenarios listed in Definition 5.1 by comparing the behavior of two clients with each other. The two clients contain exactly the same code, except that they run with different library configurations. If ConflictJS observes a behavioral difference between the two clients, the potential conflict between the two libraries is indeed a conflict.

For illustration, consider the behavior conflict illustrated in Table 5.1. Our approach tries to validate this conflict by synthesizing clients, such as the client shown in the table. The approach compares the behavior of this client with different library configurations. For the example, ConflictJS finds that on calling `Class`, there is one library that throws an exception while the other does not. That is, the approach has validated the conflict and reports it, along with the synthesized client that illustrates the conflict.

Algorithm 5.1 summarizes our approach for validating potential conflicts by synthesizing and dynamically executing library clients. The main idea is to compare the execution of a client c with different library configurations, i.e., c_{l1}, c_{l2}, $c_{l1,l2}$, and $c_{l2,l1}$, as summarized in function *conflictingConfigs*. If there are multiple different behaviors,

Algorithm 5.1 Validate potential conflicts

Input: Libraries $l1, l2$ that both write to global access path p
Output: Validated conflict between $l1$ and $l2$

1: $c_{empty} \leftarrow$ empty client
2: **if** $conflictingConfigs(c_{empty})$ **then return** "inclusion conflict"
3: $c_{types} \leftarrow$ synthesize client that checks type of p
4: **if** $conflictingConfigs(c_{types})$ **then return** "type conflict"
5: **if** type of p is non-function **then**
6: $\quad c_{values} \leftarrow$ synthesize client that checks value of p
7: $\quad$ **if** $conflictingConfigs(c_{values})$ **then return** "value conflict"
8: **else**
9: $\quad \mathcal{C}_{behavior} \leftarrow$ synthesize clients that call function p
10: $\quad$ **for each** $c_{behavior} \in \mathcal{C}_{behavior}$ **do**
11: $\quad\quad$ **if** $conflictingConfigs(c_{behavior})$ **then return** "behavior conflict"
12: **function** $conflictingConfigs(c)$
13: $\quad \mathcal{B} \leftarrow \emptyset$ ▷ Set of observed runtime behaviors
14: $\quad$ **for each** $config \in \{l1, l2, l1l2, l2l1\}$ **do**
15: $\quad\quad b_{config} \leftarrow$ execute c_{config}
16: $\quad\quad \mathcal{B} \leftarrow \mathcal{B} \cup \{b_{config}\}$
17: $\quad$ **if** $|\mathcal{B}| > 1$ **then return** true
18: $\quad$ **else return** false

then the algorithm has validated a conflict. The following describes how ConflictJS creates clients to detect the four kinds of conflicts presented in Section 5.2.2.

5.3.2.3 *Checking for Inclusion Conflicts*

At first, ConflictJS checks for inclusion conflicts (lines 1 to 2). An inclusion conflict is triggered by simply including libraries, i.e., the client is an empty client that does not contain any statements. To compare library configurations, the behavior b_{config} indicates whether including libraries causes the client to throw an exception. If one library configuration causes an exception, whereas another configuration does not, then ConflictJS reports an inclusion conflict.

For the inclusion conflict example of Table 5.1, ConflictJS reports a conflict because trying to execute the empty client after loading

curl.js and *dojo.js* causes an exception, whereas executing the empty client after loading only one of these libraries does not throw any exception.

5.3.2.4 *Checking for Type Conflicts*

For any pair of libraries $l1, l2$ and shared global access path p for which the approach has not validated an inclusion conflict, the next step is to check for type conflicts. To this end, ConflictJS synthesizes a client that reads the value at the access path p and then checks its type (lines 3 to 4). The approach again executes this client with all possible library configurations and summarizes the behavior of each configuration as the type of the access path p. If one library configuration causes the client to see type t_1, whereas another library configuration causes the client to see type $t_2 \neq t_1$, then ConflictJS reports a type conflict.

An example of a library pair with a type conflict is given in the second column of Table 5.1. The approach reports this conflict because `push` is an array when loading one library but a function when loading the other library. The "client" cell of the table shows a client that suffers from this type conflict because the conflict causes the client to crash when it tries to call a function that turns out to be an array.

5.3.2.5 *Checking for Value Conflicts*

While checking for type conflicts, the analysis gathers information about the types of values stored at a global access path. For potential conflicts that are neither validated to be an inclusion conflict nor to be a type conflict, both libraries write values of the same type to the access path. Based on this type, ConflictJS checks for the remaining two kinds of conflicts. If the type is function, the approach compares the behavior of clients that call this function, as described below. If the type is a non-function, then the approach synthesizes a client that reads the value at the access path p (lines 6 to 7). To compare the behavior of this client across library configurations, ConflictJS compares the value read at p. The analysis directly compares primitive values and deeply compares objects. If different library configurations cause the client to read different values, then ConflictJS reports a value conflict.

The "value conflict" column of Table 5.2 gives an example of a type conflict on the `pako` access path. ConflictJS synthesizes a client that

extracts the number of properties of the value stored at `pako` and the then recursively extracts the values of these properties. The approach reports a conflict because the number of `pako`'s properties depends on whether the *pako* library or the *3Dmol* library is loaded.

5.3.2.6 *Checking for Behavior Conflicts*

The most challenging kind of conflict are behavior conflicts. These conflicts occur when different libraries write functions to the same global access path but the behaviors of these functions differ. In general, deciding whether the behavior of two functions differs is undecidable. ConflictJS approaches this problem by trying to synthesize clients that expose a difference in behavior. If the analysis succeeds in generating such a client within a fixed time budget, it reports a behavior conflict.

To synthesize clients we use a simple test generator inspired by Randoop's feedback-directed, random test generation [35]. Other test generation techniques, such as symbolic or concolic execution [4, 20, 38] or search-based test generation [74], could also be used for this step. Given a function-typed access path p defined by two libraries, the test generator starts by estimating the number n of arguments that the function expects. To this end, we use the `length` property of the function object at p, which in JavaScript yields the number of declared function parameters. This number is an estimate because a function body may also access additional arguments using the built-in `arguments` value. Next, to generate a call to the function, the test generator randomly decides on a random number ranging between 0 and n of arguments to pass. For each argument, the test generator decides on the type of argument to create by randomly choosing between the following types: *boolean*, *string*, *number*, *array*, *object*, *undefined* and *null*. To create a boolean, string, or number, the generator picks from a pre-defined pool of values. For arrays, the generator randomly picks a length ranging between 0 and 10 and fills it with random strings and numbers. Finally, to create an object, the generator creates up to 10 properties and assigns randomly generated values to them.

Once the arguments are generated, the function is called using the generated arguments. If and only if the call succeeds, without raising an exception, for at least one library configuration, the generator synthesizes a client that contains this call.

To compare the behavior of synthesized clients across library configurations, ConflictJS summarizes the behavior of the client execution

based on the return value of the function and based on whether the function raises an exception. The approach reports a behavior conflict in two cases: (i) if one library configuration causes the client to crash whereas another library configuration does not cause a crash, or (ii) if both configurations do not crash but the return value of the function at p differs.

For example, consider the last column of Table 5.2. ConflictJS synthesizes clients that call the function stored at the conflicting access path `Class`. The client shown in the table throws an exception for one of the two libraries but not for the other, which is why ConflictJS reports a behavior conflict.

5.4 IMPLEMENTATION

We implement ConflictJS as a client-server-based tool that analyzes JavaScript libraries. The client component synthesizes, executes, and analyzes clients in a browser, and sends a summary of the runtime behavior to the server. The server detects potential conflicts and validates them based on execution behavior gathered in the first and second phase, respectively. Our dynamic analyses to find global writes is build on top of Jalangi [108]. When synthesizing clients to detect behavior conflicts, we set the testing budget to 50 tests per access path. In this chapter, we implement the approach only for client side JavaScript libraries and it would be straightforward to adapt for server-side npm libraries but the problem is less severe for Node.js because there is a commonly accepted module system.

5.5 RESULTS AND DISCUSSION

We apply ConflictJS to 951 popular JavaScript libraries to evaluate the effectiveness of the approach in detecting library conflicts. We focus on the following research questions:

- How effective is ConflictJS in finding library conflicts and what kinds of conflicts occur in practice? (Section 5.5.2)
- What are the root causes of conflicts between libraries? (Section 5.5.3.1)
- Do library developers make an effort to avoid conflicting scenarios by following the "single API object" pattern? (Section 5.5.3.2)

Table 5.4: JavaScript libraries used for the evaluation.

	Min	Median	Max	Total
Libraries	-	-	-	951
Lines of code	9	574	275,0852	2,750,852
Size (bytes)	148	14,645	2,517,510	68,412,720

- What are the popular access paths that developers tend to choose? (Section 5.5.3.3)
- Is there a correlation between conflicts and the popularity of a library? (Section 5.5.3.4)
- How are the global writes and conflicts distributed across libraries and access paths, respectively? (Sections 5.5.3.5 and 5.5.3.6)

5.5.1 *Experimental Setup*

Our evaluation uses 951 real-world JavaScript libraries with a total of 2,750,852 lines of JavaScript code (Table 5.4). The libraries include the popular *jQuery*, *Underscore*, and *Dojo* projects, as well as various other highly popular libraries. We obtain these libraries by downloading them from the CDNJS content delivery network.[8] At the time of starting our experiments, the content delivery network offered a total of 2,095 libraries. We remove libraries that cannot be used in isolation in a standard desktop browser, e.g., because they rely on another library or because they target mobile devices. We heuristically check for such libraries by loading each library in isolation and filtering away all libraries that throw an exception. After filtering, 951 libraries remain, which is our benchmark set for the evaluation. To run our experiments, we use an Intel Core i7–4790 CPU machine clocked at 3.60GHz with 32 GB of memory, running Chrome 55, Node.js 6.9.1 on Ubuntu 16.04.

5.5.2 *Effectiveness in Finding Library Conflicts*

8 https://cdnjs.com/

5.5.2.1 *Potential Conflicts*

When analyzing the global writes of individual libraries, ConflictJS records writes to a total of 130,714 different access paths across the 951 libraries. Intersecting the global writes of libraries reveals that 4,121 of the access paths cause a potential conflict, i.e., at least two libraries write to each of these access paths. These conflicting writes are performed by 268 of the 951 libraries, i.e., roughly one out of four libraries is involved in a potential conflict.

5.5.2.2 *Validated Conflicts*

Out of the 268 potentially conflicting libraries, ConflictJS validates 166 as certainly conflicting by synthesizing a client whose behavior depends on the library configuration. The validated conflicts are due to 1,840 distinct access paths. In other words, ConflictJS successfully validates 62% of the potentially conflicting libraries (i.e., of 268 libraries) as certainly conflicting and finds a validated conflict in 17% of all libraries (i.e., of 951 libraries).

5.5.2.3 *Kinds of Validated Conflicts*

Figure 5.5 summarizes how prevalent the four kinds of conflicts are among all validated conflicts. The two sides of the figure provide different views on the same data. Figure 5.3 focuses on pairs of conflicting libraries and shows how many of these pairs are caused by the four kinds of conflicts. If a pair of libraries is involved in multiple kinds of conflicts, then this pair is shown at the set intersection. For example, there are four pairs of libraries that have a value conflict for a global access path and a behavior conflict for another global access path. Figure 5.4 shows the distribution among the four kinds of conflicts for individual libraries. Since a single library may be involved in conflicts with different libraries, these sets overlap. For example, there are seven libraries that are involved in at least one inclusion conflict, value conflict, and behavior conflict.

There are two main take-aways of these results. First, all four kinds of conflicts are prevalent in practice, which confirms our decisions to consider all four kinds in ConflictJS. Second, the majority of conflicts are non-inclusion conflicts, i.e., they do not cause an exception just after loading the conflicting libraries. Finding such conflicts and

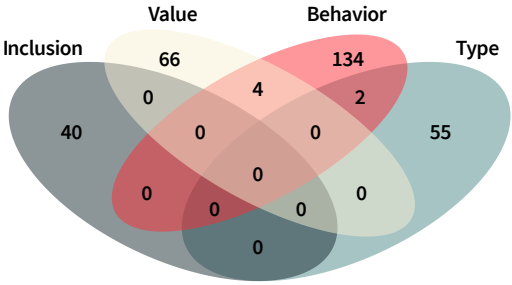


Figure 5.3: Library pairs classified by the type of conflict.

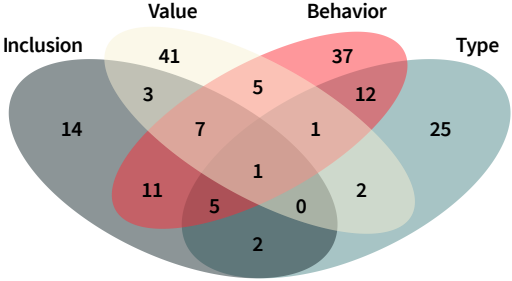


Figure 5.4: Libraries classified by the type of conflict.

Figure 5.5: Prevalence of the four kinds of validated conflicts. Note that the surface is not proportional to the numbers.

reasoning about them is challenging for both library developers and users alike.

5.5.3 *Empirical Study of Library Conflicts*

The large number of libraries considered and conflicts detected in our evaluation, enables us to learn more about how and why conflicts occur in JavaScript libraries. We discuss these findings in the following and discuss what impact they have on library developers, library users, and language designers.

5.5.3.1 *Root Causes of Conflicts*

To understand the root causes of conflicts between libraries we manually inspect a random sample of 25 conflicting libraries. During the

Table 5.5: Recurring patterns among the root causes of conflicts. All numbers are out of 25

Pattern	Description	Nb.	Example(s)
Independent implementations	Two libraries implement similar functionality and use the same global access path to store the function, but the behavior slightly differs.	5	*polymer* and *trix* both define `wrap` and `unwrap` functions. Other examples: Figure 5.1 and issue #434 of *es6-shim*.
Copied third-party code	Two libraries both copy code from a third party, e.g., another library. At least one version of the code is outdated.	5	*qooxdoo* includes an outdated copy of *sinon*. See issue #9277 of *qooxdoo*. Another example: Issue #1068 of *d3fc*.
Poor API usage	A library adds an event handler in a way that removes all other handlers for this kind of event, instead of adding to the existing event handlers.	4	*rxjs* and *gifshot* both write to `onmessage` to handle postMessage communication. Instead, they should use `addEventListener`, which allows multiple event handlers.
Convenient identifier	Two libraries use a convenient, global identifier for different purposes.	3	*mermaid*, a library for generating diagrams, writes to `_`, which is also used by *scorejs* and others. See issue #512 of *mermaid*.
Incorrect monkey patching	A library tries to extend a built-in API but accidentally removes existing functionality.	1	*PreloadJS* and *zingchart* both overwrite the built-in `JSON` in a way that destroys existing functionality. See issue #226 of *PreloadJS*.
Documented dependency	One library depends on another and documents this dependency.	4	*alloy-ui* is a framework built on top of *yui*. Clients should not be surprised by "conflicts" between them.
Fork	One library is derived from another library and modifies or extends the functionality of the original library.	3	*wysihtml* is an extended version of *wysihtml5*. Clients should never use both together.

manual inspection, we identified seven recurring patterns. Table 5.5 describes each pattern and and illustrates it with an example.

Five of the seven patterns, which account for 18 out of the 25 inspected conflicts, are unintended by the developers and likely to cause surprising behavior for library users. These patterns are shown in the upper part of Table 5.5. The patterns cover conflicts caused by independently developed variants of the same functionality, copied third-party code, poor API usage, repeated use of convenient global identifier name, and incorrect attempts to patch built-in JavaScript APIs. To double-check our intuition about whether conflicts are intended by the library developers, we reported seven conflicts to the developers of conflicting libraries. At the time of writing, four of our reports have been acknowledged and confirmed as worth fixing by the respective developers. Of the four acknowledged libraries, two have been fixed by the developers. Apart from this, based on our bug report, the developer of a library has reported a bug to another library with which it was conflicting. Subsequently, this bug report also got fixed.

For all of these five patterns, the root cause boils down to suboptimal decisions by library developers, such as programming errors or copy-and-paste of existing code. However, at least for some of them, the design of the JavaScript language and APIs may also be partially to blame. For example, instances of the "Poor API usage" pattern are caused by the fact that the JavaScript web APIs provide two orthogonal ways to attach event handlers: Setting the handler, e.g., `onmessage = ..`, which overwrites any already attached handler, and adding a handler via `addEventListener("message", ..)`, which preserves already attached handlers. The conflicts detected by ConflictJS are the result of libraries overwriting each other's event handlers by directly setting the handler. Another example is the "Incorrect monkey patching" pattern. The term "monkey patching" refers to extending built-in APIs of the JavaScript language, which is possible but non-trivial to implement without removing existing functionality.

The remaining two patterns, shown in the lower part of Table 5.5, both occur in a situation where library users are unlikely to be surprised by the conflict. One reason is that libraries depend on each other and document these dependencies clearly, so that library users know in which order to load them. Ideally, our experimental setup would filter such libraries, as we assume each library is supposed to be used independently. Another reason is that libraries provide the same or very similar overall functionality, so that library users would never include both together.

Overall, we draw two conclusion from our manual inspection. First, most conflicts reported by ConflictJS are programming errors that should be fixed by library developers to prevent clients from surprising behavior. Second, the root causes of conflicts are diverse but can be classified into a set of recurring patterns. Knowing these patterns may become the basis of guidelines for library developers what mistakes to avoid. Furthermore, the patterns can guide the design of future program repair techniques that fix conflicting code.

5.5.3.2 *The "Single API Object" Pattern*

The "single API object" pattern (Section 5.2.1) allows developers to avoid conflicts by storing all globally accessible data into a single object named like the library. If all libraries follow this pattern, no conflicts occur. To understand whether libraries follow this pattern, we check for each library whether for all writes to a global access path, the path begins with a segment that matches the name of the library, as listed in the CDNJS content delivery network. When matching an access path and a library name, we omit the *.js* suffix that some libraries use.

We find that 273 out of the 951 libraries follow the "single API object" pattern. While promising, this means that 71% of all libraries do not follow the pattern, but instead use the shared global namespace in a possibly conflicting way. We conclude that relying on developer discipline in an open environment, such as the JavaScript library ecosystem, is insufficient to enforce a conflict-avoiding policy.

5.5.3.3 *Popular Global Access Paths*

Table 5.6: Popularity of global access paths (measured in the number of libraries that write to an access path).

Libs.	Global access paths
13	$
12	localStorage.debug
10	requestAnimationFrame
9	_, jQuery, onload, require
8	clearImmediate, Promise, __core-js_shared__, __core-js_shared__.wks, setImmediate, __core-js_shared__.wks.iterator, __core-js_shared__.wks.toStringTag

The large number of potential conflicts detected by ConflictJS raises the question what global access paths are particularly popular among library developers. Table 5.6 lists the most popular global access paths along with the number of libraries that write to it. Perhaps unsurprisingly, the most popular access path is the dollar sign, `$`, which is a legal identifier name in JavaScript and used by several libraries, including *jQuery* to export their API. Another popular choice is the underscore sign, `_`, which is shared, e.g., by the *Underscore* and *Lodash* libraries. Choosing a short identifier name to export an API is tempting for library developers and potentially convenient for library users. However, the downside is that multiple libraries may (either knowingly or not) pick the same short identifier name, which likely causes surprises if these libraries are used together.

5.5.3.4 *Conflicts Versus Library Popularity*

To better understand to what extent library conflicts depend on a library's popularity, Figure 5.6 shows for each library how many stars it has and in how many conflicts it is involved. Each data point corresponds to one library. For example, one library that has 45,901 starts is involved in two conflicts. Overall, the figure shows that most conflicts are due to libraries with less than 10,000 stars. The main reason is that only few libraries have more than 10,000 stars, as illustrated in Figure 5.7. This figure shows how the libraries validated to be conflicting from our benchmark are distributed across the popularity measure. Both figures look similar, which explains the distribution of conflicts across popularity. At the same time, it is interesting to note that even some highly popular libraries are involved in conflicts, as indicated by the data points on the right end of Figure 5.6.

5.5.3.5 *Distribution of Global Writes Across Libraries*

To better understand the large number of 130,714 global writes performed by the 951 libraries, we analyze how these writes are distributed across the libraries. The results show a highly skewed distribution, with a few libraries writing to many global access paths but with a median of only one global write. The libraries that write to most global access paths are large and popular libraries, such as Amazon's AWS JDK (36,049 access paths) and Microsoft's implementation of TypeScript (5,678 global writes). Their high number of global

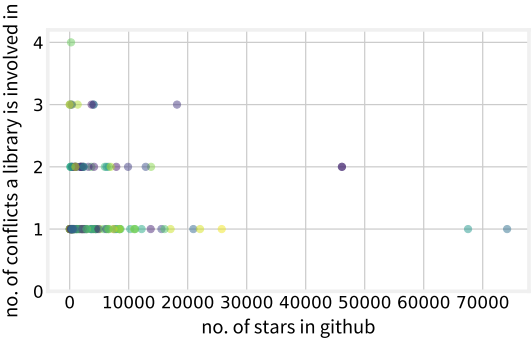


Figure 5.6: Number of conflicts each library is involved in.

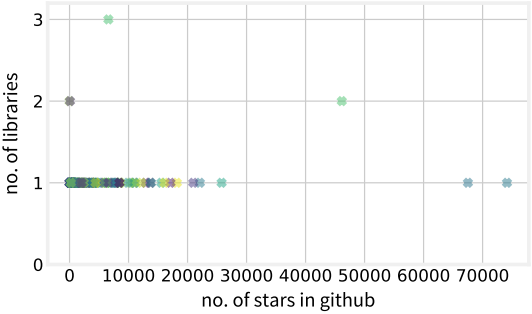


Figure 5.7: Popularity measure of libraries validated to be conflicting

Figure 5.8: Influence of popularity on number of conflicts and number of libraries. Each data point represents one library. (twelve out of 166 libraries do not have a Github repository and hence are not included here)

writes does not imply bad coding practice. For example, the many access paths written by the AWS SDK library almost all start with `AWS.`, i.e., they follow the "single API object" pattern. We conclude that judging libraries based on their total number of global writes, which might have been a simple alternative to ConflictJS, is not an effective way to find conflict-triggering libraries.

5.5.3.6 *Distribution of Conflicts Across Access Paths*

A reader may wonder how many libraries write to the same global access path. Investigating this question yields a long-tail distribution: Most global access paths (3,836) are contended for by only two libraries, but a large number of highly popular global access paths is written to by up to 13 libraries. We conclude that preventing library developers from using a few highly contended access paths, such as `$` and `_`, is insufficient to solve the problem of library conflicts, because there are many other access paths that cause conflicts.

5.6 CONCLUSION

JavaScript code, including independently developed libraries, shares the same global namespace. Because the most widely used versions of the language lack features designed for encapsulating exported APIs, library developers risk to accidentally share the same globally accessible memory locations and write different data and functions to them. This chapter defines and classifies such library conflicts, presents an automatic and scalable approach to detect them, and studies conflicts in a large set of libraries. We deal with the huge search space of possible conflicts through a two-phase approach that dynamically analyzes a corpus of libraries in isolation to detect potential conflicts and then synthesizes library clients to validate conflicts. We empirically study how and why conflicts occur, showing that a diverse set of programming errors in libraries are the primary root cause.

Our work not only provides a practical tool for library developers to detect conflicts and for library users to avoid conflicting libraries, but also highlights the importance of language features for encapsulating independently developed code. We believe that our work provides ample opportunities for future work. One direction is to complement our precise but unsound analysis with a sound (and likely imprecise) checker for library conflicts. To help developers avoid conflicts, another line of future work are repair tools that either address the coding errors that cause conflicts. Finally, future work could develop automatic code transformations to help libraries use encapsulation mechanisms provided in recent and future versions of JavaScript.

6

LEARNING FROM RUNTIME BEHAVIOR TO FIND NAME-VALUE INCONSISTENCIES

The current chapter presents another corpus-based dynamic analysis approach. In comparison to the previous chapter (Chapter 5), where we use JavaScript libraries, the current chapter dynamically analyzes Python files. The analysis yields variable names and the assigned values at runtime. Such name-value pairs are useful in training neural classifiers that can uncover inconsistencies.

6.1 MOTIVATION

Identifier names are a means to convey the (intended) semantics of code. Because using meaningful identifier names is crucial for the understandability and maintainability of code, developers strive for names that express the value or behavior a name is bound to. Hindle et al.[83] empirically show that source code is "natural", in the sense that it follows conventions and has regularities similar to natural language. Various name-based program analyses exploit this naturalness, e.g., to predict types [218, 240, 248, 263, 267], to detect bugs [226], or to restore meaningful names in obfuscated or minified code [144, 201, 211].

Sometimes, an identifier name and the value that the name refers to do not match. One possible reason is a *misleading name* that is bound to a correct value. Because such names make code unnecessarily hard to understand and maintain, developers may want to replace them with more meaningful names. Another possible reason is that a correct name refers to an *incorrect value*. Because such values may propagate through the program and cause unexpected behavior, developers should fix the corresponding code to use the correct value. We refer to both of these cases as *name-value inconsistencies*.

The following illustrates the problem with two motivating examples, both of which we found during our evaluation in a corpus of real-world computational notebooks written in Python [228]. As an example of a misleading name, consider the following code:

```
log_file = glob.glob('/var/www/some_file.csv')
```

The right-hand side of the assignment yields a list of file names, which is inconsistent with the name of the variable it gets assigned to, because `log_file` suggests a single file name. The code is even more confusing since this specific call to `glob` will return a list with at most one file name. That is, a cursory reader of the code may incorrectly assume this file name to be stored in the `log_file` variable, whereas it is actually wrapped into a list. To clarify the meaning of the variable, it could be named, e.g., `log_files` or `log_file_list`.

As an example of a name-value inconsistency caused by an incorrect value, consider the following code:

```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

The code tries to divide a dataset into training and test sets. Names like `train_size` are usually bound to non-negative integer values. However, the above code assigns the value 135.0 to the `train_size` variable, i.e., a floating point value. Unfortunately, this value causes the code to crash at the last line, where `train_size` is used as an index to slice the dataset, but indices for slicing must be integers. While the root cause and the manifestation of the crash are close to each other in this simple example, in general, incorrect values may propagate through a program and cause hard to understand misbehavior.

Finding name-value inconsistencies is difficult because it requires both understanding the meaning of names and realizing that a value that occurs at runtime does not match the usual meaning of a name. As a result, name-value inconsistencies so far are found mostly during some manual activity. For example, a developer may point out a misleading name during code review, or a developer may stumble across an incorrect value during debugging. Because developer time is precious, tool support for finding name-value inconsistencies is highly desirable.

This chapter presents Nalin, an approach for detecting name-value inconsistencies automatically. The approach combines dynamic program analysis with deep learning. On the hand one, a dynamic analysis keeps track of assignments during an execution and gathers pairs

of names and values the names are bound to. On the other hand, a neural model predicts whether a name and a value fit together. When the dynamic analysis observes a name-value pair that the neural model predicts to not fit together, then the approach reports a warning about a likely name-value inconsistency.

While simple at its core, realizing the Nalin idea involves four key challenges:

C1 Understanding the semantics of names and how developers typically use them. The approach addresses this challenge through a learned token embedding that represents semantic similarities of identifiers in a vector space. For example, the embedding maps the names `train_size`, `size`, and `len` to similar vectors, as they refer to similar concepts.

C2 Understanding the meaning of values and how developers typically use them. The approach addresses this challenge by recording runtime values and by encoding them into a vector representation based on several properties of values. The properties include a string representation of the value, its type, and type-specific features, such as the shape of multi-dimensional numeric values.

C3 Pinpointing unusual name-value pairs. We formulate this problem as a binary classification task and train a neural model that predicts whether a name and a value match. To the best of our knowledge, this work is the first to detect coding issues through neural classification over runtime values.

C4 Obtaining a dataset for training an effective model. The approach addresses this challenge by considering observed name-value pairs as correct examples, and by creating incorrect examples by combining names and values through a statistical sampling that is likely to yield an incorrect pair.

Our work relates to two closely related streams of work. First, Nalin resembles existing learning-based bug detection approaches [209, 226, 238, 252, 266], which also train a model to classify code as correct or incorrect. Our work detects problems exposed during an execution, whereas prior work is based on static analysis, and we are the first to focus on name-value inconsistencies, whereas prior work targets other kinds of problems. Second, Nalin relates to previous work on

learning from runtime behavior [265]. Instead of detecting name-value inconsistencies, their work aims at finding a meaningful vector representation of a piece of code, which can then be used to predict what coding problem the code solves and what strategy it adopts.

We train Nalin on 780k name-value pairs and evaluate it on 10k previously unseen examples from real-world Python code. The model effectively distinguishes consistent from inconsistent examples, with an F1 score of 0.87. Manually inspecting a sample of warnings raised in real-world code shows that Nalin finds true positives with a precision of 51%. We also show that the approach complements state-of-the-art static analysis-based tools that warn about frequently made mistakes, type-related issues, and name-related bugs.

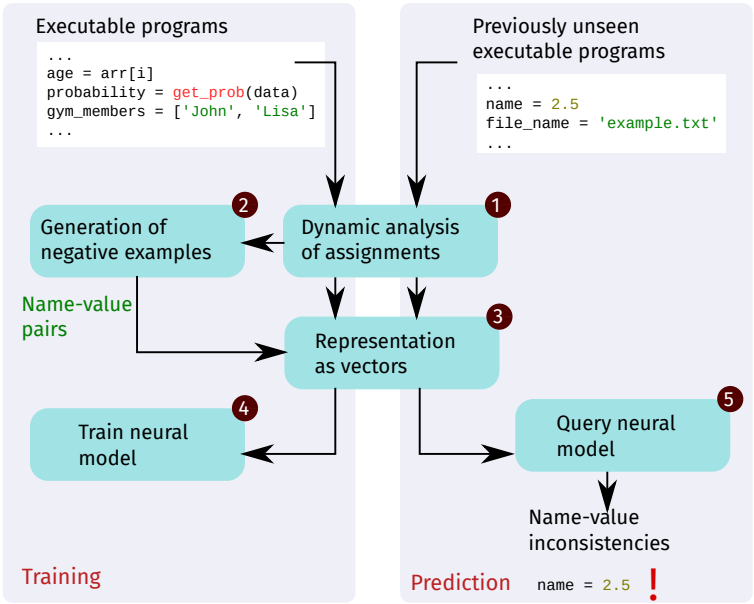
In summary, this chapter contributes the following:

- An automatic technique to detect name-value inconsistencies.
- The first approach to find coding issues through machine learning on runtime behavior.
- A type-guided generation of negative examples that improves upon a purely random approach.
- Empirical evidence of the effectiveness of the approach that finds name-value inconsistencies in real-world code with a reasonable precision.

6.2 OVERVIEW

This section describes the problem we address and gives an overview of our approach. Nalin reasons about *name-value pairs*, i.e., pairs of an identifier name and a value that gets assigned to the identifier in a program. The problem we address is to identify name-value pairs where the name is not a good fit for the value, which we call *inconsistent name-value pairs*. Identifying such pairs is an inherently fuzzy problem: Whether a name fits a value depends on the conventions that programmers follow when naming variables that refer to particular kinds of values. The fuzziness of the problem motivates a data-driven approach [262], where we use the vast amounts of available programs as guidance for what name-value pairs are common and what name-value pairs stand out as inconsistent.

Broadly speaking, Nalin consists of five components and two phases, illustrated in Figure 6.1. During the training phase, the ap-



Name	Value	Type	Length	Shape
age	23	int	null	null
probability	0.83	float	null	null
gym_members	['John', 'Lisa']	list	2	null
name	2.5	float	null	null
file_name	'example.txt'	str	11	null

Figure 6.1: Overview of the approach.

proach learns from a corpus of executable programs a neural classification model, which then serves during the prediction phase for identifying name-value inconsistencies in previously unseen programs. The following illustrates the five components of the approach with some examples. A detailed description follows in Section 6.3.

Given a corpus of executable programs, the first component is a dynamic analysis of the assignments of values to names that occur during executions of the programs. For each assignment, the analysis extracts the variable name, the value assigned to the variable, and several properties of the value, e.g., the type, length, and shape. As illustrated in Figure 6.1, properties that do not exist for a particular value are represented by *null*. For example, the analysis extracts the length of the assigned value for `gym_members`, but not for `age` and `probability`, as the corresponding values are primitives that do not have a length.

While the name-value pairs obtained by the dynamic analysis serve as positive examples, the second component generates negative examples that combine names and values in an unusual and likely inconsistent way. The motivation behind generating negative examples is that Nalin trains a classification model in a supervised manner, i.e., the approach requires examples of both consistent name-value pairs and inconsistent name-value pairs. Using the example pairs in Figure 6.1, one negative example would be the name `gym_members` paired with the floating point value 0.83, which indeed is an unusual name-value pair. Our approach for generating negative examples is a probabilistic algorithm which considers the type of values that are usually observed with a name to bias the selection of unusual values toward unusual types. The first and second component together address challenge C4 from the introduction, i.e., obtaining a dataset for training an effective model.

The third component of Nalin addresses challenges C1 and C2, i.e., "understanding" the semantics of names and values. To this end, the approach represents names and values as vectors that preserve their meaning. To represent identifier names, we build on learned token embeddings [182], which map each name into a vector while preserving the semantic similarities of names [271]. For example, the vector of `probability` will be close to the vectors of names `probab` and `likelihood`, because these names refer to similar concepts. To represent values, we present a neural encoding of values based on their string representation, type, and other properties.

Given the vector representations of name-value pairs, the fourth component trains a neural model to distinguish positive from negative examples. The result is a classifier that, once trained with sufficiently many examples, addresses challenge C3. The fifth component of the approach queries the classifier with vector representations of name-value pairs extracted from previously unseen programs. For the two new assignments shown in Figure 6.1, the trained classifier will correctly identify the assignment `name = 2.5` as unusual and raises a warning about an inconsistent name-value pair.

6.3 APPROACH

The following presents the components of Nalin outlined in the previous section in more detail.

6.3.1 *Dynamic Analysis of Assignments*

The goal of this component is to gather name-value pairs from a corpus of programs. Our analysis focuses on assignments because they associate a value with the name of, e.g., a local variable. One option would be to statically analyze all assignment in a program. However, a static analysis could capture only those values where the right-hand side of an assignment is a literal, but would miss many other assignments, e.g., when the right-hand side is a complex expression or function call. In the code corpus used in our evaluation, we find that 90% of all assignments have a value other than a primitive literal on the right-hand side, i.e., a static analysis could not gather name-value pairs from them. Instead of statically analyzing assignments, Nalin uses a dynamic analysis that observes all assignments during the execution of a program. Besides the benefit of capturing assignments that are hard to reason about statically, a dynamic analysis can easily extract additional properties of values, such as the length or shape, which we find to be useful for training an effective model.

6.3.1.1 *Instrumentation and Data Gathering*

To dynamically analyze assignments, Nalin instruments and then executes the programs in the corpus. For instrumentation, the analysis traverses the abstract syntax tree of a program and augments assignments with a call to a function that records the name and value

of the assignment. We focus on assignments where the left-hand side a single identifier, e.g., `a = foo()`, and ignore cases where the left-hand side is a more complex access path, e.g., `data.row = val` or `user_info["age"] = input`. The reasoning for excluding assignments to access paths is that it is not always clear which name to consider. In the given examples, it is not clear if `val` should be associated with `data` or `row`, and likewise if the value of `input` should be associated with `user_info` or `'age'`.

For each assignment encountered at runtime, in addition to the name of the identifier on the left-hand side and the assigned value, we extract additional properties about the assignment. Slightly abusing the term "pair" to also include these additional properties, the analysis extracts the following information:

Definition 6.1 (Name-value pair)**.** *A name-value pair is a tuple* (n, v, τ, l, s)*, where* n *denotes the variable name on the left hand side,* v *is a string representation of the value assigned to the variable,* τ *represents the type of the variable, and* l, s *represents the length and shape of the value, respectively.*

Length here refers to the number of items present in a collection or sequence type value. Some data types can be multidimensional, and *shape* refers to the number of items present in each dimension. For values without a length or shape, the corresponding entries in the tuple contains `null`. The table in Figure 6.1 shows examples of name-value pairs gathered by the analysis. Extending the approach to gather additional properties of values is straightforward. As we show in the evaluation, the additional properties overall increase the effectiveness of the model, but also encode partially redundant information, e.g., because the type of a value is implicitly encoded in its string representation.

6.3.1.2 *Filtering and Processing of Gathered Data*

Some of the gathered name-value pairs contain very little information, while some others may be too detailed to learn an effective model. To reduce noise in the training data and to increase the model's ability to generalize across similar names and values, Nalin filters and processes the gathered data in three ways.

TRUNCATE LARGE VALUES Large values are problematic for two reasons. First, they consume lots of memory, make the overall approach inefficient. For example, recording a large, multi-dimensional

array can easily fill hundreds of megabytes. Second, to capture the meaning of the value and to decide whether it is consistent with name, a small subset of large values is often sufficient. Nalin handles large values by truncating them to a maximum size. For values that have a size, e.g., strings or lists, we keep up to 100 items, namely up to 50 from the beginning plus up to 50 from the end. For other values, we obtain a string representation of the value and then truncate the string to at most 50 plus 50 characters.

MERGE TYPES We observe that the gathered data forms a long-tailed distribution of types. One of the reasons is the presence of many similar types, such as Python's dictionary type `dict` and its subclass `defaultdict`. To help the model generalize across similar types, we reduce the overall number of types by merging some of the less frequent types. Specifically, we first choose the ten most frequent types present in the dataset. For the remaining types, we replace any types that is in a subclass relationship with one of the frequent types by the frequent type. For example, consider a name-value pair *(stopwords, frozenset({"all", "afterwards", "eleven", ...}), frozenset, 337, null)*. Assuming that type *frozenset* is not among the ten most frequent types, but type `set` is, we change the name-value pair into *(stopwords, frozenset({"all", "afterwards", "eleven", ...}), set, 337, null)*.

FILTER MEANINGLESS NAMES An underlying assumption of Nalin is that developers use meaningful variable names. Unfortunately, some names are rather cryptic, such as variables called `a` or `ts_pd`. Such names help neither our model nor developers in deciding whether a name fits the value it refers to, and hence, we filter likely meaningless names. The first type of filtering considers the length of the variable names and discards any name-value pairs where the name is less than three characters long. The second type of filtering is similar to the first one, except that it targets names composed of multiple subtokens, such as `ts_pd`. We split names at underscores[1], and remove any name-value pairs where each subtoken has less than three characters.

1 https://www.python.org/dev/peps/pep-0008/#function-and-variable-names

6.3.2 *Generation of Negative Examples*

The gathered name-value pairs provide numerous examples of names and values that developers typically combine. Nalin uses supervised learning to train a classification model that distinguishes consistent, or positive, name-value pairs from inconsistent, or negative, pairs. Based on the common assumption that most code is correct, we consider the name-value pairs extracted from executions as positive examples. The following presents two techniques for generating negatives examples. First, we explain a purely random technique, followed by a type-guided technique that we find to yield a more effective training dataset.

6.3.2.1 *Purely Random Generation*

Our purely random algorithm for generating negative examples is straightforward. For each name-value pair (n, v, τ, l, s), the algorithm randomly selects another name-value pair (n', v', τ', l', s') from the dataset. Then, the algorithm creates a new negative example by combining the name of the original pair and the value of the randomly selected pair, which yields (n, v', τ', l', s').

While simple, the purely random generation of negative examples suffers from the problem of creating many name-value pairs that do fit well together. The underlying root cause is that the distribution of values and types is long-tailed, i.e., the dataset contains many examples of similar values among the most common types. For example, consider a name-value pair gathered from an assignment `num = 23`. When creating a negative example, the purely random algorithm may choose a value gathered from another assignment `num = 3`. As both values are positive integers, they both fit the name `num`, i.e., the supposedly negative example actually is a legitimate name-value pair. Having many such legitimate, negative examples in the training data make it difficult for a classifier to discriminate between consistent and inconsistent name-value pairs.

6.3.2.2 *Type-Guided Generation*

To mitigate the problem of legitimate, negative examples that the purely random generation algorithm suffers from, we present a type-guided algorithm for creating negative examples. The basic idea is to first select a type that a name is infrequently observed with, and to

Algorithm 6.1 Create a negative example

Input: Name-value pair (n, v, τ, l, s), dataset D of all pairs
Output: Negative example (n, v', τ', l', s')

1: $F_{global} \leftarrow$ Compute from D a map from types to their frequency
2: $F_{name} \leftarrow$ Compute from D and n a map from types observed with n to their frequency
3: $T_{name} \leftarrow \emptyset$ ▷ Types seen with n
4: $T_{name\_infreq} \leftarrow \emptyset$ ▷ Types infrequently seen with n
5: **for each** $(\overline{\tau} \mapsto f) \in F_{name}$ **do**
6: $\quad T_{name} \leftarrow \overline{\tau}$
7: $\quad$ **if** $f \leq$ threshold **then**
8: $\quad\quad T_{name\_infreq} \leftarrow \overline{\tau}$
9: $T_{all} \leftarrow dom(F_{global})$ ▷ All types ever seen
10: $T_{cand} = (T_{all} \setminus T_{name}) \bigcup T_{name\_infreq}$ ▷ Types never or infrequently seen with n
11: $\tau' \leftarrow weightedRandomChoice(T_{cand}, F_{global})$
12: $v', l', s' \leftarrow randomChoice(D, \tau')$
13: **return** (n, v', τ', l', s')

then select a random value among those observed with the selected type. Algorithm 6.1 shows the type-guided technique for creating a negative example for a given name-value pair. The inputs to the algorithm are a name-value pair (n, v, τ, l, s) and the complete dataset D of positive name-value pairs.

The first two lines of Algorithm 6.1 create two helper maps, which map types to their frequency. The F_{global} map assigns each type to its frequency across the entire dataset D, whereas the F_{name} map assigns each type to how often it occurs with the name n of the positive example. Next, lines 3 to 8 populate two sets of types. The first set, T_{name}, is populated with all types ever observed with name n. The second set, T_{name}, is populated with all types that are infrequently observed with name n. "Infrequent" here means that the frequency of the type among all name-value tuples with name n is below some threshold. We set the threshold to be 3% of all positive name-value pairs with name n. The goal of selecting types that are infrequent for a particular name is to create negative examples that are unusual, and hence, likely to be inconsistent.

The remainder of the algorithm (lines 9 to 13) picks a type to be used for the negative example and then creates a negative name-value

◇ Given name-value pair:

```
      years  = [2011, 2012, 2013, 2014]
(n, v, τ, l, s) = (years, [2011, 2012, 2013, 2014], list, 4, null)
```

◇ All types years has been in the dataset and their frequencies:

```
F_name = { list: 235, ndarray: 59, int: 33,
           float: 7, dict: 5, tuple:4, set:1 }
```

◇ Infrequent types for *years*:

```
T_name_infreq = {float, dict, tuple, set}
```

◇ Global frequencies of types infrequently or never seen with *years*:

```
F_global = { str: 89337, bool: 5385,
             float: 71244, dict: 21654,  ... }
```

◇ Weighted random selection of a target type:

```
τ' = float
```

◇ Random selection of a `float` value from the dataset:

```
(n, v', τ', l', s') = (years, 1.8, float, null, null)
```

Figure 6.2: Steps for creating a negative example.

pair by combining the name n with a value of that type. To this end, the algorithm computes all candidates types, T_{cand}, that are either never observed with name n or among the types T_{name_infreq} that infrequently occur with n. The algorithm then randomly selects among the candidate types, using the global type frequency as weights for the random selection. The rationale is to choose a type that is unlikely for the name n, while following the overall distribution of types. The latter is necessary to prevent the model from simply learning to spot unlikely types, but to instead learn to find unlikely combinations of names and values. Once the target type τ' for the negative example is selected, the algorithm randomly picks a value among all values (line 12) observed with type τ', and eventually return a negative example that combines name n with the selected value.

Figure 6.2 illustrates the algorithm with an example from our evaluation. The goal is to create a negative example for a name-value pair where the name is `years`. In the dataset of positive examples, the name `years` occurs with values of types `list`, `ndarray`, `int`, etc., with the frequencies shown in the figure. For example, `years` occurs 235 times with a `list` value, but only seven times with a `float` value. Among

all types that occur in the dataset, many never occur together with the name `year`, e.g., `str` and `bool`. Based on the global frequencies of types that `year` never or only infrequently occurs with, the algorithm picks `float` as the target type. Finally, a corresponding `float` value is selected from the dataset, which is *1.8* for the example, and the negative example shown at the bottom of the figure is returned.

By default, Nalin uses the type-guided generation of negative examples, and our evaluation compares it with the purely random technique. The generated negative examples are combined with the positive examples in the dataset, and the joint dataset serves as training data for the neural classifier.

6.3.3 *Representation as Vectors*

Given a dataset of name-value pairs, each labeled either as a positive or a negative example, Nalin trains a neural classification model to distinguish the two kinds of examples. A crucial step is to represent the information in a name-value pair as vector, which we explain in the following. The approach first represents each of the five components (n, v, τ, l, s) of a name-value pair as a vector, an then feeds the concatenation of these vectors into the classifier. Figure 6.3 shows an overview of the neural architecture. The following describes the vector representation in more detail, followed by a description of the classifier in Section 6.3.4.

REPRESENTING VARIABLE NAMES To enable Nalin to reason about the meaning of variable names, it maps each name into a vector representation that encodes the semantics of the name. For example, the representation should map the names `list_of_numbers` and `integers` to similar vectors, as both represent similar concepts, but the vector representations of the names `age` and `file_name` should differ from the previous vectors. To this end, our approach builds on word embeddings, which is a learned function that assigns a vector to every name. Originally proposed in natural language processing as a means to represents words [101, 182], word embeddings are becoming increasingly popular also on source code [195, 210, 223, 226, 240], where they represent individual tokens, e.g., variable names.

Specifically, we build upon FastText [182], a neural word embedding shown to represent the semantics of identifiers more accurately than other popular embeddings [271]. A key benefit of FastText is

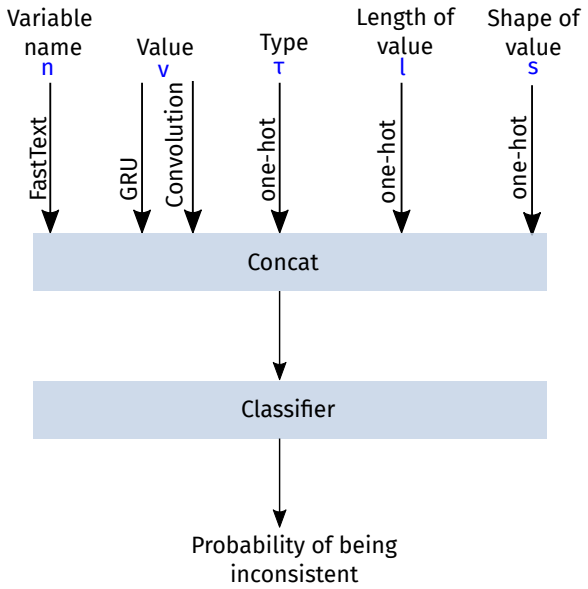


Figure 6.3: The architecture of the model.

to avoid the out-of-vocabulary problem that other embeddings, e.g., Word2vec [101] suffer from, by splitting each token into n-grams and by computing a separate vector representation for each n-gram. We pre-train a FastText model on a corpus of token sequences extracted from Python programs (Section 6.5.1). Formally, the trained FastText model M, assigns to each name n a real-valued vector $M(n) \in \mathbb{R}^d$, where $d = 100$ in our evaluation.

REPRESENTING VALUES The key challenge for representing the string representations of values as vectors is that there is a wide range of different values, including sequential structures, e.g., in values of types *string*, *ndarray*, *list*, and values without an obvious sequential structure, e.g., primitives and custom objects. The string representations of values may capture many interesting properties, including and beyond the information conveyed by the type of a value. For example, the string representation of an `int` implicitly encodes whether the value is a positive or negative number. Our goal when representing values as vector is to pick up such intricacies, without manually defining type-specific vector encoders.

To this end, Nalin represents value as a combination of two vector representations, each computed by a neural model that we jointly learn along with the overall classification model. On the one hand, we use a recurrent neural network (RNN) suitable for capturing sequential structures. Specifically, we apply gated recurrent units (GRU) over the sequence of characters, where each character is used as an input at every timestep. The vector obtained from the hidden state of the last timestep then serves as the representation of the complete sequence. On the other hand, we use a convolutional neural network (CNN) suitable for capturing non-sequential information about the value. Specifically, the approach applies a one-dimensional CNN over the sequence of characters, where the number of channels for the CNN is equal to the number of characters in the string representation of the value, the number of output channels is set to 100, Relu is the activation function, and a one-dimensional MaxPool layer serves as the final layer. Finally, Nalin concatenates the vectors obtained from the RNN and the CNN into the overall vector representation of the value.

REPRESENTING TYPES To represent the type of a value as a vector, the approach computes a one-hot vector for each type. Each vector has a dimension equal to the number of types present in the dataset. A type is represented by setting an element to one while keeping the remaining elements set to zero. For example, if we have only three types namely *int*, *float*, and *list* in our dataset then using one-hot encoding, each of them can be represented as *[1, 0, 0]*, *[0, 1, 0]* and *[0, 0, 1]* respectively. For evaluation, we set the maximum number of types to ten. More sophisticated representations of types, e.g., learned jointly with the overall model [248], could be integrated into Nalin as part of future work.

REPRESENTING LENGTH AND SHAPE Since both length and shape refer to analogous concepts, we represent them in a similar fashion. By encoding these two properties of values, we aim at capturing some of the information lost during truncation of large values (Section 6.3.1.2). The intuition is that some variable names, such as `name` or `our_pattern`, usually refer to relatively short values, whereas other names, such `poem` or `DNA5`, are typically refer to longer values. During truncation, the difference in lengths of such values may be lost, but the length and shape properties still encode them.

Because the length of a value is theoretically unbounded, we consider ten ranges of lengths and represent each of them with a one-hot vector. Specifically, Nalin consider ranges of length 100, starting from 0 until 1,000. That is, any length between 0 and 100 will be represented by the same one-hot vector, and likewise any length greater than 1,000 will be represented by the another vector. The shape of a value is a tuple of discrete numbers, which we represent similarly to the length, except that we first multiple the elements of the shape tuple. For example, for a value of shape x, y, z, we encode $x \cdot y \cdot z$ using the same approach as for the length For values that do not have a length or shape, we use a special one-hot vector.

6.3.4 *Training and Prediction*

Once Nalin has obtained a vector representation for each component of an name-value pair, the individual vectors are concatenates into the combined representation of the pair. This combined representation is then fed into a neural classifier that predicts the probability p of the name-value pair to inconsistent. The classification model consists of two linear layers with a sigmoid activation function at the end. We also add a dropout with probability of 0.5 before each linear layer. We train the model for 15 epochs with the Adam [119] optimizer, and a batch size of 128. During training, the model is trained toward predicting $p = 0.0$ for all positive examples and $p = 1.0$ for all negative examples. Once trained, we interpret the predicted probability p as the confidence Nalin has in flagging a name-value pair as inconsistent, and the approach reports only pairs with p above some threshold to the user. We use a threshold probability of 0.5 as the default for reporting a warning.

6.4 IMPLEMENTATION

Python being one of the most popular[2] programming languages, we choose it as a concrete application of Nalin and gather assignments from a corpus of computational notebooks written in Python. The type of computational notebook used for our purposes is Jupyter Notebook. We also implement our approach using Python into an end-to-end framework to find name-value inconsistencies. Each Jupyter

2 https://www.tiobe.com/tiobe-index

notebook is first converted to a Python script using *nbconvert* and then instrumented using the AST provided by *LibCST*. One of the challenges of executing a large number of Python scripts is the presence of a large number of external dependencies. We do not try to install all possible dependencies, but rather install the most popular 100 Python libraries. Additionally, executing arbitrary Python scripts downloaded from the public domain can have unknown security implications, such as making unsolicited network requests or downloading large number of files locally. We circumvent such risks by executing the scripts in a sandbox environment.

6.5 EVALUATION

Our evaluation focuses on the following research questions:

- RQ1: How effective is Nalin in detecting name-value inconsistencies?
- RQ2: What kinds of inconsistencies does the approach find in real-world code?
- RQ3: How does our approach compare to static bug detection tools?
- RQ4: What is the effect of type-guided negative example generation compared to the purely random approach?
- RQ5: Which properties of name-value pairs contribute the most to finding name-value inconsistencies?

6.5.1 *Experimental Setup*

We evaluate our approach on one million computational notebooks sampled from an existing dataset of Jupyter notebooks scrapped from GitHub [228]. The experiments have been run on a machine with Intel Xeon E5-2650 CPU having 48 cores, 64GB of memory and an NVIDIA Tesla P100 GPU. The machine runs on Ubuntu 18.04, and we use Python 3.8 for the implementation.

Excluding some malformed notebooks, we convert 985,865 notebooks into Python scripts. Some of these notebooks contain only text and no code, while for others, the code has syntax errors, or the code is very short and does not perform any assignments. All of

this decreases the number of actual Python files that Nalin instruments, and we finally obtain 598,321 instrumented files, which takes approximately two hours to instrument.

There are three main challenges in gathering name-value pairs, partially related to previously discussed challenges in reproducing Jupyter notebooks [264]. First, even with the installation of most popular Python packages, we fail to satisfy the dependencies of some files, which results in crashes during executions. Second, many Python scripts read inputs from files, e.g., a dataset for training a machine learning model, which may not be locally available. Third, many assigned values are of custom object types that do not have a string representation, i.e., we can obtain only the name and reference of the object, which is not very useful and needs to be discarded. Considering all notebooks that we can successfully execute despite these obstacles, Nalin gathers a total of 947,702 name-value pairs, of which 500,332 remain after the filtering described in Section 6.3.1.2. Running the instrumented files to extract name-value pairs takes approximately 48 hours.

The name-value pairs consist of a diverse set of values and types. Figure 6.4 shows the ten most frequent types and the corresponding frequencies across the dataset. The presence of a large number of collection types, such as `list` and `ndarray`, which usually are not fully initialized as literals shows that extracting values at run-time is worthwhile.

Before running any experiments with the model, we sample 10,000 name-value pairs as a held-out *test dataset*. Unless mentioned otherwise, all reported results are on this test dataset. On the remaining 490,332 name-value pairs, we perform an 80-20 split into *training* and *validation* data. For each name-value pair present in the training, validation, and test datasets, we create a corresponding negative example, which takes two hours in total. The total number of data points used to train the Nalin model hence is about 780k. Training takes an average of 190 seconds per epoch and once trained, prediction on the entire test dataset takes about 15 seconds.

VARIANTS OF THE APPROACH In addition to training a model on all name-value pairs in our dataset, we also experiment with two models trained on subsets of the dataset with pairs of similar types. One subset contains all 145,524 name-value pairs where the values are of type `int` or `float`. The other subset focuses on 182,404 name-

value pairs with the iterable types `str` and `list`. The intuition is that since the pairs of chosen data types are very similar to each other, developers may be more likely to confuse values and variables of these types. Experimenting on the subsets of the dataset allows us to measure if Nalin is able to pickup the subtle differences between values of similar types. As for the full dataset, we keep 10,000 pairs as a test dataset, and perform an 80-20 split of the remaining data for training and validation, respectively. We call the model trained on the full dataset *all-types*, and the variants *int-float* and *str-list*, respectively. We consider the all-types model as the default variant of Nalin and refer to it, unless mentioned otherwise.

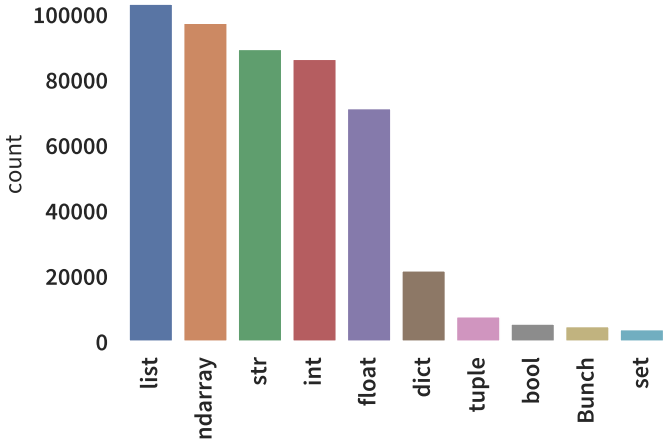


Figure 6.4: Distribution of different types in the dataset

6.5.2 *RQ1: Effectiveness in Detecting Inconsistencies*

To measure the effectiveness of Nalin in detecting inconsistencies, we apply the trained all-types, int-float, and str-list models to the held-out test datasets. The output of the model is a probability score that indicates how likely the model believes a given name-value pair to be inconsistent. We consider all probabilities above a threshold as a warning and measure the precision and recall of the approach as

follows. Precision indicates how many of the warnings produced by the model are actually inconsistent:

$$\frac{|\text{warnings that are actually inconsistent}|}{|\text{total warnings}|}$$

Recall is the percentage of inconsistencies that the model correctly reports among all inconsistencies in the dataset:

$$\frac{|\text{inconsistent pairs reported as inconsistent by the model}|}{|\text{total inconsistent pairs}|}$$

We also report the F1 score, which is the harmonic mean of precision and recall.

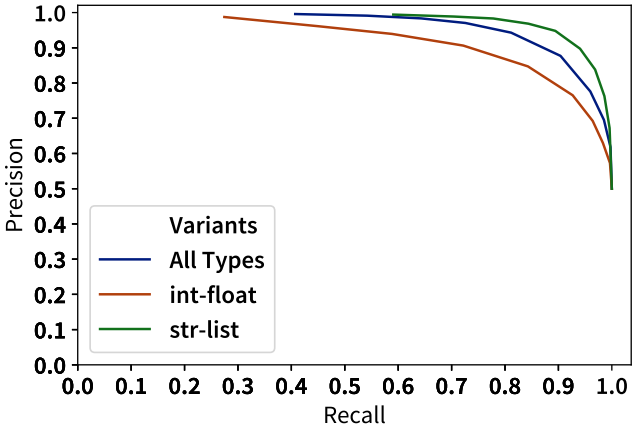


Figure 6.5: Precision and recall with different thresholds for reporting warnings.

Figure 6.5 shows the precision and recall tradeoff using different thresholds for reporting a prediction as a warning. Overall, all three models achieve high precision and recall values. For example, using the default threshold of 0.5, the all-types model has a precision of 0.94 and a recall of 0.81, giving an F1 score of 0.87. Comparing the all-types model with the type-specific variants, we find the str-list model to be the most effective. On inspecting examples in the test dataset of the str-list model, we find that it picks up minor details in name-value pairs, which contributes to the effectiveness. For

example, the model correctly classifies `color = 'yellow'` and `colors = ['red', 'green', 'blue', 'yellow']` as consistent, but reports a warning for `country = ['United States', 'France', 'Japan'...]`, where the name should be plural.

6.5.3 *RQ2: Kinds of Inconsistencies in Real-World Code*

To understand the ability of Nalin to detect name-value inconsistencies in real-world code, we inspect supposedly positive name-value pairs in the test datasets that are classified as inconsistent by the model. When using Nalin to search for previously unknown issues, these name-value pairs will be reported as warnings. For each of the all-types, int-float and str-list models, we inspect the top-30 predictions, sorted by the probability score provided by the model, and classify each warning into one of three categories:

- *Incorrect value*. Name-value pairs where the mismatch between a name and a value is due to an incorrect value being assigned. These cases cause unexpected program behavior, e.g., a program crash or incorrect output.
- *Misleading name*. Name-value pairs where the name clearly fails to match the value it refers to. These cases do not lead to wrong program behavior, but should be fixed to increase the readability and maintainability of the code.
- *False positive*. Name-value pairs that are consistent with each other, and which ideally would not be reported as a warning.

Because deciding about misleading names is to some extent subjective, two of the authors independently inspect each warning and discuss any cases of disagreements. We categorize a warning as a misleading name only when the two authors eventually agree on it, and categorize a warning as a false positive otherwise.

Figure 6.6 summarizes the results of the manual inspection. Overall, the three models have a precision of 51%, i.e., the majority of the reported warnings corresponds to either an incorrect value or a misleading name. Five of the true positives are incorrect values, while the larger part of the true positives (42 in total) are misleading names.

Table 6.1 and 6.2 show some examples of warnings produced by Nalin, along with our categorization. In the the first example of Table 6.1, Nalin produces a warning about the assignment on line 2.

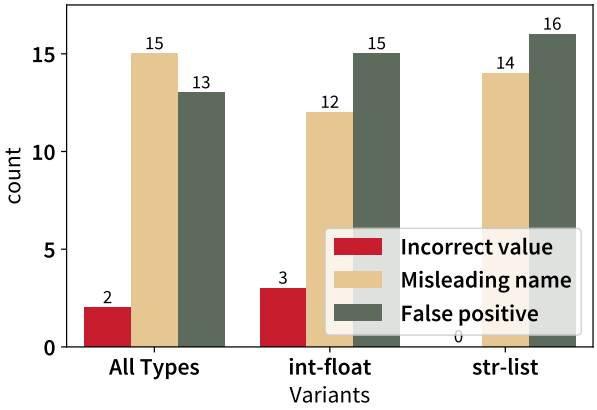


Figure 6.6: Manual inspection results.

The value assigned during the execution is a string `'Cooperate'`. Due to the string assignment, the code on line 3 crashes since the operator `>` does not support a comparison between a string and float. Nalin is correct in predicting this warning because the variable name `prob` is typically used to refer to a probability, not to a string like `'Cooperate'`. Since the assignment leads to an unexpected behavior, in this case a crash, we categorize the name-value pair as an incorrect value.

The next two examples in Table 6.1 show misleading names. For example, it is highly unusual to assign a number to a variable called `name` or to assign a list of numbers to a variable called `matchstring`. To the best of our knowledge, these misleading name do not cause unexpected behavior, but developers may still want to fix them to increase the readability and maintainability of the code.

The two examples in Table 6.2 show false positives. The examples illustrate one of the most common causes of false positives seen during our inspection, namely generic variable names that do not convey much information about the values they may refer to. Future work could further increase the precision of Nalin by filtering warnings related to such generic names.

Code Example	Category	Comment

`def Custom(information):`
`  prob = get_betraying_probability(information)`
`  # Runtime value --> 'Corporate'`
`  if(prob > 1 / 2):`
`  return D`
`  elif(prob == 1 / 2):`
`  return choice([D, C])`
`  else:`
`  return C`
	Incorrect value	This warning has been produced by the all-types model. Assigning a string type value to a variable called `prob` is unusual. This unfortunately leads to a crash in the program in the next line and as a result, we classify the warning as an incorrect value.

`name = 'Philip K. Dick'`
`...`
`name = 2.5`
`# Runtime value --> 2.5`
`if type(name) == str:`
`  print('yes')`
	Misleading name	This warning has been produced by the all-types model. Assigning a float value to a variable called `name` is highly unusual and likely to mislead developers. As the assignment does not lead to unexpected behavior, we classify it as a misleading name.

`matchstring = list(range(36))`
`# Runtime value --> [0, 1, 2,...]`
`for i in range(18):`
`  matchstring[2*i] = np.random.binomial(1, server1p)`
`  matchstring[2*i+1] = np.random.binomial(1, 1-server2p)`
`sum(matchstring)`
	Misleading name	This warning has been produced by the str-list model. Assigning a list value to a variable called `matchstring` is clearly misleading, but does not lead to unexpected behavior.

Table 6.1: Examples of warnings produced by Nalin.

Code Example	Category	Comment

`data = str(random.random() + 4)`
`MyObj = namedtuple("MyClassReplacement", ("`
`  some_string", "my_smart_function",))`
`o = MyObj(`
`  some_string=data,`
`  my_smart_function=lambda item: float(item) * 3)`
`some_string, some_function = o`
`# Runtime value --> "4.29403..905"`
	False positive	This warning has been produced by the all-types model. The value is a string that describes a range of numbers, which fits the (rather generic) name of the variable called `some_string`. Since the assigned value is indeed a string, we classify this warning as a false positive.

`while (number1 + number2 != number1) :`
`  counter += 1`
`  number2 = number2 / 10.0`
`  # Runtime value --> 1.00001e-16`
	False positive	This warning has been produced by the int-float model. The name `number2` is a generic and may hold either an int or a float value, which is inline with the observed value

Table 6.2: Examples of warnings produced by Nalin.

6.5.4 *RQ3: Comparison with Previous Bug Detection Approaches*

We compare Nalin to three state-of-the-art static analysis tools aimed at finding bugs and other kinds of noteworthy issues: (i) *pyre*, a static type checker for Python that infers types and uses available type annotations; (ii) *flake8*, a Python linter that warns about various code quality issues and commonly made mistakes; and (iii) *DeepBugs* [226], a learning-based bug detection technique aimed at name-related bugs. Since all three approaches are not limited to specific types, we use the all-types model for this comparison. We run pyre and flake8 using their default configurations. For DeepBugs, we install the "DeepBugs for Python" plugin from the marketplace of the PyCharm IDE. We apply each of the three approaches to the 30 files where Nalin has produced a warning and which have been manually inspected (RQ2).

Table 6.3: Comparison with existing static bug detectors.

Approach	Warnings	Warnings common with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30

Table 6.3 shows the number of warnings reported by the existing tools and how many of these warnings overlap with those reported by Nalin. We find that except one warning reported by pyre, none matches with the 30 manually inspected warnings from Nalin. The matching warning is a misleading name, shown on the second row of Table 6.1. The pyre type checker reports this as an "Incompatible variable type" because in the same file, the variable `name` is first assigned a string `'Philip K. Dick'` and later assigned a float value `2.5`. The 1,247 warnings produced by flake8 are mostly about coding style, e.g., "missing white space" and "whitespace after '(' ". The warnings reported by DeepBugs include possibly wrong operator usages and incorrectly ordered function arguments, but none matches the warnings reported by Nalin. We conclude that Nalin is complementary to both traditional static analysis-based tools and to a state-of-the-art learning-based bug detector aimed at name-related bugs.

6.5.5 *RQ4: Type-Guided vs. Purely Random Negative Examples*

The following compares the two algorithms for generating negative examples described in Section 6.3.2. Following the setup from RQ1, we find that the purely random generation reduces both precision and recall, leading to a maximum F1 score of 0.82, compared to 0.87 with the type-guided approach. With a reporting threshold of 0.5, the purely random approach reports a total of 820 warnings, i.e., 40% more than the 490 reported warnings by the type-guided approach. Manually inspecting the top-30 reported warnings as in RQ2, we find 21 false positives, nine misleading names, and zero incorrect values, which reduces the precision from 57% to 30% and is clearly worse then the results with the type-guided generation of negative examples. Overall, these results confirm the observation that motivates the type-guided algorithm (Section 6.3.2.2) and show that it outperforms a simpler baseline.

6.5.6 *RQ5: Ablation Study*

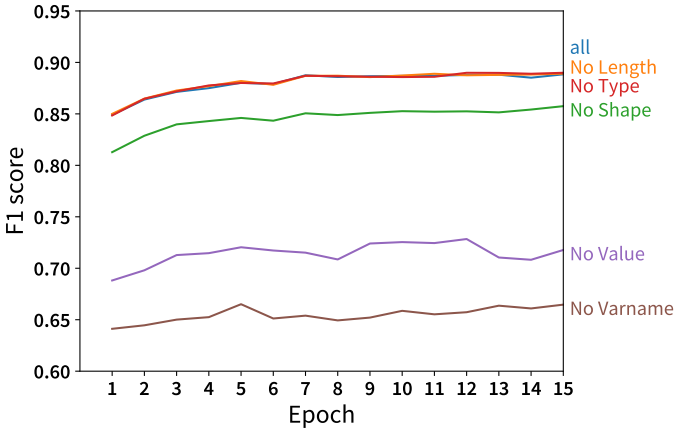


Figure 6.7: Ablation Study Results

We perform an ablation study on the all-types model to measure the importance of the different components of a name-value pair fed into the model. To this end, we set the vector representation of individual components to zero during training and prediction, and

then measure the effect on the F1 score of the model. Figure 6.7 shows the results, where the vertical axis shows the F1 score obtained on the validation dataset at each epoch during training. Each line in Figure 6.7 shows the F1 score obtained while training the model keeping that particular feature set to zero. For example, the green line ("No Shape") is for a model that does not use the shape of a value, and the blue line ("all") is for a model that uses all components of a name-value pair. We find that the most important inputs to the model are the variable name and the string representation of the value. Removing the length or the type of a value does not significantly decrease the model's effectiveness. The reason is that these properties can often be inferred from other inputs given to the model, e.g., by deriving the type from the string representation of a value. We confirm this explanation by removing both the type and the string representation of a value, which yields an F1 score similar to the model trained by removing only values.

6.6 CONCLUSION

Using meaningful identifier names is important for code understandability and maintainability. This chapter presents Nalin, which addresses the problem of finding inconsistencies that arise due to the use of a misleading name or due to assigning an incorrect value. The key novelty of Nalin is to dynamically analyze a corpus of Python files and learn from names and their values assigned at runtime, which is in contrast to traditional statistic analysis approaches. To reason about the meaning of names and values, the approach embeds them into vector representations that assign similar vectors to similar names and values. Our evaluation with almost 800k name-value pairs gathered from real-world Python programs shows that the model is effective, with an F1 score of 0.87 on previously unseen data. A manual inspection of a sample of the reported name-value inconsistencies shows that Nalin reports true positives with a precision of 51%, including various misleading names and some incorrect values that cause clear misbehavior.

We envision Nalin to be complementary to other approaches that report warnings on possible inconsistencies in code. We believe our work provide multiple avenues of future work. One direction that may improve the model is to include context information such as adding tokens that appear close to name-value pairs. Another direction is to

train using name-value pairs recorded at runtime along with extracted using a static analysis. A combined dataset will increase the size of the training dataset and possibly alleviate the false positive rate.

Part III

CORPUS-BASED INPUT REDUCTION

In addition to bug finding, we leverage large corpora of code for other tasks, such as reducing test inputs.

7

AUTOMATICALLY REDUCING TREE-STRUCTURED TEST INPUTS

Chapter 2 and 3 present two approaches where we generate a large number programs by either generating complete programs or by mutating existing programs. We also show that the generated programs are either useful for finding bugs or training neural models. As an extension of the approaches to generate programs, this chapter presents a way to reduce inputs that may be also applied to the generated programs. For example, if a generated program by TreeFuzz (Chapter 2) uncovers a bug, it is helpful for developers to get a subset of the program that still triggers the same bug. The current chapter provides a solution in that direction by reducing inputs that may be represented as trees.

7.1 MOTIVATION

Developers often have a test input that triggers behavior of interest, such as inducing a failure in a buggy program or covering particular parts of a program under test. However, the input may be larger than needed to preserve the property of interest. For example, consider a program that crashes the compiler or interpreter when given as an input. The larger this input program is, the more difficult it is to localize the fault, making the debugging process unnecessarily cumbersome [6].

To ease the task of dealing with such overly complex test inputs, several automated techniques have been proposed. Given a test input and an oracle that determines whether a reduced version of the input still preserves the property of interest, these techniques automatically reduce the input. With a reduced test input, the developer is likely to find the root cause of the bug faster and may even turn the reduced

test input into a regression test case after the bug has been fixed. Similar, reducing test inputs while preserving some testing goal, such as coverage, can help to reduce a test suite.

Existing techniques for reducing inputs roughly fall into two categories. On the one hand, delta debugging [16] and its derivatives [31] reduce inputs in a language-independent way by repeatedly removing parts of the input until no further reduction is possible. While being simple and elegant, these approaches disregard the language of the input and therefore miss opportunities for input reduction. In particular, these techniques cannot restructure inputs, which often enables further reductions. As an example, consider the following JavaScript code and suppose that it triggers a bug, e.g., by crashing the underlying JavaScript engine.

```
for (var i = 0; i < 10; i++) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```

Further suppose that the two function calls are sufficient to trigger the bug. That is, the following code is sufficient as a test input to enable a developer to reproduce and localize the bug:

```
partOfBug();
otherPartOfBug();
```

Unfortunately, existing language-independent techniques are challenged by this example. The original delta debugging algorithm blindly removes parts of the program, which is likely to lead to a syntactically invalid program or to a local minimum that is larger than the fully reduced example. Hierarchical delta debugging [31], a variant of delta debugging that considers the tree structure of the input, fails to find the reduced input because it can remove only entire subtrees, but it cannot restructure the input.

On the other hand, some techniques [88] exploit domain knowledge about the language of the test input. While being potentially more effective, hard-coding language knowledge into the approach limits it to a single kind of test input.

This chapter presents the Generalized Tree Reduction algorithm (GTR), a language-independent technique to reduce arbitrary tree-structured test inputs. The approach is enabled by two key observations. First, we observe that transformations beyond removing entire

parts of the input are beneficial in reducing inputs. GTR exploits this observation by incorporating tree transformations into the reduction process. The challenge is how to know which transformations to apply without hard-coding knowledge about a particular language. Addressing this challenge improves the effectiveness of test input reduction. Second, we observe that for most relevant input formats, there are various examples that implicitly encode information about the language. For example, there are various programs in public code repositories, millions of HTML files, and many publicly available XML documents. GTR exploits this situation to improve the efficiency of input reduction by automatically pruning the search space of transformations for a particular language after learning from a corpus of example data.

Our work focuses on inputs that can be represented as a tree. This focus is motivated by the fact that the inputs of many programs have an inherent tree structure, e.g., XML documents, or can be easily converted into a tree, e.g., the abstract syntax tree of source code. The input to GTR is a tree with a desirable property, such as triggering a bug, and an oracle that determines whether a reduced version of the tree still has the desirable property. The algorithm reduces the tree level by level, i.e., it considers all nodes of a level to minimize the whole tree, before continuing with the next level. The output of the algorithm is a reduced tree that has the desirable property according to the oracle.

At the core of our approach are tree transformations that modify a tree into a new tree with fewer nodes. We describe two transformation templates that we find to be particularly effective. The first template removes a node and all its children, drastically shrinking the tree's size. As deleting nodes alone is insufficient for various inputs, the second template replaces a node with one of its children, i.e., it pulls up a subtree to the next level of the tree. While we find these two transformation templates to be effective, the algorithm is easily extensible with additional templates. In principle, these transformation templates are applicable to arbitrary kinds of nodes in the tree. To reduce the size of the search space considered by GTR, i.e., ultimately the time required to reduce an input, we specialize the transformation patterns to a specific input language by learning from a corpus of example data. Since the learning is fully automatic, the approach remains language-independent.

To evaluate GTR, we apply the algorithm to a total of 429 inputs in the form of Python programs, JavaScript programs, PDF documents, and XML documents. The Python programs each trigger a bug in the Python interpreter, while the JavaScript programs cause inconsistencies between browsers. The PDF documents contain malicious content. The XML documents achieve a certain coverage when given to an XML validator, and that coverage should be preserved during the reduction. We find that GTR reduces the inputs to 45.3%, 3.6%, 44.2%, and 1.3% of the original size, respectively. Compared to the best existing approach [31], GTR consistently improves efficiency and also significantly improves the effectiveness of reduction in three of four experiments.

To summarize, we make the following contributions:

- We identify the lack of restructuring as a crucial limitation of existing language-independent input reduction techniques.
- We present a novel tree reduction algorithm that transforms trees based on tree transformation templates. If a set of example inputs is available, the approach automatically specializes the templates to the language of the input.
- We show the presented algorithm to be significantly more effective and efficient than two state-of-the-art techniques.
- We make our implementation available to the public.[1]

7.2 BACKGROUND

7.2.1 *Delta Debugging*

Zeller and Hildebrandt proposed delta debugging (DD) [16], a greedy algorithm for isolating failure inducing inputs. In a nutshell, DD splits the input in chunks of decreasing sizes, trying to remove some chunks while maintaining a property of the input. "Chunk" can refer, e.g., to individual characters or lines of a document. Often but not necessarily, the property is that the input induces a bug when fed to a program. DD does not guarantee to find the smallest possible input but instead ensures *1-minimality*. This property guarantees that no single part of the input can be removed without loosing the property of interest. For example, when applying line-based delta debugging to reduce a program that triggers a compiler bug, 1-minimality means that

1 https://github.com/sherfert/GTR

removing any line of the input will cause the input to not trigger the bug anymore.

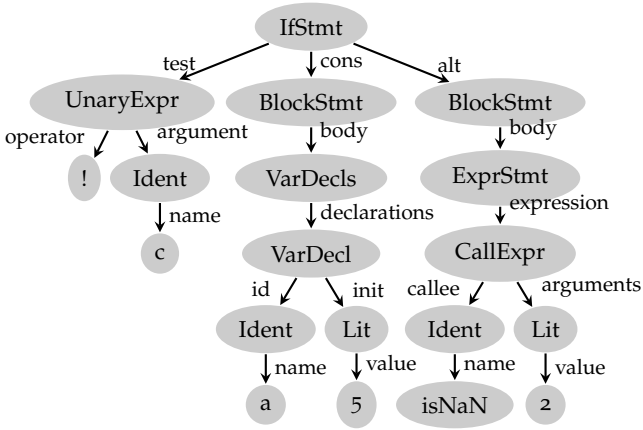
DD has an important disadvantage for structured input because it disregards the structure of the input when splitting it into chunks. As a result, DD may generate various invalid inputs and invoke the oracle unnecessarily. For instance, when applying DD to the example from the introduction, the algorithm may delete a closing bracket without removing its counterpart, generating a syntactically invalid program. Since each candidate input is given to the oracle, such invalid inputs increase the execution time of the algorithm.

7.2.2 *Hierarchical Delta Debugging*

Hierarchical delta debugging (HDD) [31] addresses the limitation that DD disregards the structure of the input. The algorithm considers the input to be a tree, which is a natural way to interpret various inputs, e.g., code represented as an abstract syntax tree (AST). HDD starts from the root of the tree and visits each level. At every level, the algorithm applies the original DD algorithm to all nodes at this level to find the smallest set of nodes necessary. The algorithm terminates after running DD on the last level of the tree.

HDD often purges large parts of the input early, leading to more reduction than DD, while also requiring fewer oracle invocations. In contrast to DD, HDD does not provide 1-minimality. To guarantee this property, HDD* repeatedly uses HDD, until no more changes to the tree are performed [31]. A limitation of HDD is that it only removed nodes (and all their children), but it does not use any other tree transformations. Therefore, for an input where the important part is deeply nested in the input tree, HDD produces far-from-minimal results. An example is the code excerpt provided in the introduction. Here, HDD attempts to remove the entire if-branches, which yields a program that does not trigger the bug anymore. The algorithm yields the following code as the reduced input:

```
for (;;) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```




```
if(!c) {
  var a = 5; // root cause of failure
} else {
  isNaN(2);
}
```

Figure 7.1: An abstract syntax tree and code for our running example.

7.3 PROBLEM STATEMENT

Previous approaches for reducing failure-inducing inputs miss opportunities for reduction because they ignore the structure of the input and because they are limited to removing parts of the input. Motivated by these limitations, we aim for an algorithm that exploits the structure of the inputs, and that finds near-minimal results even when the root cause of a failure is deeply nested inside the input. Our work focuses on inputs that can be represented as a tree.

Definition 7.1. *A* labeled ordered tree *is a recursive data structure* (l, c)*, where* l *is a textual label and* c *is the (possibly empty) ordered list of outgoing edges. An edge* $e \in c$ *is a tuple* (l, t)*, where* l *is a textual label for the edge and* t *is the child node, which itself is a labeled ordered tree. We use* $\mathcal{T}$ *to refer to the set of all trees.*

Figure 7.1 shows an example: A small piece of JavaScript code that triggers a bug, e.g., in a JavaScript engine. Suppose that the bug is triggered by the statement at line 2. The example input can be represented as a tree – in this case, the AST.

We will refer to a labeled ordered tree hereafter simply as a tree or node, depending on the context. Trees have several properties. The *size* of a tree is the number of its nodes: $size : \mathcal{T} \to \mathbb{N}$. The tree in Figure 7.1 has a size of 19. The *context* of a tree is a partial function that returns the label of the parent node and the label of the incoming edge: $context : \mathcal{T} \to (String \times String)$. The context of the root node is undefined. For the example, the context of the *UnaryExpr* node on the left side of our tree is $(IfStmt, test)$. The *level* of a node in a tree is the edge-distance from the node to the tree's root node. All nodes of a particular level in a tree can be obtained by a function $level : (\mathcal{T} \times \mathbb{N}) \to P(\mathcal{T})$, where $P(\mathcal{T})$ denotes the power set of $\mathcal{T}$. The *depth* of a tree is defined as the maximum distance of a leaf node to the root: $depth : \mathcal{T} \to \mathbb{N}$. The example tree has a depth of 5. Finally, we say that a tree t' is derived from another tree t, written $derived(t', t)$, if one can build t' from t by deleting nodes and edges, or by moving nodes and edges within the tree without changing a single label.

Definition 7.2. *An* oracle o *is a function that, given a tree, decides whether the tree provides a desired property:* $o : \mathcal{T} \to Bool$. *We use* $\mathcal{O}$ *to denote the set of all oracles.*

A tree t' is *minimal* w.r.t. an oracle o and a source tree t if t' satisfies the oracle and if there is no smaller derived tree that also satisfies the oracle. Formally, t' is minimal if $derived(t', t) \wedge o(t') = true \wedge \left(\nexists t'' \neq t' : derived(t'', t) \wedge o(t'') = true \wedge size(t'') < size(t')\right)$.

Definition 7.3. *A* tree reduction algorithm *is a function* $A : (\mathcal{T} \times \mathcal{O}) \to \mathcal{T}$ *that, given a tree* t *and an oracle* o *where* $o(t) = true$, *returns another tree* t' *for which* $o(t') = true$ *and* $size(t') \leqslant size(t)$.

The algorithm tries to find a smaller tree that still provides a property of interest, as decided by the oracle. If a reduction algorithm cannot further reduce a tree, it will return the same tree.

The goal of this work is to provide a tree reduction algorithm that returns near-minimal trees with respect to the given oracle, while maintaining the number of oracle invocations low. A small number of oracle invocations is important, as they can be costly operations, such as running a compiler, that significantly increase the overall runtime of the algorithm. In general, finding the minimal tree is impractical because the number of trees to check with the oracle grows exponentially with the size of the input tree.

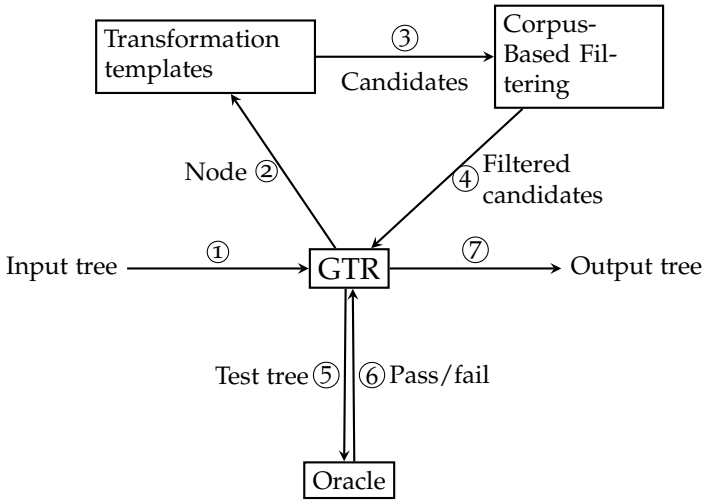


Figure 7.2: Overview of the GTR approach.

7.4 THE GENERALIZED TREE REDUCTION ALGORITHM

This section introduces a novel tree reduction algorithm, called *Generalized Tree Reduction* or GTR. Figure 7.2 shows the components of the approach and how they interact with each other. Given an input tree (step 1), the algorithm traverses the tree from top to bottom while applying transformations to reduce the tree. For example, a transformation may remove an entire subtree or restructure the nodes of the tree. The transformations are based on tree transformation templates (step 2) that specify a set of candidate transformations (step 3). To specialize a generic template to a particular input format, the approach optionally filters these candidates based on knowledge inferred from a corpus of example inputs (step 4). The algorithm applies the transformations and queries the oracle to check whether a reduced tree preserves the property of interest, e.g., whether it still triggers a particular bug (steps 5 and 6). The algorithm repeatedly reduces the tree until no more tree reductions are found. Finally, GTR returns the reduced tree (step 7).

We call the GTR algorithm "generalized" because it can express different tree reduction algorithms, depending on the provided tree transformation templates. For example, by providing a single tem-

plate that reduces entire subtrees, GTR is equivalent to the existing HDD algorithm [31] (Section 7.4.5).

Before delving into the details of GTR, we illustrate its main ideas using the running example in Figure 7.1. Given the tree representation of the input, the algorithm analyzes the tree level by level, starting at the root node. For example, the algorithm considers a transformation that removes the root node *IfStmt* and all its children, but discards this transformation because the reduced tree (an empty program) does not trigger the bug anymore. As another example, the algorithm considers transformations that replace the root node with one of its children. Replacing the root node with the *BlockStmt* that represents the then-branch yields a smaller tree that still triggers the bug. Therefore, the algorithm applies this transformation and continues to further reduce the remaining tree. Eventually, the algorithm reaches a tree that represents only the statement `var a = 5`, which cannot be reduced without destroying the property of interest.

The remainder of this section explains the GTR algorithm in detail. At first, we present the tree transformations applied by the algorithm (Sections 7.4.1 and 7.4.2). Then, we describe how GTR combines different transformations into an effective tree reduction algorithm (Section 7.4.3).

7.4.1 *Tree Transformation Templates*

The core ingredient of GTR are transformations that reduce the size of a tree. We specify such transformations with templates:

Definition 7.4. *A* transformation template *is a function* $\mathcal{T} \to P(\mathcal{T}) \cup \{DEL\}$ *that returns a set of candidate trees that are the result of transforming a given input tree. In addition to candidate trees, the template may return the special symbol DEL, which indicates that the tree should be removed rather than modified.*

In this chapter, we focus on two transformation templates, which yield a tree reduction algorithm that is more effective than the best existing algorithms.

DELETION TEMPLATE The first template addresses situations where an entire subtree of the input given to GTR is irrelevant for the property of interest. In our running example (Figure 7.1), the subtree rooted at the right-most *BlockStmt* node is such a subtree. To enable

Algorithm 7.1 Substitute-by-child template

Input: a tree *tree*
Output: a set of candidates nodes
1: **function** SBC_TEMPLATE(*tree*)
2: *candidates* $\leftarrow \emptyset$
3: **for** $i \in [0, |tree.c|]$ **do**
4: $c \leftarrow tree.c[i].t$
5: *candidates* $\leftarrow$ *candidates* $\cup \{c\}$
6: **return** *candidates*

GTR to remove such subtrees, the deletion template simply suggests for each given tree to delete it by returning the special *DEL* symbol.

SUBSTITUTE-BY-CHILD TEMPLATE The second template addresses situations where simply removing an entire subtree is undesirable because the subtree contains nodes relevant for the property of interest. We observe that a common pattern is that the root node of a subtree is irrelevant but one of its children is important for the property of interest. In the running example, the tree rooted at the *IfStmt* matches this pattern, because the if statement is irrelevant, but the nested variable declaration is crucial. To address this pattern, the substitute-by-child template (Algorithm 7.1) returns each child of a given tree's root node as a candidate for replacing the given tree. The template iterates over all children of the given tree and adds each of them to the set of candidates. Applying this transformation template to the *IfStmt* of the running example yields a set of three candidates, namely the three subtrees rooted at nodes *UnaryExpr*, *BlockStmt*, and *BlockStmt*.

Beyond these two templates, additional templates can be easily integrated into GTR, enabling the approach to express different tree reduction algorithms.

7.4.2 *Corpus-Based Filtering*

The templates defined above are completely language independent. When applying these templates to a tree that ought to conform to a specific input format, many of the candidates may be rejected by the oracle simply because they violate the input format. For example, when the deletion template suggests to remove the *UnaryExpr* from the tree in Figure 7.1, the resulting tree corresponds to syntactically

invalid JavaScript code because every if statement requires a condition. Suggesting such invalid candidates does not influence the effectiveness of our approach because the oracle rejects all invalid candidates. However, a high number of invalid candidates negatively influences the efficiency of the approach since invoking the oracle often imposes a significant runtime cost.

To address the challenge of invalid candidates, we enhance the approach with a language-specific filtering of candidates trees that rejects invalid trees before invoking the oracle. To preserve the language-independence of GTR, the filtering is based on knowledge that gets automatically inferred from a corpus of example inputs in the specific input format. For example, the approach learns from a set of JavaScript programs that if-statements require a condition, and therefore, will filter any candidates that violate this requirement.

DELETION TEMPLATE To specialize the deletion template to a particular language, we need to know which edges are mandatory for particular node types. The approach analyzes the code corpus to find a set of mandatory edge labels for each node label. An edge is considered mandatory, if it appears on all nodes with the label across the whole corpus. Based on the mandatory edges, we modify the deletion template so that a node only can be deleted if it is not a mandatory child of its parent node.

For the running example, consider again the candidate that suggests to remove from the *IfStmt* the *UnaryExpr* subtree. The corpus analysis finds that the set of mandatory edges for an *IfStmt* is {*test*, *cons*}. Based on this inferred knowledge, the algorithm will not attempt to delete the *UnaryExpr* anymore, but discards this candidate before needlessly passing the tree to the oracle.

SUBSTITUTE-BY-CHILD TEMPLATE To specialize the substitute-by-child template, we gather information on the parent node labels and incoming edge labels of nodes. Specifically, for each node label we collect a set of pairs (p, e) where p is the label of the parent, and e is the label of the incoming edge. This set of pairs is equivalent to all distinct contexts of nodes with that label and we call it the *allowed contexts*. We then replace line 5 of the substitute-by-child template (Algorithm 7.1) with the following steps:

5: **if** $context(tree) \in allowedContexts(tree.c[i].l)$ **then**
6: $\quad candidates \leftarrow candidates \cup \{c\}$

Algorithm 7.2 Generalized tree reduction

Input: tree t, oracle o, set $\mathcal{L}$ of templates
Output: reduced tree

```
1: for i ∈ [0, depth(t)] do
2:     for l ∈ L do
3:         t ← applyTemplate(t, i, o, l)

4: function applyTemplate(t, i, o, l)
5:     levelNodes ← level(t, i)                    ▷ All nodes of level i
6:     if l returns at most one transformation then
7:         newNodes ← apply DD to replace levelNodes using l
8:         return tree where newNodes replace levelNodes
9:     else
10:        return reduceLevelNodes(t, levelNodes, o, l)    ▷ Alg. 7.3
```

The specialized variant of the template checks if the child that we replace the node with can also appear in the same context as the node. For example, the approach infers that there is one valid context for a *VarDecl* node, namely *(VarDecls, declarations)*. Since *(BlockStat, body)* is not a valid context, the algorithm will immediately discard a candidate that tries to substitute *VarDecls* with *VarDecl*.

Inferring from a corpus of examples how to specialize language-independent transformation templates to an input format is optional and automatic. It is automatic because for most input formats used in practice, there are sufficiently many examples to learn from. An alternative approach could be to use a formal grammar of the input language to filter syntactically invalid trees. We rejected this idea because (i) a grammar may not be available, e.g., for proprietary formats, (ii) the checks performed by the specialized transformation templates are more lightweight than parsing the entire input tree with a grammar. Our evaluation measures the effectiveness and efficiency of GTR with and without the corpus-based filtering of candidate trees (Section 7.5.4).

7.4.3 *GTR Algorithm*

Based on the transformation templates described above, the GTR algorithm reduces a given tree by applying transformations at each level of the tree. Algorithm 7.2 summarizes the main steps. Starting at the root node, the algorithm considers each level of the tree and

applies all available transformation templates to each level using a helper function *applyTemplate* (lines 1 to 3).

DELTA DEBUGGING-BASED SEARCH When applying a template to the nodes at a particular level, the algorithm distinguishes between templates that return at most one candidate transformation, such as the deletion template, and other templates. In the first case, the algorithm needs to decide for which nodes to apply the suggested transformations. This problem can be reduced to delta debugging (DD). The chunks needed as input for DD are the nodes of the level. DD then tries to combine as many replacements as possible while querying the oracle to check if a replacement preserves the property of interest. For each node n for which the transformation template returns a node n', DD will try to replace n with n'. For each node where the symbol *DEL* is returned, DD will try to delete n. After deciding on the replacements, the result is a new list of nodes for the current level. The helper function *applyTemplate* replaces the nodes on the level with the new nodes and returns the resulting tree to the main loop of the algorithm (lines 7 and 8).

BACKTRACKING-BASED SEARCH For templates that may return more than one candidate, the algorithm must decide not only whether to apply a candidate replacement but also which of the suggested candidate replacements to apply. This problem cannot be easily mapped to DD because DD assumes to have exactly one option per chunk (typically, whether to delete it or not). Instead, we present a backtracking-based algorithm that searches for a replacement of nodes on a particular level that reduces the overall tree. Similar to DD, the algorithm is a greedy search.

Algorithm 7.3 summarizes the main steps of the backtracking-based search for replacements of nodes on a particular level. The algorithm is called at line 10 of the main GTR algorithm. The central idea is to try different *configurations* that specify which replacements to use for each node. The algorithm starts with a configuration that replaces each node with itself (lines 1 to 3). Then, the algorithm iterates through all nodes (line 6) and tries all configurations where the replacement candidate is smaller than the currently chosen replacement. That is, the algorithm avoids invoking the oracle for replacements that are less effective than an already found replacement. If the oracle confirms that replacing a node preserves the property of interest,

Algorithm 7.3 Backtracking-based reduction of level nodes

Input: tree t, list of *nodes* on the same level, oracle o, template t
Output: reduced tree

```
    ▷ Maps each node to its current replacement:
 1: conf ← empty map
 2: for n ∈ nodes do
 3:     conf.put(n, n)
 4: repeat
 5:     improvementFound ← false
 6:     for n ∈ nodes do
 7:         currentRep ← conf.get(n)
 8:         for n′ ∈ l(n) where size(n′) < size(currentRep) do
 9:             t′ ← t with each n replaced by n′
10:             conf.put(n, n′)
11:             if oracle(t′) then
12:                 improvementFound ← true
13:                 currentRep ← n′
14:             else
15:                 conf.put(n, currentRep)                ▷ Backtrack
16: until ¬improvementFound
17: return t′
```

an improvement was found w.r.t. the current replacement (lines 12 and 13). Otherwise, the algorithm must revert the replacement and backtracks to the previous configuration (line 15).

The algorithm repeats the search for a replacement of any of the nodes on the current level until no further improvement is found. The reason for repeatedly considering the list of nodes is that using an effective replacement at a later node may enable using previously impossible replacements at previous nodes, which have already been tested in the current iteration. For example, consider the following input, where the `crash(b)` call ensures the property of interest:

```
a = b = 0;
if (a)
  crash(b);
```

During the first iteration of the main loop (lines 4 to 16), the algorithm cannot reduce the assignments in the first line but reduces the input by substituting the if-statement with the `crash(b)` call. Now, during the second iteration, the algorithm again considers the assignment statement and successfully reduces it to `b = 0`, which yields the following reduced input:

```
b = 0;
crash(b);
```

The search for a reduction of the nodes in the current level guarantees to find a local optimum, i.e., a configuration where using any other replacement that yields a smaller subtree would not satisfy the oracle. As the search is greedy, it may miss a configuration that yields a smaller overall tree satisfying the oracle. Searching for a global optimum would require to explore all possible configurations, which is exponential in the number of candidates suggested.

EXAMPLE We illustrate GTR on the running example. Recall that only line 2 of Figure 7.1 is relevant for reproducing the bug. The algorithm starts on level 0, which contains only the *IfStmt*, and invokes *applyTemplate* with the deletion template. Deletion returns at most one transformation. Therefore, the algorithm applies DD to the node on this level and tries to delete it with all its children. However, this deletion would make the bug disappear and is discarded by the oracle.

Next, *applyTemplate* is invoked with the substitute-by-child template. Since this template may return multiple candidates, the algorithm invokes the backtracking-based *reduceLevelNodes* function, i.e., Algorithm 7.3. There is only one node to consider in line 6 of Algorithm 7.3, and in line 8 three different candidates are tested. The first is the *UnaryExpr* on the left side. This candidate has a size of 4. But, since the important code piece is removed, line 15 reverts this change. The next candidate is the *BlockStatement* in the middle. It has a size of 7. The oracle returns true for this transformation, so *currentRep* is updated in line 13. The third candidate is the *BlockStatement* on the right. Since it also has a size of 7, which is not smaller than the size of *currentRep*, the candidate is not tested. Now that an improvement was found, the main loop (lines 4 to 16) is repeated. As there is only one node, nothing new will be tested. After having finished both templates on level 0, GTR will advance to level 1 and continue in the same manner.

7.4.4 *GTR* Algorithm*

The existing DD and HDD* algorithms guarantee 1-minimality and 1-tree-minimality, respectively. In essence, this property states that, given a reduced input, there is no single reduction step that can

Algorithm 7.4 GTR*

Input: tree t, oracle o, set $\mathcal{L}$ of templates
Output: 1-transformation-minimal tree

```
1: current ← t
2: repeat
3:     previous ← current
4:     current ← GTR(previous, o, L)
5: until size(previous) = size(current)
6: return current
```

further reduce the input. We define a similar minimality property for GTR:

Definition 7.5. *A tree t is called* 1-transformation-minimal *w.r.t. an oracle o and a set of templates $\mathcal{L}$ if $o(t) = true \wedge \forall n$ in t and $\forall l \in \mathcal{L}$, there is no candidate n' in $l(n)$ that, when replacing n with n' yields a tree t' with $o(t') = true \wedge size(t') < size(t)$.*

In other words, for 1-transformation-minimal trees, all trees obtained by single replacements of one node of the tree cause the oracle to return *false*. The main difference to the existing 1-minimality and 1-tree-minimality properties is to consider arbitrary tree transformations.

The GTR algorithm does not guarantee to find a 1-transformation-minimal tree. The reason is that by optimizing a tree on one level, a transformation on a higher level, which had been rejected by the oracle before, can become possible. To guarantee 1-transformation-minimality, we present a variant of GTR, the GTR* algorithm (Algorithm 7.4). GTR* repeats GTR until the tree does not change its size anymore, which indicates that no transformation can be applied thereafter.

7.4.5 *Generalization of HDD and HDD**

GTR and GTR* generalize the existing HDD and HDD* algorithms, respectively. To obtain HDD, we configure GTR to include only the deletion template, without specializing the template to a particular language. The resulting algorithm applies DD on every level of the input tree by deleting a subset of the nodes on this level. This behavior is exactly what HDD does, i.e., the reduced tree is the same as returned by HDD. This generalization also applies to HDD*, where

Format	Inputs	Bytes			Lines			Nodes		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
Python	7	212	483	1,574	19	22	73	82	232	591
JS	41	32	515	21,806	1	28	458	19	223	5,529
PDF	371	2,901	22,966	1,167,807	–	–	–	155	258	4,324
XML	10	7,225	29,065	51,180	170	500	888	319	1,042	1,823

Table 7.1: Input files used for the evaluation.

we simply run GTR* with the variant of GTR that is equivalent to HDD.

7.5 EVALUATION

We evaluate GTR by applying it to four input formats and usage scenarios, including reducing fault-inducing inputs for debugging, reducing malicious inputs for easier security analysis, and reducing test inputs for more efficient testing. The evaluation compares GTR and GTR* to the existing DD, HDD, and HDD* algorithms. We focus on three research questions:

- RQ1: How effective is the approach in reducing trees?
- RQ2: How efficient is the approach?
- RQ3: What are the effects of specializing transformation templates to an input format?

7.5.1 *Experimental Setup*

7.5.1.1 *Input Formats and Oracles*

We consider four sets of inputs that comprise a total of 429 input files that can be represented as a tree. Table 7.1 summarizes the inputs and shows their size in terms of bytes, lines, and number of tree nodes. For the binary PDF format we do not report lines.

FAILURE-INDUCING PYTHON CODE We use GTR to reduce Python files that cause the Python interpreter to crash. To obtain such files,

we search the Python bug tracker for segmentation faults and stack overflows reported along with code to reproduce it. Because these files have been reported by users or developers, they are likely to have been manually reduced, presenting a non-trivial challenge to any input reduction algorithm. We use a Python parser [196] to represent code as trees. The oracle to check whether a reduced Python file preserves the property of interest is to execute the file and to check the status code returned by the Python interpreter. Only checking the status code bears the risk of misclassification, e.g., if the program is altered to return that status code without triggering the bug. Given the low number of inputs, we could exclude this possibility manually for the given inputs. As a corpus to specialize the transformation templates, we gather 900 files from popular (measured by number of stars) GitHub projects.

INCONSISTENCY-EXPOSING JAVASCRIPT CODE We also use GTR to reduce JavaScript files that cause inconsistencies between browsers. The files are generated by TreeFuzz [165], an existing fuzz testing technique. We configure TreeFuzz to generate 3,000 files and keep all files that trigger a browser inconsistency, which results in 41 files. Since these files are automatically generated, they generally contain parts that are not required to trigger a browser inconsistency, providing a good data set to complement the manually written Python files. We use Esprima [191] to transform code to trees. As the oracle, we compare the runtime behavior of a JavaScript file in Firefox 25 and Chrome 48, as described in [165]. This oracle compares read and written values as well as error types and messages. The JavaScript corpus for specializing transformation templates comprises around 140,000 files [164].

MALICIOUS PDF DOCUMENTS As a usage scenario beyond reducing inputs for debugging, we use GTR to reduce malicious PDF files while preserving their maliciousness, which facilitates further security analysis. We download malicious PDF files from the Contagio malware dump [79]. PDF documents are binary data but have an internal tree structure. Using the pdfminer [125] and iTextPDF [205] libraries, we convert between PDFs and trees. We filter the PDFs to keep only those that are classified as malicious according to PDF Scrutinizer [89] and that are compatible with our tree conversion. Out of the over 8,000 remaining files we chose a random subset of 371 files.

As the oracle, we check whether PDF Scrutinizer classifies a file as malicious. This oracle may accept a malformed PDF and classify it as malicious. This behavior is desired, since PDF viewers try to display even malformed PDFs and could thus still execute harmful code. In contrast to the above formats, PDF trees have weaker constraints over their nodes, and the malicious content, contained in embedded objects, is typically not spread over the tree. The PDF corpus for specializing templates are 16,000 files from the Contagio malware dump, including both malicious and benign documents.

TEST SUITE OF XML FILES As another usage scenario, we use GTR for test suite reduction, i.e., reducing test cases while preserving the code coverage. We download a corpus of more than 140,000 XML files that adhere to the same XML document type definition (DTD) [192]. From the corpus, we select a random subset of 10 XML files and parse them using the xmllint [206] XML validation tool. Subsequently, we measure the coverage in xmllint using gcov [204]. As the oracle, the coverage in xmllint using a reduced XML file must be at least the original coverage. For XML there was no necessity to specialize the templates because both valid and invalid XML files are accepted by xmllint. Therefore, we omit the template specialization step.

7.5.1.2 *State of the Art Approaches*

We compare our approach to our own implementations of the existing DD, HDD, and HDD* algorithms. The DD implementation works on the line-level, i.e., each line of the input is a chunk considered by DD. The HDD implementation uses the same tree representation of the inputs as the GTR implementation.

7.5.2 *Effectiveness*

To evaluate the effectiveness in reducing test inputs, we apply GTR and GTR* to the inputs in Table 7.1. To measure effectiveness, we compute the remaining size relative to the original inputs, measured both in terms of the file size and the tree size. Since DD does not represent inputs as trees, we measure only the file size for DD-reduced inputs.

Table 7.2 summarizes the results. The table shows for each approach the remaining file sizes and nodes, along with the percentage of the

	DD	HDD	HDD*	GTR	GTR*
Failure-inducing Python code:					
Remaining file size	359 (74.3%)	437 (90.5%)	423 (87.6%)	301 (62.3%)	261 (54.0%)
Remaining nodes	-	175 (75.4%)	166 (71.6%)	105 (45.3%)	100 (43.1%)
Oracle invocations	125	809	1,089	205	492
Inconsistency-exposing JavaScript code:					
Remaining file size	84 (16.3%)	49 (9.5%)	49 (9.5%)	28 (5.4%)	28 (5.4%)
Remaining nodes	-	20 (9.0%)	20 (9.0%)	8 (3.6%)	8 (3.6%)
Oracle invocations	77	21	22	16	17
Malicious PDF documents:					
Remaining file size	17,225 (75.0%)	17,304 (75.3%)	17,304 (75.3%)	17,304 (75.3%)	17,304 (75.3%)
Remaining nodes	-	114 (44.2%)	114 (44.2%)	114 (44.2%)	114 (44.2%)
Oracle invocations	358	509	665	239	389
Test suite of XML files:					
Remaining file size	28,897 (99.4%)	1,259 (4.3%)	1,246 (4.3%)	1,271 (4.4%)	940 (3.2%)
Remaining nodes	-	21 (2.0%)	20 (1.9%)	14 (1.3%)	14 (1.3%)
Oracle invocations	1,746	92	107	100	114

Table 7.2: Effectiveness and efficiency of reduction by GTR and GTR* compared to baseline approaches. For each measure, the best approach is highlighted. We report the median values over all files of a data set.

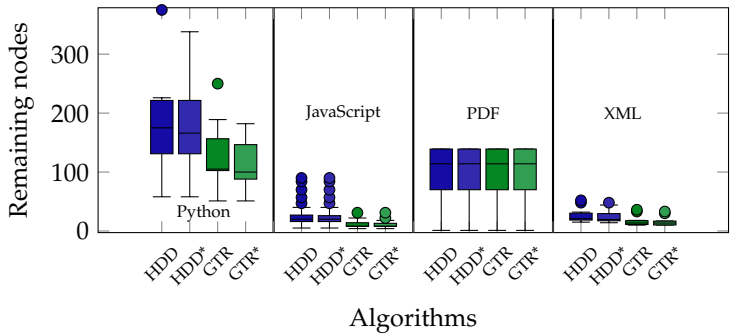


Figure 7.3: Effectiveness of reduction measured in terms of the number of the remaining tree nodes. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

original size. Each value is the median over all inputs we consider. The best approach for a particular measure and input format is highlighted. Overall, GTR* consistently yields the smallest remaining trees (closely followed by GTR), with 43.1%, 3.6%, 44.2%, and 1.3% of the original size for Python, JavaScript, PDFs, and XML files, respectively.

To better understand the variations in effectiveness across different inputs of a format, Figure 7.3 shows the distribution of remaining tree sizes. The figure shows that, even though the effectiveness varies across inputs, GTR and GTR* outperform the other approaches for most inputs.

We further discuss our results for the different formats:

- *Python*. For the Python data set, GTR* produces the smallest trees and GTR the second smallest. The relative reduction is not as high as for other formats. The reason is that these inputs have been reduced manually before reporting them to the Python developers, which leaves little room for any subsequently applied tree reduction algorithm.

- *JavaScript*. For the JavaScript data set, we observe larger reductions by all algorithms, sometimes removing more than 99% of the file. The main reason is that these files are generated by a fuzz tester and have not been processed by a human. GTR and GTR* consistently outperform all other algorithms.

- *PDF*. For the PDF data set, all tree-based algorithms are equally efficient, leaving only 44.2% of the nodes in a tree, on average. The reduced file size is about 75%, i.e., larger than the reduced tree size. The reason is that a few nodes in the tree representation of a PDF are large objects, such as images or embedded code, and that these large objects often contain the malicious content. Surprisingly, DD actually achieves marginally better file size reductions compared to the other algorithms. However, after examining these files manually, we noticed they were not valid PDF files anymore, even though PDF Scrutinizer still flags them as malicious. These syntactically invalid files would likely not help a security analyst that much. In contrast, removing more than half of a PDF's nodes is a vast improvement for a security analyst who manually inspects the file's content.

- *XML*. For the XML test suite, GTR and GTR* achieve the best reductions, sometimes removing up to 98% of the trees. DD is only able to reduce the XML files minimally. Our hypothesis is that when DD removes random lines, a malformed XML file result, which trigger only the error handling code of xmllint.

In summary, GTR and GTR* are more or as effective as the best existing input reduction approach with respect to the remaining tree size. Using GTR, the median percentage of nodes after reduction for four different input formats is 45.3%, 3.6%, 44.2%, and 1.3%, respectively.

7.5.3 *Efficiency*

We evaluate the efficiency of our approach by measuring the number of oracle invocations required to reduce a tree. Using this metric instead of, e.g., wall clock time, is motivated by two reasons. First, invoking the oracle typically is the most important operation during automated input reduction, because it often involves running a complex piece of software, such as a compiler or interpreter, on non-trivial inputs, such as large programs. Second, wall clock time is highly dependent on the implementation of the tree reduction and the oracle. To check that the number of oracle invocation is a meaningful measure, we compare the time spent in the oracle with the time in other parts of the algorithm for the JavaScript data set. For

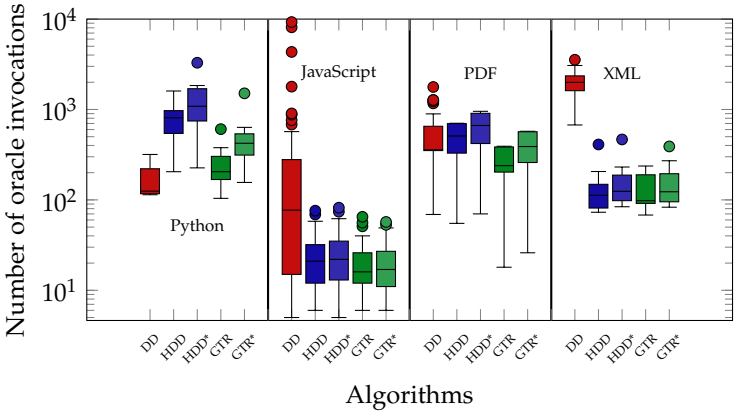


Figure 7.4: Number of oracle invocations. Note the logarithmic scale. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

each algorithm, the oracle invocation time dominates and comprises more than 98% of the total execution time, on average.

Table 7.2 shows the median number of oracle invocations required by the different approaches to reduce a single file. Figure 7.4 illustrates the distributions of this number for each input format. For three of the four formats, GTR and GTR* require fewer oracle invocations, i.e., are more efficient, than their counterparts HDD and HDD*. The GTR* and HDD* algorithms both need more invocations than their *-less counterparts, which is unsurprising because they run the algorithm, including oracle invocations, multiple times.

We next discuss the results for the different formats:

- *Python*. GTR needs 64% more invocations than DD but HDD needs 295% more invocations than GTR.
- *JavaScript*. GTR is the most efficient approach. HDD needs 31% more invocations, and DD 381% more invocations.
- *PDF*. GTR is the most efficient approach. HDD needs 113% more invocations, and DD 150% more invocations.
- *XML*. HDD is the most efficient approach. GTR needs 9% more invocations, and DD 1898% more invocations.

The large difference in the results for DD can be explained by the size of the input files. Since the Python files are relatively small and cannot be reduced as much as the relatively large JavaScript files, DD reduces them quickly. In contrast, the structure-unaware search of DD takes significantly more oracle invocations for larger files.

> To summarize, GTR is either more efficient or only slightly less efficient than the best existing approach.

7.5.4 *Benefits of Corpus-Based Filtering*

Our approach specializes language-independent transformation templates to a specific input language by learning filtering rules from a corpus of examples of inputs. To evaluate how the corpus-based filtering influences the effectiveness and efficiency of GTR, we compare the approach with a variant of GTR that does not filter any candidate transformations. For both variants, we perform the same experiments as described in Sections 7.5.2 and 7.5.3, except for the XML format, where we do not use any corpus.

We find that the GTR variant without filtering of candidates achieves the same effectiveness for JavaScript and PDF and slightly higher reductions (5%) for Python. The reason is that the corpus does not mirror all facets of the target languages, which may cause the filtering to overly constrain the transformations. For example, if the corpus would not contain any if-statement without an else-branch, then GTR would not consider removing the else-branch. Fortunately, the results show that such overly constrained filtering is very unlikely.

The GTR variant without filtering needs significantly more oracle invocations. For the Python data set, the variant needs 423 invocations (median), whereas the full GTR approach needs only 205 invocations. For the JavaScript data set, the results are 30 and 16 invocations without and with filtering, and for the PDF data set 611 and 239 invocations, respectively.

Finally, we measure how long extracting language-specific information from the corpus takes. In total, extracting this knowledge takes around 21 minutes, 19 seconds, and 15 seconds for the JavaScript, Python and PDF data sets, respectively.

In summary, comparing GTR with and without specialization transformations shows that both variants are roughly equally effective and that the specialization significantly improves the efficiency of the algorithm.

7.6 CONCLUSION

In this chapter, we present GTR, a novel algorithm to reduce tree-structured test inputs in a generalized and language-independent way. Our algorithm applies tree transformations hierarchically to reduce a given test input. The algorithm combines Delta Debugging and a greedy backtracking-based search to choose which transformations to apply. To specialize generic tree transformation templates to a particular input format, GTR automatically infers language-specific filters from a corpus of examples. We compare our approach with three existing algorithms, DD, HDD, and HDD*, on 429 test inputs. In three of four experiments, GTR outperforms other algorithms in reduction effectiveness. At the same time, GTR is either only slightly less or even more efficient than the best existing approach. We envision GTR to be applied to various problems that benefit from reduced inputs, e.g., to reduce bug-triggering inputs provided by users or fuzz testing techniques, to reduce test suites for more efficient test execution, or to reduce potentially malicious code or documents before a manual security analysis.

Part IV

RELATED WORK AND CONCLUSIONS

This parts provides the most closely related work, the conclusions and the possible future research directions.

8

RELATED WORK

The current chapter presents the most closely related work to this dissertation. We first provide some of the corpus-based analysis approaches (Section 8.1) that are related to this dissertation. The second part (Section 8.2) contains the related work that uses learning as the core approach to solve software engineering problems. In the remaining parts of this chapter, we list the related work on test synthesis (Section 8.3), usage of natural language in software engineering (Section 8.4), bug seeding (Section 8.5) and work that minimizes test inputs (Section 8.7).

8.1 CORPUS-BASED ANALYSIS OF SOURCE CODE

The fundamental theme of this dissertation is corpus-based analyses to solve software engineering problems. There is a large body of related work that uses corpus-based analyses, e.g., to find anomalies that correspond to bugs [65], for code completion [45], to recommend API usages [56], for plagiarism detection [18, 30], and to find copy-paste bugs [29]. Broadly, the corpus-based analysis can be divided into non-learning and learning-based approaches. This section provides the related work that does not use learning as the core approach to tackle the target problems mentioned above. The learning-based related work is mentioned in Section 8.2.

STATIC ANALYSIS Lint-like checkers such ESLint [268], JSHint [59], JSLint [62], Pyre [269], flake8 [270] are popular tools for JavaScript and Python that search for bad coding practices through lightweight static analysis. Other static analysis approaches extract call graphs [94] for IDE support. Some of the above approaches, e.g., ESLint warn about excessive use of global variables within a single file, but they do not

analyze conflicts across files or libraries in comparison to ConflictJS (Chapter 5). The static analysis based approaches mentioned above also do not reason about names and report warning about such inconsistencies that we present in NL2Type (Chapter 4) and Nalin (Chapter 6).

Some work related to ConflictJS statically analyzes library clients to understand types and other properties of a library [99], and search for code injection vulnerabilities [231]. Our work synthesizes library clients instead of analyzing existing clients. Our work differs from the above by analyzing multiple libraries and their potential interactions, instead of a single library.

DYNAMIC ANALYSIS A survey by Andreasen et al. [178] summarizes the dynamic analyses for JavaScript. Existing analyses include determinacy analysis [107], dynamic data race detectors [87, 106, 115, 139], dynamic model checkers [134], profilers to detect performance problems [131, 135, 199], taint and information-flow analyses [46, 57, 114], analyses to understand code changes [126], the root cause of a crash [163], find violations of common coding rules [132], type inconsistencies [142], check if a library implementation matches its interface specification [112] and to understand asynchronous behavior [146]. Similarly, for Python, Xu et al. [172] find bugs using program traces. All these techniques are orthogonal to ConflictJS (Chapter 5), which is the first to focuses on library conflicts and in contrast to Nalin (Chapter 6) which uses learning to find name-value inconsistencies.

MINING CODE PATTERNS Osman, Lungu, and Nierstrasz [123] describe an empirical study of frequent bug fixing code changes. Negara et al. [122] identify repetitive code changes from fine-grained sequences of code changes recorded in an IDE. Hanam, Brito, and Mesbah [155] extract recurring bug patterns for JavaScript. In contrast, SemSeed (Chapter 3) mines only concrete changes that correspond to a bug fix rather than any change made by a developer. Nguyen et al. [242] mine semantic change patterns by converting the correct and buggy files to program dependence graphs. Instead of a graph, in SemSeed, we leverage embeddings of tokens as the semantic representation and extract changes as a sequence of tokens. Kim et al. [96] manually inspect human-written patches to infer common fix patterns. SemSeed addresses the inverse problem of seeding bugs, instead of fixing them.

Code clone detection [13, 29, 34, 42, 168] relates to the semantic matching part of SemSeed (Section 3.3.2.3). 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.

TYPE CHECKING AND TYPE INFERENCE Several approaches address the lack of type annotations in dynamically typed languages by either inferring [49, 53, 69, 98] or checking types through static [27, 53, 60, 75], dynamic [142], or hybrid [82] analysis, which can help detecting otherwise missed bugs [186]. Hackett and Guo [82] use type inference in a JIT compiler to type-specialize the emitted machine code. TypeDevil [142] observes types at runtime and reports type inconsistencies as potential bugs. Because none of these approaches can guarantee to infer correct types for all values, lots of real-world JavaScript code still lacks type information. Our work in NL2Type (Chapter 4) addresses the problem by analyzing natural language elements instead of code. Similarly, in contrast to ConflictJS (Chapter 5), none of the approaches have been applied to multiple libraries. Another difference is that most type checkers focus on soundness and therefore suffer from false positives, whereas ConflictJS validates potential conflicts.

ANALYSIS OF LIBRARIES There exists many related work that analyze libraries and report interesting results such as the prevalence of JavaScript libraries, the presence of conflicts in libraries of other languages. Nikiforakis et al. [85] report that 88% of the websites include at least one remote library, and that libraries are loaded from over 300.000 unique URLs. Another study [193] shows that a web site includes a median of 9 and a maximum of 202 externally hosted scripts. These numbers illustrate the risk of accidental conflicts between libraries and need for approaches like ConflictJS (Chapter 5). Other studies investigate recurring performance bottlenecks [169], dynamic code loading [66, 80], insecure coding practices [55, 109], type coercions [143], type-related errors [185], the use of trivial software packages [175], the root causes of failures [105], and the use of callbacks [130]. Beyond JavaScript, Eshkevari et al. [111] report conflict-like problems in PHP applications and Pollux [160] determines the effects of upgrading a library.

GRAMMAR INFERENCE The problem of inferring a grammar has been studied for decades, as summarized in this book [61]. Existing approaches either infer grammars from examples, both examples included and not included in the language, and from an oracle that answers membership queries. More recent work infers and samples grammars to create new input data for fuzz testing [158, 180]. Bastani et al.[180] propose an approach that requires a set of input examples and blackbox access to the program. Instead, Höschele and Zeller [158] use dynamic taint analysis of the program under test. Both approaches infer a context-free grammar that approximates the structure of valid inputs. TreeFuzz (Chapter 2) shares the idea to infer a model of input data to create new data for testing. In contrast to the existing approaches, our work infers probabilistic models that express properties beyond the expressiveness of context-free grammars, e.g., ancestor constraints (Section 2.3.2.5) and identical subtree rules (Section 2.3.2.6).

Our corpus-based filtering in GTR (Chapter 7) relates to work on inferring grammars [41, 158, 180] and probabilistic models of structured program inputs in TreeFuzz. As an alternative to inferring language constraints from a corpus, GTR could reuse inferred grammars and models to prune candidate trees.

8.2 LEARNING APPROACHES ON SOURCE CODE

Hindle et al. [83] show that code is "natural" and therefore amenable to statistical language modeling. As a result, several statistical language models for programs have been proposed, e.g., based on n-grams [91, 104, 110], graphs [140, 144] and recurrent neural networks [124]. These models are useful for code completion [83, 91, 104, 124, 140, 149, 167, 221, 233, 249, 258], plagiarism detection [116], to deobfuscate code [210], and to infer appropriate identifier names [110, 144]. Other work model code changes and then makes predictions about them [250, 255], or trains models for program repair [189, 252], code search [216, 229] and apply bug fixes [245]. The current dissertation contributes to such recent stream of research that applies statistical modeling and machine learning to programs.

EMBEDDINGS OF CODE Embeddings of code are one important topic, e.g., using ASTs and AST paths [208, 210, 247], from program executions [202], control-flow graphs [215, 266], graph representation

of code [203], or a combination of token sequences and a graph representation of code [254]. Our encoder of variable names in NL2Type (Chapter 4), Nalin (Chapter 6) and SemSeed (Chapter 3) could benefit from being combined with an encoding of the code or natural language constructs surrounding the point of interest using those ideas.

CODE GENERATION PHOG [149] and Deep3 [167] learn a model to predict how to complete existing data, e.g., for code completion. They pick the model depending on the context of the prediction and automatically synthesize a function that extracts this context. For performance reasons, PHOG and Deep3 limit the search space of the synthesis, e.g., by not synthesizing functions with loops. As a result, these approaches cannot express some of the model extractors supported by TreeFuzz (Chapter 2), such as ancestor constraints (Section 2.3.2.5) and identical subtree rules (Section 2.3.2.6). Dnn4C, also used for code completion, is a neural model of code that learns not only from tokens, but also from syntactic and type information [221]. Other work uses source code models to predict parts of code [176]. TreeFuzz differs from all the above approaches by sampling probabilistic models for fuzz testing, i.e., by creating new data from scratch instead of predicting how to complete existing data. Furthermore, we introduce the idea to combine multiple statistical models into a single framework that can be easily extended with additional model extractors. Maddison and Tarlow propose a machine learning technique to generate "natural" source code [121]. TreeFuzz differs from their work by evaluating the usefulness of generated programs and by showing that our approach applies to tree data other than programs.

DETECT DEFECTS IN CODE Work by Godefroid et al. [187] proposes neural network-based learning of a statistical language model and how to sample such a model for fuzz testing. After learning a model from examples, their approach occasionally breaks out of the learned structure by choosing the least likely completion of partial data instead of the most likely. The goal of such surprising data is to stress-test the parsing component of the program under tests. In contrast, TreeFuzz (Chapter 2) aims at creating syntactically correct data that gets past the parser and tests subsequent components of the program under test. DeepBugs introduced learning-based and name-based bug detection [226], Wang, Liu, and Tan [171] for defect

prediction, Li et al. [219] to detect vulnerabilities, Gupta et al. [189] to detect and fix syntactic programming mistakes. In contrast to these approaches, TreeFuzz do not focus only on specific types of defect patterns and in comparison to Nalin (Chapter 6) by being purely static.

Type inference is useful for dynamically typed languages to detect otherwise missed bugs. Besides NL2Type (Chapter 4), we are aware of two other probabilistic type inference approaches for JavaScript: JSNice [144] and DeepTyper [218]. Section 4.4.4 discusses and compares with both approaches, showing that NL2Type outperforms both of them.

Neural attribute machines [177] are another neural network-based technique to learn and sample a probabilistic language model. Given an attribute grammar, which expresses rich structural constraints of the language, the neural network learns to respect these constraints. Similar to TreeFuzz, their work addresses the problem of enforcing language properties that go beyond the expressiveness of context-free grammars. In contrast to their work, we do not require an attribute grammar for each language, but instead infer constraints using language-independent model extractors.

LEARNING FROM EXECUTIONS Despite the recent surge of work on learning on code, learning on data gathered during executions is a relatively unexplored area. One of the few existing techniques is a model that embeds student programs based on dynamically observed input-output relations [141]. Wang et al.'s "blended" code embedding learning [265] combines runtime traces, which include values of multiple variables, and static code elements to learn a distributed vector representation of code. Beyond code embedding, BlankIt [261] uses a decision tree model trained on runtime data to predict the library functions that a code location may use. In contrast to these papers, Nalin (Chapter 6) addresses a different problem and feeds one value at a time into the model.

8.3 TEST SYNTHESIS

Fuzz testing has been used to test UNIX utilities [8], compilers [1–3, 10, 12, 81], runtime engines [39, 84, 137], refactoring engines [33], other kinds of applications [40], specific language features [23], and to find and exploit security vulnerabilities [25, 50, 68, 97]. Blackbox

fuzz testing either starts from existing data or generates new data based on a model that describes the required data format. For complex input formats, the model-based approach has the advantage that it avoids producing input data that is immediately rejected by the program. However, several authors mention the difficulties of creating an appropriate model for a particular target language [25, 81, 84], e.g., saying that "HTML is a good example of a complex file format for which it would be difficult to create a generator" [25]. TreeFuzz (Chapter 2) addresses this problem by inferring probabilistic, generative models of the data format.

Whitebox fuzz testing analyzes the program under test to generate inputs that cover not yet tested paths, e.g., using symbolic execution [40, 54], concolic execution [20, 26], or taint analysis [50]. In contrast, TreeFuzz is independent of a particular program under test and therefore trivially scales to complex programs. In particular for compiler testing, a recent survey by Chen et al. [251], and older surveys by Boujarwah and Saleh [11] and Kossatchev and Posypkin [22], the empirical comparison by Chen et al. [150] of different compiler testing approaches provide in detail the available related work to TreeFuzz.

On a related note, the test synthesis part of ConflictJS (Chapter 5) relates to generating test cases, such as feedback-directed, random test generation [35], symbolic and concolic execution [20, 26, 38], and search-based testing [74]. JSeft [138] exploits fixtures extracted from executions to create tests. These techniques could help the second phase of ConflictJS to further increase the percentage of validated behavior conflicts.

8.4 EXPLOITING NATURAL LANGUAGE FOR SOFTWARE ENGINEERING

As we explain in Chapter 6, identifier names are a means to convey the (intended) semantics of code. Because using meaningful identifier names is crucial for the understandability and maintainability of code, developers strive for names that express the value or behavior a name is bound to. The importance of meaningful names during programming has been studied and established [28, 58]. There are several techniques for finding poorly named program elements, e.g., based on pre-defined rules [43], by comparing method names against method bodies [51], and through a type inference-like analysis of

names and their occurrences [64]. To improve identifier names, rule-based expansion [78], n-gram models of code [110], and learning-based techniques that compare method bodies and method names have been proposed [239, 260]. In contrast to Nalin (Chapter 6), most of the above focuses on method names, whereas we target variables. Moreover, none of the existing work exploits dynamically observed values.

The perhaps most popular kind of name-based analysis is probabilistic type inference [173], often using deep neural network models [218, 248, 263, 267] that reason about the to-be-typed code. RefiNum uses names to identify conceptual types, which further refine the usual programming language types [214]. All of the above work is based on the observation that the implicit information embedded in identifiers is useful for program analyses. Nalin is the first to exploit this observation to find name-value inconsistencies.

PREDICTING NAMES When names are completely missing, e.g., in minified, compiled, or obfuscated code, learned models can predict them [144, 201, 211, 237]. Another line of work predicts method names given the body of a method [127, 148, 210], which beyond being potentially useful for developers serves as a pseudo-task to force a model to summarize code in a semantics-preserving way. Finally, DeepBugs uses natural language information in code, in particular identifier names, to detect code that is likely to be incorrect [226]. Nalin differs by considering values observed at runtime, and not only static source code, and by checking names for inconsistencies with the values they refer to, instead of predicting names from scratch.

NATURAL LANGUAGE VS. CODE Beyond natural language in the form of identifiers, comments and documentation associated with code are another valuable source of information. Prior work on analyzing comments focuses on finding inconsistencies between comments and code [36, 51, 90], on inferring executable specifications for a method [212], on identifying comments that have textual references to identifier names [197], on finding semantically similar verbs that occur in method names and method-level comments [95], and on finding redundant comments [220]. Nalin differs by focusing on variable names instead of comments, by comparing the natural language artifact against runtime values instead of static code, and by using a learning-based approach and to the best of our knowledge, NL2Type

(Chapter 4) is the first to predict types from comments. Another line of work uses natural language documentation to infer specifications of code [86, 153, 241], which is complementary to NL2Type and Nalin.

Code search allows developers to find code snippets through natural language queries [154, 174, 216, 229]. Similar to NL2Type, these approaches use embeddings of natural language words. Huo, Li, and Zhou [159] propose a neural network that predicts which file is buggy from a natural language bug report. Other approaches predict natural language information from source code, e.g., by predicting function name-like summaries for code snippets [148], or by de-obfuscating minified JavaScript code [144, 201, 211]. In contrast, NL2Type uses the available identifier names, along with comments, to make predictions.

8.5 BUG SEEDING

One approach to bug seeding is to apply mutations, e.g., based on a predefined set of transformation patterns [63, 117]. Brown et al. [183] propose to infer such patterns from code changes and Tufano et al. [246] uses a learning approach. 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. [244] 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 and 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 [183] and [246] is that our approach handles unbound tokens, seeding bugs even if this requires an application-specific identifier or literal.

Tailored mutation operators [103, 117, 147, 184, 194], 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 [147] or hard coded as in [103, 117].

A related work called IBIR by Khanfir et al. [257] also learns from past bugs how to seed new bugs. 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 [40, 170, 225, 235], automatically seeded bugs have been proposed for evaluating fuzz testing [151, 227]. 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.

8.6 BUG BENCHMARKS

Several bug benchmarks have been proposed, including SIR [19], Defects4J [118], BugSwarm [243] Bugbench [24], BegBunch [47], iBugs [32], ManyBugs [136], Codeflaws [200], DbgBench [181]. SemSeed complements such manually curated bug datasets by automatically seeding bugs into a target program.

8.7 MINIMIZING TEST INPUTS

Delta debugging (DD) [16] sets the foundations to automate the process of minimizing test inputs. In contrast to GTR presented in Chapter 7, it cannot handle large structured inputs effectively. A parallelization of DD has also been explored [157], which is orthogonal to our contributions. The hierarchical variant HDD [31] applies DD on hierarchical documents. However, HDD fails to restructure trees in a way that allows obtaining significantly smaller results, and also is less efficient than GTR. An improved HDD variant proposed later uses a different kind of grammars [156]. This targets the conversion of code documents to trees and is complementary to our findings.

C-Reduce [88] is a variant of DD that applies domain-specific transformations to reduce C code. These transformations include changing identifiers and constants, removing pairs of parentheses or curly braces, or inlining functions. One big advantage of C-Reduce is to never produce any input with non-deterministic or undefined behavior. There are two big differences in comparison to GTR. First, the transformations are source-to-source and not tree-to-tree. Second, C-Reduce loses generality by applying domain-specific changes to the document. However, many of the transformation included by C-Reduce can be expressed in a more general way and are included in our approach. For example, “removing an operator and one of its operands (e.g., changing a+b into a or b)” is equivalent to replacing the operator node with one of its children in the tree.

When the input to a program is not as complex as a code document itself, more efficient techniques can be employed. One possibility is to minimize the path constraints of the input that led to a particular failure [37]. However, the set of path constraints grows exponentially for more complex inputs, rendering this approach unfeasible for code inputs.

Another interesting approach taints parts of each input to identify the parts relevant to a failure [48]. The default setting taints each byte independently, making it possible to also account for complex inputs. At the same time, this disregards the structure of the document, similar to DD, and thus becomes overly expensive. Another setting tracks inputs on a per-entity basis, which is insufficient for test input reduction.

Various techniques aim at localizing a fault in the buggy program itself, instead of the input. Zeller applied DD also to the program with this goal [15]. There are various other approaches for fault localization [9, 17, 52].

Program slicing reduces a program while maintaining its behavior with respect to a particular variable [5]. Ideally, the smaller slice contains the bug and eases its localization. Dynamic slicing [7] focuses on the subset of the program that give a variable its value with the current input. Just slicing the variables that appear in the line causing the bug (if known) does not guarantee to obtain a program that produces the same buggy behavior, though. The combination of DD with dynamic forward and backward slicing has also been explored previously [21]. Another approach is to record traces to find shorter program executions with the observed buggy behavior [70, 77, 133], which ultimately also reveals likely locations for the bug. Fault localization and test input reduction have different goals. In a first step, a tester confronted with a failure needs a small (and fast running) input to reproduce the failure. In a second step, the bug must be located in the program and fixed. Finally, the small input can be turned into a regression test. Thus, both techniques complement each other.

Test input reduction is particularly important for automatically generated test inputs for example the JavaScript programs generated by TreeFuzz (Chapter 2). There are multiple approaches for such approaches also known as fuzz testing that we present in Section 8.3. These formats can be represented as a tree and may benefit from reduction via GTR.

9

CONCLUSIONS

The current dissertation presents six approaches, all of which use corpus-based analyses to either complement or improve state-of-the-art approaches to bug finding or input reduction. In the following, we provide a summary of the primary contributions of the thesis and provide some insights into possible future research directions.

9.1 CONTRIBUTIONS

Broadly, our contributions can be divided into three parts. First, we analyze source code examples to either train statistical models or extract code idioms. While the trained models are useful for generating new valid source code examples or for finding inconsistencies in code, the extracted code idioms are useful for mutating existing code to generate new realistic code. Second, we use runtime information by executing code corpora to either find bugs or train neural classifiers that can effectively predict inconsistencies between identifier names and the corresponding assigned values. Finally, we use corpus-based analysis to guide our input reduction approach that aid in debugging. We find our approach to be effective in reducing source code as well as other input formats that can be represented as a tree structure.

9.2 FUTURE RESEARCH DIRECTIONS

We envision the following potential research directions based on our thesis. In particular, we believe corpus-based reasoning can provide useful training data for machine learning-based models. The following highlights some of the most promising research directions that we believe to be natural extension to our thesis:

GENERATION OF REALISTIC EXAMPLES: A large body of research is focused on the applications of machine learning in solving software engineering problems [208]. This has been primarily driven by the availability of large corpora of code. For machine learning models that primarily deal with the classification of correct to incorrect code, the available code is used as correct or positive examples during training. The negative or incorrect examples are in most cases generated using some ad-hoc heuristics. We believe to make the trained models more effective, it is necessary to invest more research efforts into generating realistic code examples. Although in Chapter 3, we present one such approach and find that it improves upon a popular baseline, we believe there is ample room for improvement. For example, the current approach in SemSeed mutates code location first based on an abstracted token-based similarity. As a future work, a more generalized approach can be adopted that reasons about similarity not simply based on token level but possibly on code fragments.

HYBRID ANALYSIS FOR GENERATING TRAINING DATA: In this dissertation, we only use either data obtained from static or dynamic analysis of the code corpora. We believe training using a dataset obtained by the combination of both type of analysis is a promising research direction worth pursuing.

META-LEARNING ON CODE: Machine learning approaches are primarily data driven. As a result, most of the machine learning approaches on source code have been developed for popular programming languages such as Java, JavaScript (Chapter 2, Chapter 4), Python (Chapter 6). This can limit the ability of machine learning approaches to generalize for other programming languages. A future research direction worth pursuing is the application of meta-learning approaches for source code. For example, a model is trained using samples obtained from one programming language and is used for another.

EXPLAINABLE ARTIFICIAL INTELLIGENCE: The works presented in this dissertation, such as Nalin (Chapter 6) to classify a name-value pair as buggy or NL2Type (Chapter 4) to predict types, are considered as black boxes i.e., it is not possible to properly explain why any of these models take a particular decision. With the adoption of machine learning approaches for wide range of software engineering tasks

such as bug finding, code completion, program repair, it is needless to say that future research efforts are required to explain the decision mechanisms of the models involved. With explainability, developers will be more confident about the models and will boost adoption.

BIBLIOGRAPHY

[1] Richard L. Sauder. "A General Test Data Generator for COBOL." In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE '62 (Spring). San Francisco, California: ACM, 1962, pp. 317–323. DOI: 10.1145/1460833.1460869. URL: http://doi.acm.org/10.1145/1460833.1460869 (cit. on p. 196).

[2] K. V. Hanford. "Automatic Generation of Test Cases." In: *IBM Syst. J.* 9.4 (Dec. 1970), pp. 242–257. ISSN: 0018-8670. DOI: 10.1147/sj.94.0242. URL: http://dx.doi.org/10.1147/sj.94.0242 (cit. on p. 196).

[3] Paul Purdom. "A sentence generator for testing parsers." In: *Bit Numerical Mathematics* 12 (3 1972), pp. 366–375. DOI: 10.1007/BF01932308 (cit. on p. 196).

[4] J. C. King. "Symbolic Execution and Program Testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (cit. on p. 122).

[5] Mark Weiser. "Program slicing." In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 439–449 (cit. on p. 201).

[6] Iris Vessey. "Expertise in debugging computer programs: A process analysis." In: *International Journal of Man-Machine Studies* 23.5 (1985), pp. 459–494 (cit. on p. 163).

[7] Hiralal Agrawal and Joseph R Horgan. "Dynamic program slicing." In: *ACM SIGPLAN Notices*. Vol. 25. 6. ACM. 1990, pp. 246–256 (cit. on p. 201).

[8] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." In: *Communications of the ACM* 33.12 (1990), pp. 32–44 (cit. on p. 196).

[9] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. "Fault localization using execution slices and dataflow tests." In: *ISSRE*. Vol. 95. 1995, pp. 143–151 (cit. on p. 201).

[10] Colin J Burgess and M Saidi. "The automatic generation of test cases for optimizing Fortran compilers." In: *Information and Software Technology* 38.2 (1996), pp. 111–119 (cit. on p. 196).

[11] Abdulazeez S Boujarwah and Kassem Saleh. "Compiler test case generation methods: a survey and assessment." In: *Information and software technology* 39.9 (1997), pp. 617–625 (cit. on p. 197).

[12] William M. McKeeman. "Differential Testing for Software." In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107 (cit. on pp. 14, 196).

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670 (cit. on p. 193).

[14] Strategic Planning. "The economic impacts of inadequate infrastructure for software testing." In: *National Institute of Standards and Technology* (2002) (cit. on p. 1).

[15] Andreas Zeller. "Isolating cause-effect chains from computer programs." In: *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. ACM. 2002, pp. 1–10 (cit. on p. 201).

[16] Andreas Zeller and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200 (cit. on pp. 164, 166, 200).

[17] Manos Renieres and Steven P Reiss. "Fault localization with nearest neighbor queries." In: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE. 2003, pp. 30–39 (cit. on p. 201).

[18] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 76–85. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872770. URL: http://doi.acm.org/10.1145/872757.872770 (cit. on p. 191).

[19] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435 (cit. on p. 200).

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 213–223 (cit. on pp. 122, 197).

[21] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. "Locating faulty code using failure-inducing chops." In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM. 2005, pp. 263–272 (cit. on p. 201).

[22] A. S. Kossatchev and M. A. Posypkin. "Survey of Compiler Testing Methods." In: *Program. Comput. Softw.* 31.1 (Jan. 2005), pp. 10–19. ISSN: 0361-7688. DOI: 10.1007/s11086-005-0008-6. URL: http://dx.doi.org/10.1007/s11086-005-0008-6 (cit. on p. 197).

[23] Christian Lindig. "Random testing of C calling conventions." In: *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. 2005, pp. 3–12 (cit. on p. 196).

[24] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. "Bugbench: Benchmarks for evaluating bug detection tools." In: *Workshop on the evaluation of software defect detection tools*. Vol. 5. 2005 (cit. on p. 200).

[25] Peter Oehlert. "Violating Assumptions with Fuzzing." In: *IEEE Security & Privacy* 3.2 (2005), pp. 58–62 (cit. on pp. 196, 197).

[26] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, pp. 263–272 (cit. on p. 197).

[27] Peter Thiemann. "Towards a Type System for Analyzing JavaScript Programs." In: *European Symposium on Programming (ESOP)*. 2005, pp. 408–422 (cit. on p. 193).

[28] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. "What's in a Name? A Study of Identifiers." In: *International Conference on Program Comprehension (ICPC)*. IEEE, 2006, pp. 3–12 (cit. on p. 197).

[29] Zhenmin Li, S. Lu, S. Myagmar, and Yuanyuan Zhou. "CP-Miner: finding copy-paste and related bugs in large-scale software code." In: *Software Engineering, IEEE Transactions on* 32.3 (2006), pp. 176–192. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.28 (cit. on pp. 191, 193).

[30] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis." In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: ACM, 2006, pp. 872–881. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150522. URL: http://doi.acm.org/10.1145/1150402.1150522 (cit. on p. 191).

[31] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging." In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 142–151 (cit. on pp. 164, 166, 167, 171, 200).

[32] Valentin Dallmeier and Thomas Zimmermann. "Extraction of bug localization benchmarks from history." In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 433–436 (cit. on p. 200).

[33] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. "Automated testing of refactoring engines." In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 185–194 (cit. on pp. 13, 196).

[34] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. "Deckard: Scalable and accurate tree-based detection of code clones." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 96–105 (cit. on p. 193).

[35] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation."

In: *International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 75–84 (cit. on pp. 122, 197).

[36] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. "/*iComment: bugs or bad comments?*/." In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 2007, pp. 145–158 (cit. on p. 198).

[37] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. "Finding bugs in dynamic web applications." In: *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 261–272 (cit. on p. 201).

[38] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008, pp. 209–224 (cit. on pp. 122, 197).

[39] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing." In: *PLDI*. Vol. 43. ACM. 2008, pp. 206–215 (cit. on pp. 13, 196).

[40] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In: *Network and Distributed System Security Symposium (NDSS)*. 2008 (cit. on pp. 13, 14, 196, 197, 200).

[41] Zhiqiang Lin and Xiangyu Zhang. "Deriving input syntactic structure from execution." In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, pp. 83–93 (cit. on p. 194).

[42] Chanchal K Roy and James R Cordy. "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. 2008, pp. 172–181 (cit. on p. 193).

[43] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. "Lexicon bad smells in software." In: *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 95–99 (cit. on p. 197).

[44] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. 1st. O'Reilly Media, Inc., 2009. ISBN: 0596516495, 9780596516499 (cit. on p. 89).

[45] Marcel Bruch, Martin Monperrus, and Mira Mezini. "Learning from examples to improve code completion systems." In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 213–222 (cit. on p. 191).

[46] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. "Staged Information Flow for JavaScript." In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 50–62 (cit. on p. 192).

[47] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. "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. 2009, pp. 16–20 (cit. on p. 200).

[48] James Clause and Alessandro Orso. "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting." In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 249–260 (cit. on p. 201).

[49] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. "Static Type Inference for Ruby." In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 1859–1866. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529700. URL: http://doi.acm.org/10.1145/1529282.1529700 (cit. on pp. 76, 193).

[50] Vijay Ganesh, Tim Leek, and Martin C. Rinard. "Taint-based directed whitebox fuzzing." In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 2009, pp. 474–484 (cit. on pp. 14, 196, 197).

[51] Einar W. Høst and Bjarte M. Østvold. "Debugging Method Names." In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2009, pp. 294–317 (cit. on pp. 197, 198).

[52] Tom Janssen, Rui Abreu, and Arjan JC van Gemund. "Zoltar: A toolset for automatic fault localization." In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 662–664 (cit. on p. 201).

[53] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript." In: *Proc. 16th International Static Analysis Symposium (SAS)*. Vol. 5673. LNCS. Springer-Verlag, 2009 (cit. on pp. 76, 193).

[54] David Molnar, Xue Cong Li, and David A. Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs." In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 67–82. URL: http://dl.acm.org/citation.cfm?id=1855768.1855773 (cit. on p. 197).

[55] Chuan Yue and Haining Wang. "Characterizing insecure JavaScript practices on the web." In: *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*. 2009, pp. 961–970 (cit. on p. 193).

[56] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. "MAPO: Mining and Recommending API Usage Patterns." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2009, pp. 318–343 (cit. on p. 191).

[57] Thomas H. Austin and Cormac Flanagan. "Permissive dynamic information flow analysis." In: *PLAS*. 2010 (cit. on p. 192).

[58] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. "Exploring the Influence of Identifier Names on Code Quality: An Empirical Study." In: *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2010, pp. 156–165 (cit. on p. 197).

[59] Crockford. *JSHint, A Static Code Analysis Tool for JavaScript*. 2010. URL: https://jshint.com (cit. on pp. 1, 191).

[60] Phillip Heidegger and Peter Thiemann. "Recency Types for Analyzing Scripting Languages." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2010, pp. 200–224 (cit. on p. 193).

[61] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010 (cit. on p. 194).

[62] JSLint. *JSLint*. 2010. URL: http://www.jslint.com (cit. on p. 191).

[63] Yue Jia and Mark Harman. "An analysis and survey of the development of mutation testing." In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678 (cit. on p. 199).

[64] Julia L. Lawall and David Lo. "An automated approach for finding variable-constant pairing bugs." In: *International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 103–112 (cit. on p. 198).

[65] Martin Monperrus, Marcel Bruch, and Mira Mezini. "Detecting Missing Method Calls in Object-Oriented Software." In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2010, pp. 2–25 (cit. on p. 191).

[66] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. "JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications." In: *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*. 2010 (cit. on p. 193).

[67] Radim Rehurek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora." English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50 (cit. on p. 89).

[68] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications." In: *NDSS*. 2010 (cit. on pp. 13, 196).

[69] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. "Dynamic Inference of Static Types for Ruby." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 459–472. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926437. URL: http://doi.acm.org/10.1145/1926385.1926437 (cit. on pp. 76, 193).

[70] Martin Burger and Andreas Zeller. "Minimizing reproduction of software failures." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 221–231 (cit. on p. 201).

[71] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. "Improving the Tokenisation of Identifier Names." In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011, pp. 130–154 (cit. on p. 81).

[72] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. "Natural Language Processing (Almost) from Scratch." In: *Journal of Machine Learning Research* 12 (2011), pp. 2493–2537. URL: http://dl.acm.org/citation.cfm?id=2078186 (cit. on p. 77).

[73] ECMA. *Standard ECMA-262, ECMAScript Language Specification, 5.1 Edition*. European Computer Manufacturers Association (ECMA), 2011 (cit. on p. 109).

[74] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software." In: *SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 2011, pp. 416–419 (cit. on pp. 122, 197).

[75] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "Typing Local Control and State Using Flow Analysis." In: *European Symposium on Programming (ESOP)*. 2011, pp. 256–275 (cit. on p. 193).

[76] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing." In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678 (cit. on p. 45).

[77] Manu Jose and Rupak Majumdar. "Cause clue clauses: error localization using maximum satisfiability." In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 437–446 (cit. on p. 201).

[78] Dawn Lawrie and Dave Binkley. "Expanding identifiers to normalize source code vocabulary." In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2011, pp. 113–122 (cit. on p. 198).

[79] Mila Parkour. *Contagio Malware Dump*. 2011. URL: http://contagiodump.blogspot.de/2010/08/malicious-documents-archive-for.html (cit. on p. 180).

[80] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2011, pp. 52–78 (cit. on p. 193).

[81] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." In: *PLDI*. 2011, pp. 283–294 (cit. on pp. 13, 14, 28, 29, 32, 33, 196, 197).

[82] Brian Hackett and Shu-yu Guo. "Fast and Precise Hybrid Type Inference for JavaScript." In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 239–250. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254094. URL: http://doi.acm.org/10.1145/2254064.2254094 (cit. on p. 193).

[83] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. "On the naturalness of software." In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 837–847 (cit. on pp. 14, 77, 133, 194).

[84] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments." In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. Bellevue, WA: USENIX Association, 2012, pp. 38–38. URL: http://dl.acm.org/citation.cfm?id=2362793.2362831 (cit. on pp. 13, 33, 41, 196, 197).

[85] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "You are what you include: large-scale evaluation of remote JavaScript inclusions." In: *CCS*. 2012, pp. 736–747 (cit. on pp. 108, 193).

[86] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. "Inferring method specifications from natural language API descriptions." In: *2012 34th international conference on software engineering (ICSE)*. IEEE. 2012, pp. 815–825 (cit. on p. 199).

[87] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. "Race Detection for Web Applications." In: *Conference on Programming Language Design and Implementation (PLDI)*. 2012 (cit. on p. 192).

[88] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-case reduction for C compiler bugs." In: *ACM SIGPLAN Notices*. Vol. 47. 6. ACM. 2012, pp. 335–346 (cit. on pp. 164, 200).

[89] Florian Schmitt, Jan Gassen, and Elmar Gerhards-Padilla. "PDF Scrutinizer: Detecting JavaScript-based attacks in PDF documents." In: *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*. IEEE. 2012, pp. 104–111 (cit. on p. 180).

[90] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. "tcomment: Testing javadoc comments to detect comment-code inconsistencies." In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 260–269 (cit. on p. 198).

[91] Miltiadis Allamanis and Charles A. Sutton. "Mining source code repositories at massive scale using language modeling." In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. 2013, pp. 207–216 (cit. on p. 194).

[92] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. "Reversible debugging software." In: *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013) (cit. on p. 1).

[93] Amin Milani Fard and Ali Mesbah. "JSNOSE: Detecting JavaScript code smells." In: *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE. 2013, pp. 116–125 (cit. on pp. 2, 108).

[94] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Efficient construction of approximate call graphs for JavaScript IDE services." In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 752–761 (cit. on p. 191).

[95] Matthew J. Howard, Samir Gupta, Lori L. Pollock, and K. Vijay-Shanker. "Automatically mining software-based, semantically-similar words from comment-code mappings." In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. 2013, pp. 377–386 (cit. on p. 198).

[96] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. "Automatic patch generation learned from human-written patches." In: *International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811 (cit. on p. 192).

[97] Sebastian Lekies, Ben Stock, and Martin Johns. "25 million flows later: large-scale detection of DOM-based XSS." In: *ACM Conference on Computer and Communications Security*. 2013, pp. 1193–1204 (cit. on p. 196).

[98] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. "TeJaS: Retrofitting Type Systems for JavaScript." In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 1–16. ISBN: 978-1-4503-2433-5. DOI: 10.1145/2508168.2508170. URL: http://doi.acm.org/10.1145/2508168.2508170 (cit. on pp. 76, 193).

[99] Magnus Madsen, Benjamin Livshits, and Michael Fanning. "Practical static analysis of JavaScript applications in the presence of frameworks and libraries." In: *ESEC/SIGSOFT FSE*. 2013, pp. 499–509 (cit. on pp. 108, 192).

[100] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space." In: *CoRR* abs/1301.3781 (2013). arXiv: 1301.3781. URL: http://arxiv.org/abs/1301.3781 (cit. on p. 82).

[101] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. "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*. 2013, pp. 3111–3119 (cit. on pp. 55, 58, 145, 146).

[102] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. "Efficient JavaScript Mutation Testing." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 74–83 (cit. on pp. 46–48, 72).

[103] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. "Efficient JavaScript mutation testing." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 74–83 (cit. on p. 199).

[104] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. "A statistical semantic language model for source code." In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 2013, pp. 532–542 (cit. on pp. 14, 194).

[105] Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. "An Empirical Study of Client-Side JavaScript Bugs." In: *Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2013, pp. 55–64 (cit. on p. 193).

[106] Veselin Raychev, Martin Vechev, and Manu Sridharan. "Effective Race Detection for Event-Driven Programs." In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2013 (cit. on p. 192).

[107] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Dynamic determinacy analysis." In: *PLDI*. 2013, pp. 165–174 (cit. on p. 192).

[108] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript." In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2013, pp. 488–498 (cit. on pp. 117, 123).

[109] Sooel Son and Vitaly Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites." In: *NDSS*. 2013 (cit. on p. 193).

[110] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. "Learning natural coding conventions." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 2014, pp. 281–293 (cit. on pp. 194, 198).

[111] Laleh Eshkevari, Giuliano Antoniol, James R. Cordy, and Massimiliano Di Penta. "Identifying and Locating Interference Issues in PHP Applications: The Case of WordPress." In: *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: ACM, 2014, pp. 157–167. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597153. URL: http://doi.acm.org/10.1145/2597008.2597153 (cit. on p. 193).

[112] Asger Feldthaus and Anders Møller. "Checking correctness of TypeScript interfaces for JavaScript libraries." In: *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2014, pp. 1–16 (cit. on pp. 108, 192).

[113] Rahul Gopinath, Carlos Jensen, and Alex Groce. "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, 2014, pp. 189–200. DOI: 10.1109/ISSRE.2014.40. URL: https://doi.org/10.1109/ISSRE.2014.40 (cit. on p. 48).

[114] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. "JSFlow: Tracking information flow in JavaScript and its APIs." In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014, pp. 1663–1671 (cit. on p. 192).

[115] Shin Hong, Yongbae Park, and Moonzoo Kim. "Detecting Concurrency Errors in Client-Side Java Script Web Applications." In: *ICST*. 2014, pp. 61–70 (cit. on p. 192).

[116] Chun-Hung Hsiao, Michael J. Cafarella, and Satish Narayanasamy. "Using web corpus statistics for program analysis." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 2014, pp. 49–65 (cit. on p. 194).

[117] René Just. "The Major mutation framework: Efficient and scalable mutation analysis for Java." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 433–436 (cit. on pp. 46, 47, 72, 199).

[118] René Just, Darioush Jalali, and Michael D. Ernst. "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*. 2014, pp. 437–440 (cit. on pp. 46, 200).

[119] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 148).

[120] Quoc Le and Tomas Mikolov. "Distributed representations of sentences and documents." In: *International conference on machine learning*. PMLR. 2014, pp. 1188–1196 (cit. on p. 56).

[121] Chris J. Maddison and Daniel Tarlow. "Structured Generative Models of Natural Source Code." In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. 2014, pp. 649–657 (cit. on p. 195).

[122] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. "Mining fine-grained code changes to detect unknown change patterns." In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 803–813 (cit. on p. 192).

[123] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. "Mining frequent bug-fix code changes." In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 343–347 (cit. on p. 192).

[124] Veselin Raychev, Martin T. Vechev, and Eran Yahav. "Code completion with statistical language models." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, p. 44 (cit. on p. 194).

[125] Yusuke Shinyama. *PDFMiner*. 2014. URL: https://euske.github.io/pdfminer/ (cit. on p. 180).

[126] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Hybrid DOM-Sensitive Change Impact Analysis for JavaScript." In: *ECOOP*. 2015, pp. 321–345 (cit. on p. 192).

[127] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. "Suggesting accurate method and class names." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, pp. 38–49 (cit. on p. 198).

[128] François Chollet et al. *Keras*. https://keras.io. 2015 (cit. on p. 90).

[129] ECMA. *Standard ECMA-262, ECMAScript Language Specification, 6th Edition*. European Computer Manufacturers Association (ECMA), 2015 (cit. on p. 110).

[130] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. "Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript." In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. 2015, pp. 247–256 (cit. on p. 193).

[131] Liang Gong, Michael Pradel, and Koushik Sen. "JITProf: Pinpointing JIT-unfriendly JavaScript Code." In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2015, pp. 357–368 (cit. on p. 192).

[132] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. "DLint: Dynamically Checking Bad Coding Practices in JavaScript." In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2015, pp. 94–105 (cit. on pp. 2, 108, 192).

[133] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. "On the use of delta debugging to reduce recordings and facilitate debugging of web applications." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 333–344 (cit. on p. 201).

[134] Casper Svenning Jensen, Anders Møller, Veselin Raychev, and Martin Vechev. "Stateless Model Checking of Event-Driven Applications." In: *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 2015 (cit. on p. 192).

[135] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. "MemInsight: platform-independent memory debugging for JavaScript." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015,*

Bergamo, Italy, August 30 - September 4, 2015. 2015, pp. 345–356 (cit. on p. 192).

[136] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. "The ManyBugs and IntroClass benchmarks for automated repair of C programs." In: *IEEE Transactions on Software Engineering* 41.12 (2015), pp. 1236–1256 (cit. on p. 200).

[137] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. "Many-core compiler fuzzing." In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 65–76 (cit. on p. 196).

[138] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. "JSEFT: Automated Javascript Unit Test Generation." In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–10 (cit. on p. 197).

[139] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. "Detecting JavaScript Races that Matter." In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2015 (cit. on p. 192).

[140] Anh Tuan Nguyen and Tien N. Nguyen. "Graph-Based Statistical Language Model for Code." In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015, pp. 858–868 (cit. on pp. 14, 194).

[141] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. "Learning Program Embeddings to Propagate Feedback on Student Code." In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, pp. 1093–1102. URL: http://proceedings.mlr.press/v37/piech15.html (cit. on p. 196).

[142] Michael Pradel, Parker Schuh, and Koushik Sen. "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript." In: *International Conference on Software Engineering (ICSE)*. 2015 (cit. on pp. 2, 108, 192, 193).

[143] Michael Pradel and Koushik Sen. "The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2015 (cit. on p. 193).

[144] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting Program Properties from "Big Code"." In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 111–124. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2677009. URL: http://doi.acm.org/10.1145/2676726.2677009 (cit. on pp. 77, 78, 89, 90, 93, 94, 133, 194, 196, 198, 199).

[145] Duyu Tang, Bing Qin, and Ting Liu. "Document Modeling with Gated Recurrent Neural Network for Sentiment Classification." In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*. 2015, pp. 1422–1432. URL: http://aclweb.org/anthology/D/D15/D15-1167.pdf (cit. on p. 84).

[146] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Understanding asynchronous interactions in full-stack JavaScript." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 1169–1180 (cit. on p. 192).

[147] Miltiadis Allamanis, Earl T Barr, René Just, and Charles Sutton. "Tailored mutants fit bugs better." In: *arXiv preprint arXiv:1611.02516* (2016) (cit. on p. 199).

[148] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code." In: *ICML*. 2016, pp. 2091–2100 (cit. on pp. 198, 199).

[149] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. "PHOG: Probabilistic Model for Code." In: *ICML*. 2016, pp. 2933–2942 (cit. on pp. 14, 194, 195).

[150] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. "An Empirical Comparison of Compiler Testing Techniques." In: *ICSE*. 2016 (cit. on p. 197).

[151] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. "LAVA: Large-Scale Automated Vulnerability Addition." In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 2016, pp. 110–121 (cit. on pp. 45, 47, 48, 200).

[152] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 1050–1059. URL: http://proceedings.mlr.press/v48/gal16.html (cit. on p. 89).

[153] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. "Automatic generation of oracles for exceptional behaviors." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 213–224 (cit. on p. 199).

[154] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. "Deep API Learning." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 631–642. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950334. URL: http://doi.acm.org/10.1145/2950290.2950334 (cit. on p. 199).

[155] Quinn Hanam, Fernando Santos De Mattos Brito, and Ali Mesbah. "Discovering bug patterns in JavaScript." In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 144–156 (cit. on p. 192).

[156] Renáta Hodován and Ákos Kiss. "Modernizing hierarchical delta debugging." In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM. 2016, pp. 31–37 (cit. on p. 200).

[157] Renáta Hodován and Akos Kiss. "Practical Improvements to the Minimizing Delta Debugging Algorithm." In: *Proceedings of the 11th International Joint Conference on Software Technologies*. 2016, pp. 241–248 (cit. on p. 200).

[158] Matthias Höschele and Andreas Zeller. "Mining input grammars from dynamic taints." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 2016, pp. 720–725 (cit. on pp. 15, 194).

[159] Xuan Huo, Ming Li, and Zhi-Hua Zhou. "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. 2016, pp. 1606–1612 (cit. on p. 199).

[160] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. "POLLUX: safely upgrading dependent application libraries." In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 290–300 (cit. on p. 193).

[161] Ji Young Lee and Franck Dernoncourt. "Sequential Short-Text Classification with Recurrent and Convolutional Neural Networks." In: *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*. 2016, pp. 515–520. URL: http://aclweb.org/anthology/N/N16/N16-1062.pdf (cit. on p. 84).

[162] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. "Recurrent Neural Network for Text Classification with Multi-Task Learning." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. 2016, pp. 2873–2879. URL: http://www.ijcai.org/Abstract/16/408 (cit. on p. 84).

[163] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. "Feedback-directed instrumentation for deployed JavaScript applications." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 899–910 (cit. on p. 192).

[164] Multiple authors. *Learning from "Big Code"*. 2016. URL: http://learnbigcode.github.io/datasets/ (cit. on p. 180).

[165] Jibesh Patra and Michael Pradel. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Tech. rep. TUD-CS-2016-14664. TU Darmstadt, Department of Computer Science, 2016 (cit. on pp. 8, 180).

[166] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. "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*. 2016, pp. 761–774 (cit. on pp. 61, 69, 90).

[167] Veselin Raychev, Pavol Bielik, and Martin Vechev. "Probabilistic Model for Code with Decision Trees." In: *OOPSLA*. 2016 (cit. on pp. 14, 194, 195).

[168] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. "SourcererCC: Scaling code clone detection to big-code." In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1157–1168 (cit. on p. 193).

[169] Marija Selakovic and Michael Pradel. "Performance Issues and Optimizations in JavaScript: An Empirical Study." In: *International Conference on Software Engineering (ICSE)*. 2016, pp. 61–72 (cit. on p. 193).

[170] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "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*. 2016 (cit. on p. 200).

[171] Song Wang, Taiyue Liu, and Lin Tan. "Automatically learning semantic features for defect prediction." In: *ICSE*. 2016, pp. 297–308 (cit. on pp. 46, 195).

[172] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. "Python predictive analysis for bug detection." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 121–132 (cit. on pp. 2, 192).

[173] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. "Python probabilistic type inference with natural language support." In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 607–618. DOI: 10.1145/2950290.2950343. URL: https://doi.org/10.1145/2950290.2950343 (cit. on p. 198).

[174] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. "From word embeddings to document similarities for improved information retrieval in software engineering." In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 404–415 (cit. on p. 199).

[175] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm." In: *FSE*. 2017 (cit. on p. 193).

[176] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs." In: *CoRR* abs/1711.00740 (2017). arXiv: 1711.00740. URL: http://arxiv.org/abs/1711.00740 (cit. on pp. 90, 195).

[177] M. Amodio, S. Chaudhuri, and T. Reps. "Neural Attribute Machines for Program Generation." In: *ArXiv e-prints* (May 2017). arXiv: 1705.09231 [cs.AI] (cit. on p. 196).

[178] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. "A Survey of Dynamic Analysis and Test Generation for JavaScript." In: *ACM Computing Surveys* (2017) (cit. on pp. 1, 108, 192).

[179] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. "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, 2017. URL: https://openreview.net/forum?id=SyK00v5xx (cit. on p. 56).

[180] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing Program Input Grammars." In: *PLDI*. 2017 (cit. on pp. 14, 15, 32, 194).

[181] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. "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*. 2017, pp. 117–128 (cit. on p. 200).

[182] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. "Enriching Word Vectors with Subword Information." In: *TACL* 5 (2017), pp. 135–146. URL: https://transacl.org/ojs/index.php/tacl/article/view/999 (cit. on pp. 46, 55, 61, 138, 145).

[183] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. "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*. 2017, pp. 511–522 (cit. on pp. 47, 48, 54, 57, 63, 66, 199).

[184] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. "Assessment of class mutation operators for C++ with the MuCPP mutation system." In: *Information and Software Technology* 81 (2017), pp. 169–184 (cit. on p. 199).

[185] Zheng Gao, Christian Bird, and Earl T. Barr. "To type or not to type: Quantifying detectable bugs in JavaScript." In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 758–769 (cit. on p. 193).

[186] Zheng Gao, Christian Bird, and Earl T. Barr. "To type or not to type: quantifying detectable bugs in JavaScript." In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75. URL: https://doi.org/10.1109/ICSE.2017.75 (cit. on p. 193).

[187] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: machine learning for input fuzzing." In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, pp. 50–59 (cit. on pp. 14, 195).

[188] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. "On Calibration of Modern Neural Networks." In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, 2017, pp. 1321–1330. URL: http://proceedings.mlr.press/v70/guo17a.html (cit. on p. 88).

[189] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. "DeepFix: Fixing Common C Language Errors by Deep Learning." In: *AAAI 2017*. 2017 (cit. on pp. 194, 196).

[190] Satia Herfert, Jibesh Patra, and Michael Pradel. "Automatically Reducing Tree-Structured Test Inputs." In: *ASE*. 2017 (cit. on p. 8).

[191] Ariya Hidayat. *Esprima*. 2017. URL: http://esprima.org/ (cit. on pp. 34, 180).

[192] Hindawi. *Hindawi XML Corpus*. 2017. URL: https://www.hindawi.com/corpus/ (cit. on p. 181).

[193] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web." In: *NDSS*. 2017 (cit. on pp. 105, 108, 193).

[194] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. "Enabling mutation testing for android apps." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 233–244 (cit. on p. 199).

[195] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. "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*. 2017, pp. 438–449 (cit. on pp. 50, 145).

[196] Python Software Foundation. *Python AST library*. 2017. URL: https://docs.python.org/2/library/ast.html (cit. on p. 180).

[197] Inderjot Kaur Ratol and Martin P. Robillard. "Detecting fragile comments." In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, pp. 112–122 (cit. on p. 198).

[198] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. "Detecting Argument Selection Defects." In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2017 (cit. on pp. 52, 73).

[199] Marija Selakovic, Thomas Glaser, and Michael Pradel. "An Actionable Performance Profiler for Optimizing the Order of Evaluations." In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2017, pp. 170–180 (cit. on p. 192).

[200] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. "Codeflaws: a programming competition benchmark for evaluating automated program repair tools." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 180–182 (cit. on p. 200).

[201] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. "Recovering clear, natural identifiers from obfuscated JS names." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 2017, pp. 683–693 (cit. on pp. 133, 198, 199).

[202] Ke Wang, Rishabh Singh, and Zhendong Su. "Dynamic Neural Program Embedding for Program Repair." In: *CoRR* abs/1711.07163 (2017). URL: http://arxiv.org/abs/1711.07163 (cit. on p. 194).

[203] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection." In: *CCS*. 2017, pp. 363–376 (cit. on p. 195).

[204] gcov. *gcov – A Test Coverage Program*. 2017. URL: https://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Gcov.html (cit. on p. 181).

[205] iText Group NV. *iText PDF*. 2017. URL: http://itextpdf.com/ (cit. on p. 180).

[206] xmllint. *The XML C parser and toolkit of Gnome*. 2017. URL: http://xmlsoft.org/xmllint.html (cit. on p. 181).

[207] Miltiadis Allamanis. "The Adverse Effects of Code Duplication in Machine Learning Models of Code." In: *arXiv preprint arXiv:1812.06469* (2018) (cit. on p. 69).

[208] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness." In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 81 (cit. on pp. 194, 204).

[209] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs." In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018. URL: https://openreview.net/forum?id=BJOFETxR- (cit. on p. 135).

[210] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "A General Path-Based Representation for Predicting Program Properties." In: *PLDI*. 2018 (cit. on pp. 50, 145, 194, 198).

[211] Rohan Bavishi, Michael Pradel, and Koushik Sen. "Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts." In: *CoRR* arXiv:1809.05193 (2018) (cit. on pp. 133, 198, 199).

[212] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. "Translating code comments to procedure specifications." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 2018, pp. 242–253 (cit. on p. 198).

[213] DARPA CGC. *Darpa Cyber Grand Challenge (CGC) Binaries*. https://github.com/CyberGrandChallenge/. 2018 (cit. on p. 45).

[214] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. "RefiNym: Using Names to Refine Types." In: *ESEC/FSE*. 2018 (cit. on p. 198).

[215] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. "Path-based function embedding and its application to error-handling specification mining." In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 423–433 (cit. on p. 194).

[216] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search." In: *ICSE*. 2018 (cit. on pp. 194, 199).

[217] Andrew Habib and Michael Pradel. "How Many of All Bugs Do We Find? A Study of Static Bug Detectors." In: *ASE*. 2018 (cit. on p. 45).

[218] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. "Deep Learning Type Inference." In: *Proceedings of the 2018 12th Joint Meeting on Foundations of Software Engineering*. ACM, 2018 (cit. on pp. 77, 78, 89, 90, 93–95, 133, 196, 198).

[219] Zhen Li, Shouhuai Xu Deqing Zou and, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection." In: *NDSS*. 2018 (cit. on pp. 46, 90, 196).

[220] Annie Louis, Santanu Kumar Dash, Earl T Barr, and Charles Sutton. "Deep learning to detect redundant method comments." In: *arXiv preprint arXiv:1806.04616* (2018) (cit. on p. 198).

[221] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. "A Deep Neural Network Language Model with Contexts for Source Code." In: *SANER*. 2018 (cit. on pp. 194, 195).

[222] John-Paul Ore, Sebastian G. Elbaum, Carrick Detweiler, and Lambros Karkazis. "Assessing the type annotation burden." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018, pp. 190–201. DOI: 10.1145/3238147.3238173. URL: https://doi.org/10.1145/3238147.3238173 (cit. on p. 76).

[223] Md. Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "Building Language Models for Text with Named Entities." In: *Proceedings of the 56th Annual Meeting*

of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers. 2018, pp. 2373–2383. DOI: 10.18653/v1/P18-1221. URL: https://www.aclweb.org/anthology/P18-1221/ (cit. on p. 145).

[224] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. "ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries." In: *ICSE*. 2018, pp. 741–751 (cit. on p. 8).

[225] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "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*. 2018, pp. 697–710 (cit. on p. 200).

[226] Michael Pradel and Koushik Sen. "DeepBugs: A learning approach to name-based bug detection." In: *PACMPL* 2.OOPSLA (2018), 147:1–147:25. URL: https://doi.org/10.1145/3276517 (cit. on pp. 46, 48, 50, 68, 90, 133, 135, 145, 157, 195, 198).

[227] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. "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*. 2018, pp. 224–234 (cit. on pp. 47, 48, 200).

[228] Adam Rule, Aurélien Tabard, and James D Hollan. "Exploration and explanation in computational notebooks." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12 (cit. on pp. 134, 149).

[229] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. "Retrieval on source code: a neural code search." In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM. 2018, pp. 31–41 (cit. on pp. 194, 199).

[230] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. "Bugs. jar: a large-scale, diverse dataset of real-world java bugs." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 10–13 (cit. on p. 46).

[231] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. "Understanding and Automatically Preventing Injection Attacks on Node.js." In: *Network and Distributed System Security Symposium (NDSS)*. 2018 (cit. on p. 192).

[232] The Linux Foundation. *2018 Node.js User Survey Report*. May 2018 (cit. on p. 75).

[233] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. "code2seq: Generating Sequences from Structured Representations of Code." In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019. URL: https://openreview.net/forum?id=H1gKYo09tX (cit. on p. 194).

[234] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. "Getafix: Learning to Fix Bugs Automatically." In: *OOPSLA*. 2019, 159:1–159:27 (cit. on p. 46).

[235] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain." In: *IEEE Trans. Software Eng.* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841. URL: https://doi.org/10.1109/TSE.2017.2785841 (cit. on p. 200).

[236] Rafael-Michael Karampatsis and Charles A. Sutton. "How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset." In: *CoRR* abs/1905.13334 (2019). arXiv: 1905.13334. URL: http://arxiv.org/abs/1905.13334 (cit. on pp. 52, 69, 73).

[237] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. "DIRE: A Neural Approach to Decompiled Identifier Naming." In: *ASE*. 2019 (cit. on p. 198).

[238] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. "Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks." In: *OOPSLA*. 2019 (cit. on pp. 46, 135).

[239] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. "Learning to spot and refactor inconsistent method names." In: *Proceedings of the 41st International Conference on*

Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. 2019, pp. 1–12. URL: https://dl.acm.org/citation.cfm?id=3339507 (cit. on p. 198).

[240] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. "NL2Type: Inferring JavaScript function types from natural language information." In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 304–315. DOI: 10.1109/ICSE.2019.00045. URL: https://doi.org/10.1109/ICSE.2019.00045 (cit. on pp. 8, 50, 133, 145).

[241] Manish Motwani and Yuriy Brun. "Automatically generating precise Oracles from structured natural language specifications." In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 188–199. DOI: 10.1109/ICSE.2019.00035. URL: https://doi.org/10.1109/ICSE.2019.00035 (cit. on p. 199).

[242] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. "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. 2019, pp. 819–830 (cit. on p. 192).

[243] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 339–349 (cit. on p. 200).

[244] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. "On learning meaningful code changes via neural machine translation." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 25–36 (cit. on pp. 53, 199).

[245] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. "An empirical study on learning bug-fixing patches in the wild via neural machine translation." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.4 (2019), pp. 1–29 (cit. on p. 194).

[246] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. "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, 2019, pp. 301–312. DOI: 10.1109/ICSME.2019.00046. URL: https://doi.org/10.1109/ICSME.2019.00046 (cit. on pp. 47, 48, 52, 57, 66, 199).

[247] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. "A Novel Neural Source Code Representation based on Abstract Syntax Tree." In: *ICSE*. 2019 (cit. on p. 194).

[248] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. "Typilus: Neural Type Hints." In: *PLDI*. 2020 (cit. on pp. 133, 147, 198).

[249] Gareth Ari Aye and Gail E. Kaiser. "Sequence Model Design for Code Completion in the Modern IDE." In: *CoRR* abs/2004.05249 (2020). arXiv: 2004.05249. URL: https://arxiv.org/abs/2004.05249 (cit. on p. 194).

[250] Shaked Brody, Uri Alon, and Eran Yahav. "A Structural Model for Contextual Code Changes." In: *OOPSLA*. 2020 (cit. on p. 194).

[251] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. "A Survey of Compiler Testing." In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3363562. URL: https://doi.org/10.1145/3363562 (cit. on pp. 8, 197).

[252] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. "Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs." In: *ICLR*. 2020 (cit. on pp. 135, 194).

[253] Aryaz Eghbali and Michael Pradel. "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, 2020, pp. 956–967. URL: https://ieeexplore.ieee.org/document/9286132 (cit. on pp. 52, 73).

[254] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. "Global Relational Models of Source Code." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=B1lnbRNtwr (cit. on p. 195).

[255] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. "CC2Vec: Distributed Representations of Code Changes." In: *ICSE*. 2020 (cit. on p. 194).

[256] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. "Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code." In: *ICSE*. 2020 (cit. on p. 55).

[257] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "IBIR: Bug Report driven Fault Injection." In: *CoRR* abs/2012.06506 (2020). arXiv: 2012.06506. URL: https://arxiv.org/abs/2012.06506 (cit. on p. 199).

[258] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. "Code Prediction by Feeding Trees to Transformers." In: *arXiv preprint arXiv:2003.13848* (2020) (cit. on p. 194).

[259] Yi Li, Shaohua Wang, and Tien N. Nguyen. "DLFix: Context-based Code Transformation Learning for Automated Program Repair." In: *ICSE*. 2020 (cit. on p. 46).

[260] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. "Suggesting Natural Method Names to Check Name Consistencies." In: *ICSE*. 2020 (cit. on p. 198).

[261] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. "BlankIt library debloating: getting what you want instead of cutting what you don't." In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 164–180. DOI: 10.1145/3385412.3386017. URL: https://doi.org/10.1145/3385412.3386017 (cit. on p. 196).

[262] Michael Pradel and Satish Chandra. "Neural Software Analysis." In: *CoRR* abs/2011.07986 (2020). arXiv: 2011.07986. URL: https://arxiv.org/abs/2011.07986 (cit. on p. 136).

[263] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. "TypeWriter: Neural Type Prediction with Search-based Validation." In: *FSE*. 2020 (cit. on pp. 133, 198).

[264] Jiawei Wang, KUO Tzu-Yang, Li Li, and Andreas Zeller. "Assessing and Restoring Reproducibility of Jupyter Notebooks." In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 138–149 (cit. on p. 150).

[265] Ke Wang and Zhendong Su. "Blended, precise semantic program embeddings." In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 121–134. DOI: 10.1145/3385412.3385999. URL: https://doi.org/10.1145/3385412.3385999 (cit. on pp. 136, 196).

[266] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. "Learning Semantic Program Embeddings with Graph Interval Neural Network." In: *OOPSLA*. 2020 (cit. on pp. 135, 194).

[267] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. "LambdaNet: Probabilistic Type Inference using Graph Neural Networks." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=Hkx6hANtwH (cit. on pp. 133, 198).

[268] ESLint. *Find and fix problems in your JavaScript code*. 2021. URL: https://eslint.org/ (cit. on pp. 1, 191).

[269] Facebook. *Pyre*. 2021. URL: https://pyre-check.org (cit. on pp. 1, 191).

[270] Flake8. *Flake8 Your Tool For Style Guide Enforcement*. 2021. URL: https://flake8.pycqa.org (cit. on pp. 1, 191).

[271] Yaza Wainakh, Moiz Rauf, and Michael Pradel. "IdBench: Evaluating Semantic Representations of Identifier Names in Source Code." In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2021 (cit. on pp. 46, 56, 138, 145).

[272] *CDNJS*. https://cdnjs.com/. Accessed: 2018-07-22 (cit. on p. 90).

[273] *DeepTyper artifact*. https://github.com/deeptyper/deeptyper. Accessed: 2018-07-22 (cit. on p. 94).

[274] *Escodegen ECMAScript code generator*. https://github.com/estools/escodegen. Accessed: 1-Nov-2016 (cit. on p. 34).

[275] *Flow: Static Type Checker for JavaScript*. https://flow.org/. accessed: 2018-09-12 (cit. on pp. 76, 96).

[276] *JSDoc tool*. https://github.com/jsdoc3/jsdoc. Accessed: 2018-07-22 (cit. on p. 89).

[277] *JSNice Artifact*. https://files.sri.inf.ethz.ch/jsniceartifact/index.html. Accessed: 2018-07-22 (cit. on p. 93).

[278] *Learning from Big Code datasets*. http://learnbigcode.github.io/datasets/. Accessed: 1-Nov-2016 (cit. on p. 35).

[279] *The ESTree specification*. https://github.com/estree/estree/blob/master/es2015.md. Accessed: 1-Nov-2016 (cit. on p. 40).

[280] *Typescript*. https://github.com/Microsoft/TypeScript. Accessed: 2018-07-22 (cit. on pp. 76, 96).

[281] *W3C Markup Validation Service*. https://validator.w3.org/. Accessed: 1-Nov-2016 (cit. on p. 36).

[282] *parse5: WHATWG HTML5 specification-compliant, fast and ready for production HTML parsing/serialization toolset for Node.js*. https://github.com/inikulin/parse5. Accessed: 1-Nov-2016 (cit. on p. 34).

ACADEMIC CV

March 2015 - May 2021
Doctoral Degree in Computer Science
University of Stuttgart, Germany

July 2011 - May 2013
Master Degree in Computer Science
National Institute of Technology Durgapur, India

June 2005 - June 2009
Bachelor Degree in Computer Science
West Bengal University of Technology, India