DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Learning a Static Analyzer from Code

Stefan Knilling

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Learning a Static Analyzer from Code

Maschinelles Lernen von statischen Analysen aus Code

Author:	Stefan Knilling
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisors:	Dr. Elmar Jürgens (TUM)
	Dr. Benjamin Hummel (CQSE GmbH)
	Dr. Stefan Krüger (CQSE GmbH)
Submission Date:	November 15, 2021

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, November 15, 2021 Stefan Knilling

Abstract

Static code analysis helps to detect defects and code smells early in the software lifecycle and thus leads to a decrease in maintenance cost. However, manually crafting rules for a static code analysis is a hard task. The analyzer should be sound and precise. It therefore needs to consider edge cases and program-specific environments. To support the creation of analysis rules, Bielik et al. [1] propose an approach to learn analysis rules from program code. The approach uses decision trees to learn analysis rules from input-output samples. To improve the generalization of the analysis rules, additional input-output samples are generated through code modification.

In this thesis, we replicate the approach of Bielik et al. to learn analysis rules for an allocation site analysis. The goal of an allocation site analysis is to detect source code locations that result in a heap allocation. An allocation site analysis is for example used underneath in static analyzers or directly assists developers to find memory leaks and bottlenecks. In addition to the implementation, our contribution is an in-depth case study to evaluate whether the learning approach is applicable to a production-ready static code analysis. The case study evaluates the learned analysis rules on a generated dataset of input-output samples and additionally on code from open source projects.

We show that accuracy, precision, and recall of the analysis rules are high for data from the generated dataset that is structurally similar to the training data. However, accuracy, precision, and recall are significantly lower for our evaluation of the learned analysis rules on open source code. We thus conclude that the rules are not yet sufficient for practical application.

Kurzfassung

Statische Code-Analyse hilft dabei, Fehler und Code-Smells früh im Software-Lebenszyklus zu erkennen und führt daher zu geringeren Wartungskosten. Analyseregeln für eine statische Code-Analyse manuell zu erstellen ist jedoch eine schwierige Aufgabe. Das Analyseprogramm soll gründlich und präzise sein. Daher muss es Randfälle und die programmspezifischen Umgebungen berücksichtigen. Um die Erstellung von Analyseregeln zu unterstützen, schlagen Bielik et al. [1] einen Ansatz zum Erlernen von Analyseregeln aus Programmcode vor. Der Ansatz verwendet Entscheidungsbäume, um Analyseregeln aus Input-Output-Beispielen zu lernen. Um die Generalisierbarkeit der Analyseregeln zu verbessern, werden durch Code-Modifikation zusätzliche Input-Output-Beispiele erzeugt.

In dieser Masterarbeit replizieren wir den Ansatz von Bielik et al., um Analyseregeln für eine Allocation-Site-Analyse zu erlernen. Das Ziel der Allocation-Site-Analyse ist es, Quellcodestellen zu erkennen, die zu einer Allokation von Heap-Speicher führen. Eine Allocation-Site-Analyse wird beispielsweise als Bestandteil statischer Analyseprogramme eingesetzt oder unterstützt Entwickler direkt beim Finden von Speicherlecks und -engpässen. Neben der Implementierung besteht unser Beitrag in einer ausführlichen Fallstudie, die untersucht, ob der Lernansatz auf eine für den produktiven Einsatz geeignete statische Code-Analyse anwendbar ist. Die Fallstudie evaluiert die Analyseregeln auf einem generierten Datensatz bestehend aus Input-Output-Beispielen und zusätzlich auf Code von quelloffenen Projekten.

Wir zeigen, dass Accuracy, Precision und Recall der Analyseregeln für Datenpunkte aus dem erzeugten Datensatz, die den Trainingsdaten strukturell ähneln, hoch sind. Accuracy, Precision und Recall sind jedoch deutlich geringer für unsere Auswertung der erlernten Analyseregeln auf quelloffenem Code. Wir schließen daraus, dass die erlernten Regeln noch nicht für eine praktische Anwendung geeignet sind.

Contents

1 Introduction

With an increasing size and complexity of modern software systems arises the risk of an increasing amount of defects and code smells. Humphrey [2] found that software systems contain more than 100 defects per 1,000 source lines of code and Jones [3] discovered about five bugs per function point. According to Shull et al. [4], finding and fixing defects after delivery is 100 times more expensive than in an earlier project phase. Analogously, code smells can hint towards issues with the quality of the code and eventually lead to the introduction of faults [5]. Therefore, it is important to find defects and code smells early in the development process to prevent an increase in maintenance effort.

One approach that helps to detect defects and code smells during development is static code analysis. A static code analysis, conducted by a static analyzer, examines programs without executing them. Static analyzers are often applied in between compilation and testing [6]. In recent development environments, many static analyzers are integrated to provide immediate feedback and to thereby support program development [7].

Defining a static analyzer and the underlying analysis rules is a hard task. The analyzer should scale to large software systems, be precise to avoid false positives, and be thorough to not miss too many issues within the program code. These requirements are further complicated by language-specific corner cases that should be considered and program-specific environments, for instance functions from the standard library or from third party dependencies, that have to be modeled by the analysis [1].

To help experts to craft scalable and robust analyzers, Bielik et al. [1] introduced an approach to apply machine learning to learn static analysis rules from code. Given a dataset of training samples and a domain-specific language to describe analysis rules, the goal of the approach is to learn analysis rules that can then be used to conduct static analyses. Each training sample maps source code locations to the expected analysis output. Based on a dataset of training samples, decision tree learning is applied to derive analysis rules in the given domain-specific language. To increase the robustness of the analysis, additional training samples are derived by modifying the original samples and feeding the created modifications back into the training procedure.

This work aims to replicate and to complement the approach introduced by Bielik et al. [1]. Our contribution is to reimplement the learning approach of Bielik et al. [1], apply it to the use case of allocation site analysis for JavaScript, and provide an in-depth evaluation. Allocation sites are locations within the source code where objects are created and heap memory is allocated [8]. The goal of an allocation site analysis is to find these source code locations. We then use the learned analysis rules to create a static analysis called *line-based analysis* that determines whether a line of code can be considered an allocation site. Using the learned analysis rules and the line based analysis, we conduct a case study to thoroughly evaluate

the learning approach. The evaluation of Bielik et al. [1] for the allocation site analysis relies on manual evaluation of analysis rules and is thus rather sparse. In our case study, we therefore extend the evaluation of Bielik et al. by measuring accuracy, precision, and recall of the learned analysis rules for a dataset of labeled allocation site samples. Furthermore, we analyze the correlation of the size of the training dataset and the accuracy of the analysis. Beyond the dataset, we analyze the generalization of the learned analysis rules for open source code using the implemented line-based analysis. Thereby, we want to determine whether the learned analysis rules are applicable to a production-ready static code analysis. Our results show that we can learn analysis rules that are accurate for structurally similar data, but do not generalize well to open source code. We conclude that the learned analysis rules are not applicable in practice.

The following chapter introduces theoretical foundations and concepts that appear throughout this thesis. After covering related work, we detail the approach to learn the analysis rules. Subsequently, we describe the instantiation and implementation of the approach for the allocation site analysis for JavaScript. Finally, the approach and the implementation are evaluated in a case study.

2 Theoretical Foundations

This chapter covers the fundamental concepts that are related to this thesis. The first section defines and describes static code analysis. Subsequently, machine learning is introduced with a focus on concepts that are applied in the approach described in Chapter 4. In the final section, we address the intersection of static code analysis and machine learning.

2.1 Static Code Analysis

In 1978, Johnson [9] introduced *Lint*, a tool to detect and report error-prone and wasteful code constructions in C programs. This early work reflects the origin of static code analyses as an idea to enforce additional, stricter rules on program code than those provided by the compiler. More generally, static code analysis is an approach to reason about computer programs without executing them [7]. In the following subsections, we first present areas of application and then discuss sensitivities of static code analyses. We further look at input representations of the program code for static analysis with a focus on abstract syntax trees that we use in our implementation in Chapter 5. Finally, with the *allocation site analysis*, we introduce a specific kind of static analysis that occurs throughout the subsequent chapters.

2.1.1 Areas of Application

Static code analysis has various areas of application. It can be applied to optimize compilers by providing relevant program properties. With information about unreachable code, for instance, compilers can reduce the code size. Furthermore, static code analysis is used to reason about program correctness by targeting generic correctness properties. For example, such properties might assure that variables are initialized before they are used and that there do not exist inputs that might lead to a division by zero. Finally, static code analysis is a key feature in modern IDEs to support program development, for example by annotating variables with potential types or by specifying the location of the definition of a called function [7].

2.1.2 Sensitivities

In this subsection, we introduce sensitivities of static code analyses. We cover context, flow, and path sensitivity.

Context sensitivity A context sensitive analysis utilizes additional information about the context of variables and abstract objects. This additional information is used by the analysis to distinguish between different executions. There are various forms of context

sensitivity based on the type of the context such as call-site sensitivity, object sensitivity, and type sensitivity [10]. A call-site sensitive analysis, for example, considers the supplied arguments of a function call as context and can thus distinguish multiple executions of a function for different sets of arguments [11].

Flow sensitivity Flow sensitivity implies that an analysis adheres to the control flow of a program. That is, a flow sensitive analysis considers the order of statements, while a flow insensitive analysis can handle statements in any order [10].

Path sensitivity A path sensitive analysis distinguishes between different execution paths of a program [12]. For an if-statement, for example, the analysis considers both execution paths that result from evaluating the condition of the if-statement.

2.1.3 Code Representations

The representations of code extend along the compilation pipeline from developer-written source code through several intermediate representations to the executable machine code. One of the intermediate representations that is of interest for this thesis is *Abstract Syntax Trees*. An Abstract Syntax Tree (AST) represents source code as a tree and abstracts details like punctuation or delimiters [13]. To generate an AST, the source code is parsed based on the context-free grammar (CFG) of the respective programming language. Each internal node of the AST relates to a non-terminal and leaves relate to terminals in the CFG [14]. That is, internal nodes correspond to language constructs such as function declarations or expression statements and leaves, for example, to concrete values of identifiers or literals. Edges describe a parent-child relation between the connected nodes. Figure 2.1 shows a code snippet in JavaScript and the corresponding AST according to the *ESTree*[1] specification. The internal nodes denote non-terminals like *VariableDeclaration*, *FunctionDeclaration*, or *Identifier* and the leaves contain terminal values like the string *greeting* or the function name *print*. Each node is connected to its parent with an undirected edge.

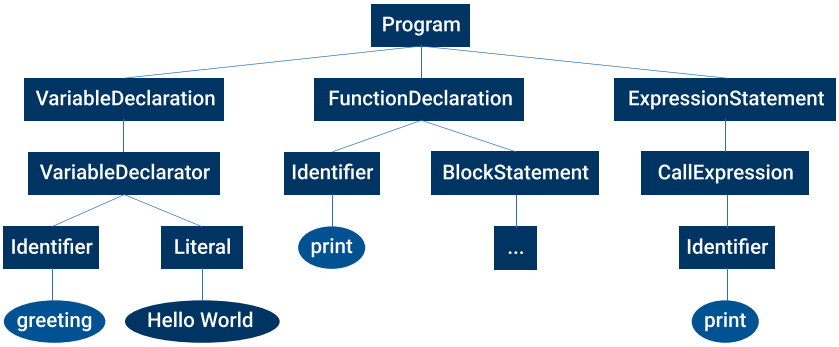
2.1.4 Allocation Site Analysis

An *allocation site analysis* is a static code analysis to detect source code locations that result in a heap allocation [1]. These source code locations are called allocation sites [8]. A source code location can for example be denoted by a line number, a start and end position, or a node in the AST. An allocation site analysis is used underneath in static analyzers [1]. For instance, allocation sites are used as abstract heap locations in a points-to analysis, which computes the set of abstract heap locations a variable points to [1, 10]. With the line-based analysis, we introduce a variant of an allocation site analysis that can be used directly to support developers in Section 5.6.

[1] https://github.com/estree/estree

```
1 let greeting = 'Hello World';
2 function print() {
3   console.log(greeting);
4 }
5 print();
```

(a) JavaScript code fragment



(b) Abstract Syntax Tree (AST) derived from the code fragment in (a)

Figure 2.1: Exemplary JavaScript code snippet and the corresponding AST according to the *ESTree* specification

2.2 Machine Learning

Besides static code analysis, machine learning in the form of decision trees plays an important role in this thesis. Therefore, this chapter first defines machine learning and the categorization into supervised and unsupervised learning, and finally introduces decision trees as an approach to machine learning.

2.2.1 Definition and Classification

Machine learning was already approached in 1959 by Samuel [15] to program computers to be able to learn from data instead of programming everything in detail. The term machine learning implies the philosophical question of how machines can learn. One approach is to see learning as a change in behavior of a program that was not explicitly programmed. For a program to change and to learn new behavior, Joshi [16] points out three factors: (1) Data that is consumed by the program, (2) a distance or error metric to determine the gap between current and expected behavior, and (3) a feedback mechanism that uses the error to improve the program's behavior.

Machine learning approaches are commonly classified into the two categories *supervised* and *unsupervised* learning.

Supervised learning Supervised learning models learn from labeled training data and apply that knowledge to new unlabeled data [17]. The training data consist of example input-output pairs that are used to train a learning function. The knowledge, the learning function thereby gains, is then used to label previously unlabeled data [18].

Unsupervised learning Unsupervised learning on the other hand utilizes unlabeled input samples. Therefore, unsupervised learning methods try to find patterns and relationships within data without utilizing a predefined context [17].

2.2.2 Decision Trees

A *decision tree* is a supervised approach to machine learning that classifies data based on its features [19]. A tree is built from a dataset of training samples where each sample consists of a set of feature values and a class label [20]. The root node and each internal node within the tree represent the label of a feature. Leaves contain class labels and edges depict concrete feature values [19].

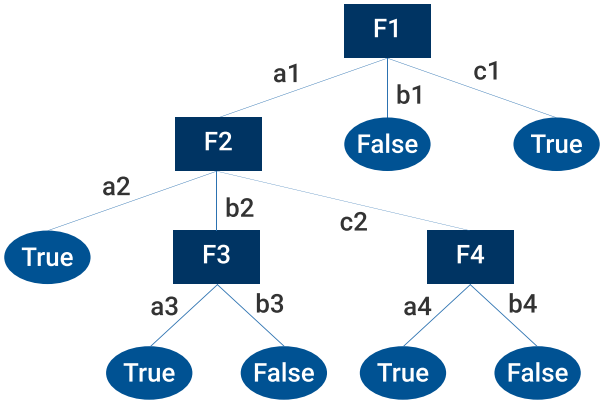


Figure 2.2: Example of a decision tree for binary classification (Adapted from [19]).

Figure 2.2 shows an example of a decision tree with four features $F1, F2, F3$, and $F4$. A hypothetical instance with feature values $\langle F1 : a1, F2 : c2, F3 : a3, F4 : b4 \rangle$ would be routed through the nodes $F1$, $F2$, and $F4$ to finally be classified with class label *False*.

To induce a decision tree from a dataset of training samples, different approaches that for example rely on an information gain or gini index to find the best split for the training data can be applied [19]. One algorithm to induce decision trees is the ID3 algorithm [21] that is applied in an adapted manner in the approach in Chapter 4.

2.2.3 Evaluation Metrics

Evaluation metrics are used to examine the performance of machine learning models on test data. In our evaluation, we use the metrics *accuracy*, *precision*, and *recall*. To calculate these metrics, machine learning models are executed on test data and the prediction of the model for each sample is recorded. Accuracy, precision, and recall [22, 23] are then defined as

$$Accuracy := \frac{True\ Positives + True\ Negatives}{True\ Positives + \ TrueNegatives + False\ Positives + False\ Negatives}$$

$$Precision := \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Recall := \frac{True\ Positives}{True\ Positives + False\ Negatives}.$$

Accuracy is a simple metric that provides a fast insight into the performance of a model by relating correct to incorrect predictions. However, accuracy does not properly account for imbalanced data with a dominant negative class [23]. We therefore additionally use precision and recall to get insights on the rate of true positives in comparison to false positives and false negatives, respectively. In summary, accuracy provides a quick overview and in combination with precision and recall a complete picture for our evaluation.

2.3 Application of Machine Learning to Static Code Analysis

Having introduced both static code analysis and machine learning, the final section of this chapter discusses approaches on how to utilize machine learning techniques for static code analysis. We distinguish between the usage of machine learning to create and to refine a static analysis.

Machine learning to create a static analysis Machine learning can be used as a building block to support the creation of static analyses. An exemplary application of machine learning is to derive analysis rules from source code. The approach by Bielik et al. [1] that is replicated in this thesis and described in detail in Chapter 4 extracts analysis rules that can then be used as a building block to create a static analysis. Another area where machine learning is used to create static code analyses is malware detection. For example, sources and sinks within source code are classified to prevent injection vulnerabilities [24] or to detect malicious code in network traffic [25].

Machine learning as an analysis refinement Furthermore, machine learning can help to refine static analyses. For example, machine learning techniques are applied to filter out false positives produced by an analysis. Typically, a classifier is trained to separate true positive from false positive analysis results [26, 27].

To apply machine learning to static code analysis, machine learning models are fed with different representations of code as input data. These can either be meta information like

source code comments, the source code itself, or an intermediate representation as described in Section 2.1.3. The code representation that we use in this thesis are Abstract Syntax Trees. ASTs are used as input for different machine learning techniques. Many applications of ASTs as input can be found in combination with neural networks [13, 28]. However, ASTs can also be used as input for decision trees [1] which we will cover in more detail in Chapters 4 and 5.

3 Related Work

In this chapter, we first discuss the relation between our work and the approach of Bielik et al. [1] that we aim to replicate. We then continue with related approaches that apply machine learning techniques to the field of static code analysis. Finally, we cover related work that uses the traditional approach of crafting static analyzers manually without applying machine learning techniques.

3.1 Learning Analysis Rules from Code

Bielik et al. [1] propose an approach to learn rules for a static code analysis from source code. They generate a dataset of input-output samples each consisting of program code and a corresponding analysis label by instrumenting the source code of the programs and subsequently executing these instrumented programs. They use this dataset of labeled programs to learn analysis rules in a given domain specific language. The domain specific language specifies the syntax of the analysis rules. Given the domain specific language and the dataset of input-output samples, decision trees are used to learn analysis rules with an adapted version of the ID3 algorithm. As the authors aim to create rules that generalize beyond the dataset of input-output samples used for training, they additionally propose a counter-example guided learning procedure. This procedure efficiently generates new input-output samples by modifying the source code of programs from the initial dataset. First, the procedure performs program modifications such as inserting dead code or renaming variables to create new potential samples. Subsequently, candidate analysis rules that have been learned from the initial dataset of input-output samples are evaluated against these newly created samples. If the candidate analysis rules do not produce the expected analysis result for a modified sample, it is added to the initial dataset and consequently used to learn more general analysis rules.

Bielik et al. [1] implement their approach for an allocation site analysis and a points-to analysis for JavaScript code. In their evaluation, they report a high precision of 99.9% of the learned analysis rules for the points-to analysis. For this evaluation, they used a dataset derived from the programs of the original dataset by applying various code modifications. Bielik et al. do not evaluate the analysis rules on code that is unrelated to the training dataset. They furthermore do not examine the precision or comparable metrics of the allocation site analysis.

Our goal is to replicate the promising results of Bielik et al. [1] to learn analysis rules for an allocation site analysis from JavaScript source code. We provide our own implementation of the approach of Bielik et al. to first learn and to then refine the analysis rules. Our

implementation uses decision trees and the adapted version of the ID3 algorithm proposed by Bielik et al. to learn analysis rules. We additionally implement the counter-example guided learning procedure to improve the generalization of the learned analysis rules. Following Bielik et al., we implement a tool to automatically generate training data from instrumented programs. In contrast to Bielik et al., we focus on the implementation of the allocation site analysis and do not implement the points-to analysis. A key difference to the work of Bielik et al. is our in-depth evaluation of the allocation site analysis that systematically examines the performance of the learned analysis rules measuring accuracy, precision, and recall. We evaluate the learned analysis rules for the allocation site analysis both on subsets of the generated dataset and on code from open source projects. For the evaluation on open source code, we implement a static code analysis on top of the learned analysis rules that examines code files and determines whether a line of code within a file is an allocation site. In summary, our evaluation has a broader scope and aims to investigate the practical applicability of the learned analysis rules.

To the best of our knowledge, there are no other comparable approaches in literature that aim to learn the underlying analysis rules of a static analysis from code.

3.2 Static Analysis and Machine Learning

The application of machine learning techniques to the field of static code analysis is a common topic in recent literature and many different approaches are proposed. In the following, we give a few related examples where machine learning techniques are applied to static code analysis.

Alikhashashneh et al. [27] and Koc et al. [29, 26] provide similar approaches that use machine learning to distinguish between relevant true positive and false positive analysis reports. They both point out that current static code analysis tools report many false positives. Their approaches therefore apply different machine learning methods such as Support Vector Machines or Random Forests to train a classifier that is able to distinguish between relevant and non-relevant analysis reports. Similar to our work, they use machine learning techniques to improve static code analyses. While the approaches of Alikhashashneh et al. and Koc et al. aim to improve the results of already given static code analyses in a downstream process step, we use machine learning to create analysis rules with the goal of preventing false positive reports upfront.

Chibotaru et al. [30] present an approach to detect sources, sinks, and sanitizers in source code to then automatically find paths from sources to sinks without a sanitizer. These paths pose a security risk because they are prone to injection vulnerabilities. The authors use linear optimization to learn a specification that describes sources, sanitizers, and sinks from a dataset of programs in a semi-supervised fashion. They manually annotate a small subset of the dataset of programs and use these labeled samples alongside the unlabeled data to infer the specification. This specification is then used to find vulnerable, unsanitized program paths. In their experimental evaluation on Python code, they show that the learned specification reaches a precision of 67% on average. Similar to our approach, Chibotaru et al. learn rules

that describe program properties from a dataset of programs. Our learning approach is fully supervised and operates on labeled training samples only. While the semi-supervised learning procedure of Chibotaru et al. requires a small amount of manually labeled training data, we automatically generate a big dataset of labeled data from program execution to avoid manual labeling. With our supervised learning approach, we aim to reach a higher precision by using labeled training data only.

The work of Raychev et al. [31] utilizes Structured Support Vector Machines to learn a probabilistic model from code which is subsequently used to predict properties of programs. They apply this model to predict type annotations and variable names of JavaScript programs. Both type annotations and meaningful variable names help developers to better understand source code. Analogous to Raychev et al., we use program code to learn and predict properties of programs. However, we do not learn a probabilistic model, but use decision trees to learn analysis rules from program code. These rules are then used as part of a static analysis to reason about a program.

Paletov et al. [32] infer security rules for cryptographic APIs from code changes. They mine code changes from open source repositories and derive usage changes from the mined code changes. Non-relevant usage changes are filtered out and the remaining usage changes are clustered into semantically related groups using hierarchical clustering. Finally, the authors manually derive security rules from these clusters. Similar to our work, Paletov et al. use Abstract Syntax Trees (AST) as input code representation and aim to derive a set of rules. In difference to our approach, they do not directly learn from code but from changes to the code. In addition, our approach does not require a final manual step to derive the analysis rules but instead learns the rules fully automatically.

To detect malicious JavaScript code, Wang et al. [33] use a deep learning model and logistic regression. They design and train a model for a Stacked Denoising Autoencoder with JavaScript code snippets encoded in a binary format. Each code snippet is labeled either as malicious or benign. A logistic regression uses these features to detect malicious code. In line with our work, the model of Wang et al. learns features of JavaScript code so that these features do not have to be crafted manually. We also extract features from JavaScript code based on a training dataset, but our features are represented in the form of analysis rules and aim to detect allocation sites in JavaScript code rather than malicious code snippets. Instead of a deep learning model, we use decision trees to learn the analysis rules. We further use ASTs as input representation unlike Wang et al. who use a binary representation of JavaScript code.

3.3 Static Analysis Without Machine Learning

This section covers related approaches that manually define and create static code analyses. We focus on approaches that operate on JavaScript code as we also learn analysis rules for JavaScript in this thesis.

Kashyap et al. [34] implement a static analyzer that can for example be applied to type inference and pointer analysis. The design of their analyzer consists of an intermediate code

representation for JavaScript called *notJS*, abstract semantics for notJS, and abstract domains. All these analysis components are designed manually. For example, the intermediate code representation notJS contains specific design decisions such as a differentiation between pure expressions that are guaranteed to not throw an exception and impure statements without this guarantee. In line with this thesis, Kashyap et al. craft the underlying rules of static code analyses. However, they rely on the traditional way of designing the components of the static analysis by hand. The approach we use instead tries to reduce the effort and complexity of creating a static analyzer by providing analysis rules learned from code. These rules can be integrated into existing static analyzers to improve their performance or can be used to craft new analyzers.

The work of Lee et al. [35] introduces a static analysis framework for JavaScript called *SAFE*. To support various different analyses, the authors specify several intermediate code representations of JavaScript and formally defined analysis components. Similar to our work, Lee et al. aim to provide a foundation that can be used to create static analyzers. Their detailed formal specifications of the intermediate representations require a deep knowledge of the JavaScript standard. The approach we use instead creates components of a static analysis in the form of analysis rules without requiring a deep understanding of each edge case of the language by directly learning the analysis rules from code.

4 Approach

A major part of this thesis is the reimplementation of the approach to learn analysis rules introduced by Bielik et al. [1]. Before we describe the implementation for the use case of an allocation site analysis in Chapter 5, we first introduce the approach in an abstract manner that is applicable to different types of analyses. Thereby, this chapter closely follows the initial work of Bielik et al. [1].

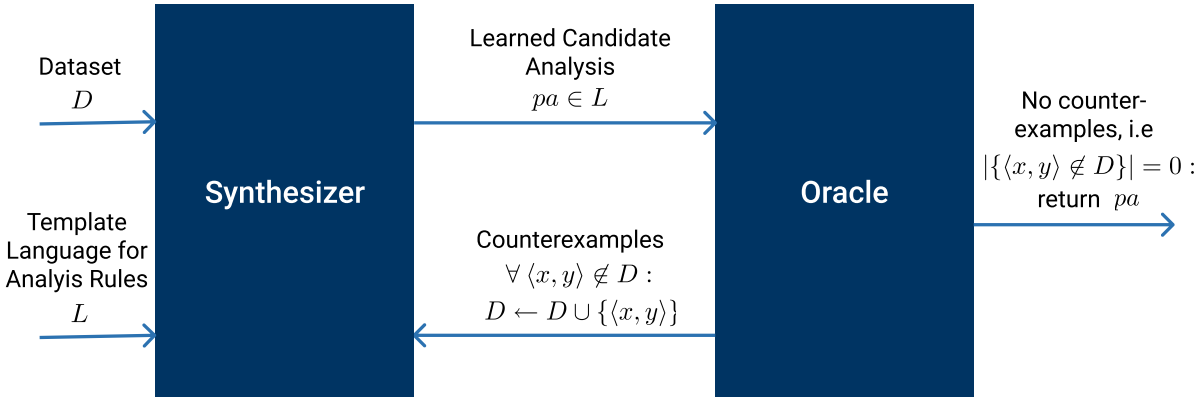


Figure 4.1: Overview of the approach to learn static analysis rules. During Synthesis, a candidate analysis *pa* is learned from a dataset D. The language L is used to describe the rules. The oracle generates new data that is fed back to the synthesis to improve analysis generalization (Adapted from [1]).

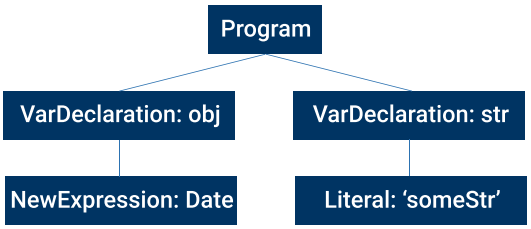
The approach consists of two major components: the *synthesizer* and the *oracle*. Figure 4.1 provides an overview of both components and their interactions. The synthesizer takes a dataset D, consisting of pairs of programs and their respective desired analysis output, and a domain-specific language L as input. It returns a candidate analysis $pa \in L$. To improve the generalization of the model beyond the training data, the oracle takes a candidate analysis *pa* as input and derives new, unseen training samples. For this, the code of the initial samples in D is modified to find samples that are not yet correctly classified by *pa*. These incorrectly classified samples are then fed back to the synthesizer. Both components, the synthesizer and the oracle, are interconnected in a counter example guided loop. The following subsections describe the synthesis of analyses and the oracle to create new training samples in detail.

4.1 Program Synthesis

The goal of the program synthesis is to learn an analysis $pa \in L$ that is both sound and precise. The analysis is learned from a dataset $D = \{\langle x^j, y^j \rangle\}_{j=1}^{N}$ of tuples consisting of programs x^j from a programming language T_L and expected analysis outputs y^j from an abstract domain of analysis results. Figure 4.2 shows an example of learning analysis rules for an allocation site analysis from code. Figures 4.2a and 4.2b show the code to learn from and its AST representation, respectively. The dataset D consisting of two training samples is shown in Figure 4.2c. Each sample maps a node within the AST to a boolean that determines whether or not the given node is considered an allocation site. The variable declaration $VarDeclaration : obj$ is an allocation site as a new object is instantiated and heap memory is allocated. In contrast, the node $VarDeclaration : str$ is not considered an allocation site because strings are primitive data types in JavaScript and no new object is created. Thus, in our example, a program x^j is concretely represented by the source code given in Figure 4.2a and a position within the code in the form of an AST node. The decision tree in Figure 4.2d represents a candidate analysis that is learned from D. The analysis consists of one guard that checks whether the child of a node in the AST is a *NewExpression*. If the guard evaluates to true, a node is considered an allocation site. Otherwise, it is no allocation site. In the following, we will refer to this example in order to illustrate the theoretical explanations with a concrete application.

```
1  const obj = new Date();
2  const str = 'someStr';
```

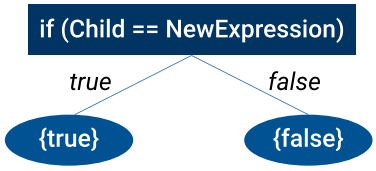
(a) An exemplary JavaScript code snippet.



(b) The simplified AST for the code in (a).

$$
\begin{aligned}
D = \{ & \\
& \langle VarDeclaration : obj, \{true\} \rangle, \\
& \langle VarDeclaration : str, \{false\} \rangle \\
\} &
\end{aligned}
$$

(c) The training Data D.



(d) A candidate analysis pa learned from D.

Figure 4.2: An Example of learning rules for an allocation site analysis from code.

As a first step, we recap the soundness and precision of an analysis as defined by Bielik et al. [1]. Next, we describe the template language L that is used to construct analysis rules and then introduce the procedure to learn these rules.

4.1.1 Analysis Soundness and Precision

A candidate analysis pa is a function $pa : T_L \rightarrow \mathcal{P}(A)$ that maps a program $x^j \in T_L$ to the abstract domain of analysis results $\mathcal{P}(A)$. While A represents the set of potential analysis results, its power set $\mathcal{P}(A)$ contains all subsets of A and thus denotes the abstract domain of analysis results. An analysis is considered *sound* and *precise* if it correctly maps each program x^j in D to the expected analysis result y^j. This can be formalized as

$$\forall j \in \{1, \ldots, N\} \; : \; y^j = pa(x^j). \tag{4.1}$$

However, a perfectly sound and precise analysis for large datasets as defined in (4.1) is unrealistic in practice due to noisy data. A simple example of noisy data can be given by two samples $\langle x, y \rangle \in D$ and $\langle x, y' \rangle \in D$ with $y \neq y'$. While both programs x are equal, a different analysis result is expected in each case. These contradicting requirements cannot be fulfilled by any learned analysis since pa can only map the program x to either y or y'.

Analysis Soundness To be able to handle noisy data, Bielik et al. [1] slightly relax the definition above. An analysis pa is sound, if the condition

$$\forall j \in \{1, \ldots, N\} \; : \; y^j \subseteq pa(x^j) \tag{4.2}$$

is fulfilled. An equality of the produced analysis result $pa(x^j)$ and the expected result y^j is no longer required. Instead, the analysis may approximate the expected analysis result by taking a superset of all y^j.

As an example, consider $A = \{true, false\}$ as potential analysis results and a dataset D consisting of two noisy samples $s_1 = \langle x, \{true\} \rangle \in D$ and $s_2 = \langle x, \{false\} \rangle \in D$. A candidate analysis pa can now map a noisy sample to the superset $\{true, false\}$ as approximation to produce a sound analysis. Given the approximation $pa(s_1) = pa(s_2) = \{true, false\}$, we can derive that

$$\begin{aligned} &\{true\} \subseteq pa(s_1) \wedge \{false\} \subseteq pa(s_2) \\ \implies &\forall j \in \{1, 2\} \; : \; y^j \subseteq pa(x^j) \end{aligned}$$

and thus pa is sound as defined in (4.2).

Hence, according to the definition of soundness in (4.2), we can learn a sound analysis for data that includes noise. However, the definition of soundness no longer guarantees a precise analysis. Consider a trivially sound analysis that maps each program x^j to the union of all expected analysis results $\cup_{j=1}^{N} y^j \subseteq A$ of a dataset D as analysis result. While such an analysis is sound according to (4.2), it is also imprecise.

Analysis Precision To compute the precision of an analysis $pa \in L$ on a dataset D, a cost function is defined. First, the cost r of a sample $\langle x \in T_L, y \subseteq A \rangle \in D$ given a candidate analysis $pa \in L$ is calculated as

$$r : T_L \; x \; A \; x \; L \rightarrow \mathbb{R}, \; r(x, y, pa) = \begin{cases} 1, \; y \neq pa(x) \\ 0, \; else \end{cases} \quad . \tag{4.3}$$

If the result produced by pa differs from the expected result y, a cost of 1 is incurred. Otherwise, there is no cost.

The cost of an analysis pa for the complete dataset D is then defined as the sum of the costs for all samples

$$cost(D, pa) := \sum_{\langle x,y \rangle \in D} r(x, y, pa). \tag{4.4}$$

If $cost(D, pa) = 0$, (4.4) entails that pa is sound and precise according to the definition in (4.1) as a cost of 0 implies that

$$\forall j \in \{1, \ldots, N\} \; : \; r(x^j, y^j, pa) = 0 \tag{4.5}$$

has to hold. It follows from (4.3) and (4.5) that

$$\forall j \in \{1, \ldots, N\} \; : \; y^j = pa(x^j)$$

which exactly corresponds to the definition of a correct analysis.

Example Looking at our example in Figure 4.2 on page 14, we can conclude that the analysis pa given in Figure 4.2d is both sound and precise for the training data D (Figure 4.2c). The analysis returns the correct labels $\{true\}$ and $\{false\}$ for the first and second sample, respectively. Hence, pa is sound as $\forall j \in \{1, 2\} \; : \; y^j \subseteq pa(x^j)$ holds. Furthermore, as $\forall j \in \{1, 2\} \; : \; y = pa(x)$, we also know that $cost(D, pa) = 0$ and therefore, pa is precise.

4.1.2 Language Template for Analysis Rules

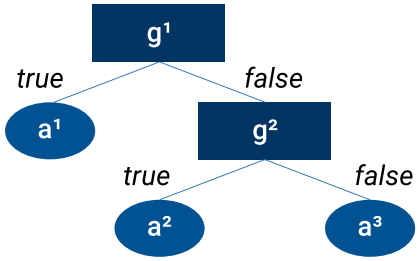
Before we describe how analysis rules are learned, we first introduce the template for the language L, which is used to describe analysis rules. Figure 4.3 gives an overview of the language template. The language L consists of *actions*, *guards*, and *if-then-else* statements that combine actions and guards in a recursive manner. A learned analysis $pa \in L$ can thus be represented as a tree with guards for if-then-else statements as internal nodes and actions as leaves. For if-then-else statements, a guard is used as a condition and depending on the evaluation of the guard, the respective *if* or *else* branch is taken. The concrete specification and semantics of actions and guards is context specific for each analyzer that should be learned. We show a detailed example for the allocation site analysis in Section 5.2.

4.1.3 Learning Analysis Rules

Having defined the language L, the next step is to learn an analysis $pa \in L$ from a dataset D. The learned analysis should be sound while minimizing over-approximation. The authors [1] thus define that a learned analysis pa should be sound on the samples provided in D while minimizing the cost $cost(D, pa)$ for the sake of precision.
As previously noted, an instance of L can be represented as a tree with guards as internal nodes and actions as leaves. Thus, to learn an analyzer, decision tree learning is applied.

$a \in Actions$
$g \in Guards$
$l \in L ::= a \mid \textbf{if } g \textbf{ then } l \textbf{ else } l$

(a) Grammar of the template language L.



(b) Example of an analysis $pa \in L$ with guards g and actions a displayed as a decision tree.

Figure 4.3: Specification of the grammar of the template language L and an exemplary analysis composed from L (adapted from [1]).

Algorithm 1 Synthesis algorithm to learn an analyzer $pa \in L$ from a dataset D.

procedure SYNTHESIZE(D)
  $a_{best} \leftarrow genAction(D)$
  if $cost(D, a_{best}) = 0$ **then return** a_{best}
  end if
  $g_{best} \leftarrow genBranch(a_{best}, D)$
  if $g_{best} = \bot$ **then return** $approximate(D)$
  end if
  $p_1 \leftarrow Synthesize(\{\langle x, y\rangle \in D \mid g_{best}(x)\})$
  $p_2 \leftarrow Synthesize(\{\langle x, y\rangle \in D \mid \neg g_{best}(x)\})$
  return $if\ g_{best}\ then\ p_1\ else\ p_2$
end procedure

Bielik et al. [1] propose an adapted version of the ID3 [21] algorithm. Following ID3, the algorithm recursively creates a decision tree top-down in a greedy fashion. The algorithm guarantees soundness of the resulting analysis and locally maximizes an information gain metric to create branches. Algorithm 1 outlines the procedure as pseudo code. To generate actions and branches, the algorithm specifies three helper functions *genAction*, *genBranch* and *approximate* that are detailed in the following.

genAction The function *genAction* selects the best analysis a_{best} that only consists of actions. The criterion for the best analysis is the minimization of cost given by

$$a_{best} = genAction(D) := \underset{a \in Actions}{\text{argmin}}\ cost(D, a).$$

Simply put, *genAction* returns the action a_{best} with the lowest cost for a given dataset D. In an optimal scenario, $cost(D, a_{best}) = 0$ which indicates a sound and precise analysis that can directly be returned. Otherwise, the algorithm continues with generating a new branch.

Example In our running example of learning an allocation site analysis in Figure 4.2 on page 14, we have to consider two actions *NewAlloc* and *NoAlloc* that mark a node in the AST as an allocation site and as no allocation site, respectively. To select the best analysis a_{best} for the complete dataset D, we first calculate the cost for an analysis that only consists of one of the two actions *NewAlloc* and *NoAlloc*. For an analysis consisting of *NewAlloc* only, this calculation yields

$$\begin{aligned} &cost(D,\ NewAlloc) \\ &= r(VarDeclaration : obj,\ \{true\},\ NewAlloc) + r(VarDeclaration : str,\ \{false\},\ NewAlloc) \\ &= 0 + 1 = 1. \end{aligned}$$

Analogously, $cost(D,\ NoAlloc) = 1$. As the cost for both analyses is 1, either of the two analyses can be selected as a_{best}. However, we cannot find an action with a cost of 0 and the algorithm thus creates a new branch.

genBranch Branches are created by splitting the dataset into two parts based on a condition $g \in Guards$. To determine which condition is selected, an information gain that quantifies the gain in analysis correctness over selecting the imprecise analysis a_{best} is defined.

The information gain metric is based on a standard entropy definition. For a vector $w = \langle w_1, ..., w_k \rangle$ with $w_i \in C$, where C can be any set, the entropy H is defined as

$$H(w) := -\sum_{c \in C} \frac{count(c, w)}{k} log_2 \left(\frac{count(c, w)}{k} \right).$$

The function $count(c, w) := |\{i \in \{1, \ldots, k\} \mid w_i = c\}|$ calculates the occurrences of $c \in C$ in a vector w. In the context of branch generation, a specific vector $w_d^{a_{best}}$ is defined

for a dataset $d = \{\langle x_i, y_i \rangle\}_{i=1}^{|d|} \subseteq D$ as

$$w_d^{a_{best}} := \langle r(x_i, y_i, a_{best}) \mid i \in \{1, \ldots, |d|\} \rangle.$$

Thus, $w_d^{a_{best}}$ contains the cost of each sample in a dataset d with respect to a_{best}. A guard $g \in Guards$ is a function that maps a program x to a co-domain that consists of two distinct values. Therefore, it is used to split the dataset D into two distinct subsets D^g and $D^{\neg g}$. They are defined as $D^g := \{\langle x, y \rangle \in D \mid g(x)\}$ and $D^{\neg g} := D \setminus D^g$. The information gain is then constructed as follows:

$$IG^{a_{best}}(D, g) := H(w_D^{a_{best}}) - \frac{|D^g|}{|D|} H(w_{D^g}^{a_{best}}) - \frac{|D^{\neg g}|}{|D|} H(w_{D^{\neg g}}^{a_{best}}).$$

Using this definition of the information gain, the guard g_{best} with the highest information gain that yields the best split of D can now be determined as

$$\begin{aligned} g_{best} &= genBranch(a_{best}, D) \\ &:= \begin{cases} \text{argmax}_{g \in Guards}\, IG^{a_{best}}(D, g), & \exists g \in Guards \,:\, IG^{a_{best}}(D, g) > 0 \\ \bot, & else \end{cases} . \end{aligned}$$

If there exists at least one guard with positive information gain, the guard with highest information gain is selected. Otherwise, the information gain is 0 for all guards and $\bot$ is returned which denotes that the respective branch of the analysis has to be approximated.

Example Returning to the example shown in Figure 4.2 on page 14, we assume that $a_{best} = NewAlloc$. We analyze one guard g with the condition $Child == NewExpression$ as depicted in Figure 4.2d. This guard splits the dataset D into the subsets $D^g = \{\langle VarDeclaration : obj, \{true\} \rangle\}$ and $D^{\neg g} = \{\langle VarDeclaration : str, \{false\} \rangle\}$. For the vectors w, we thus get $w_D^{NewAlloc} = \langle 0, 1 \rangle$, $w_{D^g}^{NewAlloc} = \langle 0 \rangle$ and $w_{D^{\neg g}}^{NewAlloc} = \langle 1 \rangle$. We can now calculate the information gain

$$\begin{aligned} IG^{NewAlloc}(D, g) &= H(w_D^{NewAlloc}) - \frac{|D^g|}{|D|} H(w_{D^g}^{NewAlloc}) - \frac{|D^{\neg g}|}{|D|} H(w_{D^{\neg g}}^{NewAlloc}) \\ &= 1 - \frac{1}{2} \cdot 0 - \frac{1}{2} \cdot 0 = 1. \end{aligned}$$

As our guard g has a positive information gain, it is selected as best guard $g_{best} = g$. The *sythenize* procedure then continues with the next recursion steps given datasets D^g and $D^{\neg g}$ as input.

approximate An information gain of 0 is a consequence of noisy data with contradicting labels as previously illustrated in Section 4.1.1. In this case, the analysis result for the current branch has to be over-approximated to be equal to the abstract domain of possible

analysis results A. Theoretically, this asserts a sound analysis according to definition (4.2). In practice, only few branches should require approximation. The analysis result for samples that run into an approximated branch is ambiguous. Therefore, approximated branches reduce the precision of the analysis rules.

The *synthesize* procedure is called recursively to derive the final analyzer $pa \in L$ for a dataset D.

4.2 Oracle

The second major component besides the synthesizer is the oracle. While analyses produced by the synthesizer are sound and precise for the training data D, they might not generalize well to data beyond D. This overfitting leads to correct analysis results for samples that are included in the initial dataset D, but to incorrect analysis results for samples that are not part of D.

The candidate analysis *pa* shown in Figure 4.2d on page 14 is an example of an analysis that does not generalize well. We previously saw that it is both sound and precise for the corresponding dataset D. The analysis *pa* considers all AST nodes with a *NewExpression* as child to be an allocation site. However, there are exceptions to this rule: if the object constructor in JavaScript is called with an object as argument, it only returns the argument's object reference and therefore does not allocate new heap memory. Figure 4.4 shows an example of this edge case. In line one, a new empty object *obj* is allocated and passed to the object constructor in the second line. The object constructor then returns the reference to *obj* and does not allocate new heap memory. However, *pa* would still consider the declaration of *noAlloc* in the second line as an allocation site. Thus, *pa* does not generalize to the given edge case and is partly imprecise.

```
1 let obj = {};
2 const noAlloc = new Object(obj);
```

Figure 4.4: Invoking the object constructor with an object as argument does not allocate new heap memory, but instead returns the argument's object reference.

To address the problem of overfitting, Bielik et al. [1] propose the oracle to automatically derive new programs that are incorrectly evaluated by an analysis *pa* from the available samples in D. That is, for the set of programs $P_D = \{x \mid \langle x, y \rangle \in D\}$, the oracle finds counter examples $p \in T_L$ where $p \notin P_D$ and the analysis soundness as defined in equation 4.2 is violated for program p and the given analysis *pa*. The derived programs are added to the training data in D and used in addition to train a new analysis. To find suitable programs, the oracle performs two steps: determining positions for modifications within programs from P_D and executing selected modifications at these positions.

4.2.1 Determine Modification Positions

The oracle should be able to quickly find counter examples. However, the space of potential programs in T_L is large and thus naively modifying programs at random positions is not feasible. Hence, Bielik et al. [1] suggest to select program positions that are traversed during the execution of an analysis pa. As we have previously seen, an analysis $pa \in L$ can be represented as a decision tree containing different paths. Modifying program positions in program p, which are relevant for path selection in pa, are likely to affect the selected analysis path. A different path in pa might finally lead to a different analysis results. The traversed program positions when running an analysis are recorded by instrumenting the analyzer.

4.2.2 Selection of Modifications

Besides the approach to find relevant program positions, the authors [1] define techniques to modify these positions to derive potential counter examples. The techniques can either preserve or potentially alter the program's semantics.

Semantic-preserving modifications The first technique modifies a program in a way that preserves the original semantics of the program. Furthermore, the transformations should not influence the results of the analysis pa. Given a program $p \in P_D$, a set of programs with the same semantics is created. Formally, the transformation is a function $F_{sp} : T_L \times X \to \mathcal{P}(T_L)$. The function F_{sp} takes a program and a position within the program, for example a program counter or an AST node, as input and returns a set of modified programs. A correct analysis implies that

$$\forall p' \in F_{sp}(p, n) \; : \; pa(p) = pa(p') \tag{4.6}$$

holds. That is, the analysis pa produces the same result for the original program and all of the modified program variants. If (4.6) is violated for a program $p' \in F_{sp}(p, n)$, the oracle has found a suitable counter example and the analysis pa is thus incorrect. The counter example p' is returned to the synthesis. Examples for semantic-preserving modifications are inserting dead code and renaming user-defined identifiers.

Modifications that may not be semantic preserving Modifications that may alter the semantics are specified by the transformation function $F_{sa} : T_L \times X \to \mathcal{P}(T_L)$. The transformed programs come with different behaviors that may not be part of any program $p \in P_D$. These programs' behaviors are thus not incorporated in an analyzer pa that was trained on a dataset D. Modifications that may impact the semantics are, for instance, changing values of constants or adding arguments and parameters to functions.

Bielik et al. [1] show that finding a counter example for a given analysis pa by carefully selecting the modification positions is far more efficient than simply applying modifications to randomly selected program positions. These counter examples help to improve the generalization of the analysis rules beyond the training data. For the example in Figure 4.2 on page 14, the synthesis could come up with an alternative analysis as shown in Figure 4.5. This

analysis is based on the assumption that a variable declaration is an allocation site if its right sibling in the AST also is a variable declaration. For the first training sample of D, the node $VarDeclaration : obj$ has a variable declaration as right sibling and is considered an allocation site by the analysis. The second node of the training data $VarDeclaration : str$ does not have a variable declaration as its right sibling and is therefore correctly labeled as no allocation site. Hence, the analysis is sound and precise for the training data D. However, the relation between an allocation site and the node type of the right sibling is a coincidence specific to the example and the analysis does thus not generalize to arbitrary code. To eliminate this coincidental relation, the oracle could create a counter example that inserts dead code in between the two code lines of Figure 4.2a on page 14. Given this additional counter example, the synthesis might produce an analysis that better generalizes to data beyond the training samples.

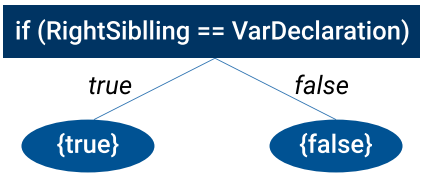


Figure 4.5: An alternative analysis for the example in Figure 4.2 that overfits to the training data D.

In the next chapter, we cover the implementation details of the synthesis and the oracle.

5 Instantiation and Implementation

In this chapter, we describe the concrete instantiation and the implementation details to learn analysis rules for an allocation site analysis following the approach in Chapter 4. Our learned analysis aims to find allocation sites in JavaScript code. The implementation is also written in JavaScript to make use of the rich ecosystem. Our implementation tries to closely follow the approach taken by Bielik et al. [1]. However, as we do not have access to the source code or similar implementation details, this chapter is based on our specific implementation and might therefore partly diverge from the original implementation by Bielik et al. [1].

In the first section, we examine how the abstract variables introduced in Chapter 4 are instantiated with concrete values for learning rules for the allocation site analysis. Next, we describe the syntax, semantics, and implementation details of the domain-specific language L_{alloc} used to describe analysis rules for the allocation site analysis. We then continue with the implemented approach to collect training data. Finally, we present our JavaScript implementation of the program synthesis and the oracle.

5.1 Variable Instantiation

This section covers how the abstract properties as described in Chapter 4 are mapped to concrete values for the allocation site analysis. We saw that an analysis is a function $pa : T_L \to \mathcal{P}(A)$. The set of potential analysis results $A = \{true, false\}$ for an allocation site analysis holds the two elements *true* representing an allocation site and $false$ denoting no allocation site. The abstract domain $\{\{true\}, \{false\}, \{true, false\}\} \subset \mathcal{P}(A)$ includes all non-empty subsets of A. We do not consider the empty set as it is not a potential result of the allocation site analysis. Further, we define $T_L := AST \times X$ as a set of tuples containing JavaScript programs $t \in AST$ in their abstract syntax tree (AST) representation and positions $n \in X$ within t. In our implementation, X denotes all nodes within the respective AST and a position n is one of these nodes. In summary, an analysis pa maps a node within a given AST to one of the sets $\{true\}$, $\{false\}$, or, in case of approximation, $\{true, false\}$.

For the branch generation, Bielik et al. [1] further instantiate the set $C := \{true, false\}$ that is used to calculate the information gain. The oracle modification functions $F_{sp} : T_L \times X \to \mathcal{P}(T_L)$ and $F_{sa} : T_L \times X \to \mathcal{P}(T_L)$ take an additional modification position from X as input where X is again defined as the set of all program nodes of the AST t from T_L.

The instantiation of the domain-specific language L_{alloc} is more complex and described in the following section.

5.2 Domain-Specific Language

As previously introduced, the template language L consists of *actions*, *guards*, and *if-then-else* statements. A summary of the template is shown in Figure 4.3 on page 17. For the task of learning an allocation site analysis, Bielik et al. [1] instantiate L to provide a domain-specific language L_{alloc}. The underlying idea of the domain-specific language is to define means to navigate and condition an AST. In the following, we first discuss the specification of the syntax and the semantics of L_{alloc}. Afterwards, we cover implementation details.

5.2.1 Specification of Syntax and Semantics

In this subsection, we detail the syntax and semantics of the domain specific language L_{alloc}.

Syntax Figure 5.1 gives an overview of the syntax of L_{alloc}. The authors [1] define two basic instruction types: *moves* and *writes*. Moves define operations to navigate the AST and writes collect information about properties of traversed AST nodes. For the allocation site analysis, an action can either denote an allocation site (NewAlloc), no allocation site (NoAlloc), or an approximation of the result (Approx). Guards of L_{alloc} are composed of move operations and one write operation. Furthermore, a context ctx determines the right side of the condition of an if-then-else statement.

$$
\begin{aligned}
m \in \mathit{Move} &:= \mathrm{Up} \mid \mathrm{Left} \mid \mathrm{Right} \mid \mathrm{DownFirst} \mid \mathrm{DownLast} \mid \mathrm{PrevNodeValue} \mid \mathrm{PrevNodeType} \\
w \in \mathit{Write} &:= \mathrm{WriteValue} \mid \mathrm{WritePos} \mid \mathrm{WriteType} \mid \mathrm{HasPrevNodeValue} \\
a \in \mathit{Actions} &:= \mathrm{NewAlloc} \mid \mathrm{NoAlloc} \mid \mathrm{Approx} \\
g \in \mathit{Guards} &:= \mathit{Move}^* \; ; \mathit{Write} \\
\mathit{ctx} \in \mathit{Context} &:= (N \cup \Sigma \cup \mathbb{N} \cup \{\mathit{true}, \mathit{false}\}) \\
l \in L_{alloc} &:= \epsilon \mid a \mid \textbf{if } g = \mathit{ctx} \textbf{ then } l \textbf{ else } l
\end{aligned}
$$

Figure 5.1: Syntax of the domain-Specific language L_{alloc} to describe rules for an allocation site analysis (adapted from [1]).

Semantics Computing an analysis pa utilizes a program state $\sigma := \langle s, t, n, ctx \rangle \in L_{alloc} \times AST \times X \times Context$. A program s consists of program instructions from L_{alloc}. Initially, before any operations from s are computed, s describes the complete analysis pa. The variable t is an AST, n the current position in the tree, and ctx the current program context. The context ctx consists of either a non-terminal symbol N from the tree, for example a node type, a terminal symbol, for instance a value of a literal, a natural number $\mathbb{N}$, or a boolean value. A computation executes an operation op from s, for example a move or a write, and transforms the state σ to a new state σ' denoted as $\sigma \rightarrow \sigma'$. We describe the semantics of the language elements of L_{alloc} in the following in detail.

Writes A *Write* operation is a function

$$wr : Write \times AST \times X \to Context$$

that takes a specific write operation $w \in Write$, an AST $t \in AST$ as well as a position $n \in X$ within t as input and returns a value $ctx \in Context$. Our write operations as listed in Figure 5.1 are specified as follows based on Bielik et. al's [1] definitions:

- $wr(\text{WriteValue}, t, n) = ctx$ writes the value of the node n. This value is a terminal symbol from Σ. If the node at position n does not contain a terminal symbol, the special value $\emptyset$ is returned.
- $wr(\text{WritePos}, t, n) = ctx$ returns the index $i \in \mathbb{N}$ of the node at position n within an array of children kept by the parent of n.
- $wr(\text{WriteType}, t, n) = ctx$ writes the type of the node at the current position n as context. The type is a non-terminal symbol from the domain N.
- $wr(\text{HasPrevNodeValue}, t, n) = ctx$ determines whether the value of the current node was previously seen within the same function or the global scope and returns either $true$ or $false$. To formalize this constraint, $wr(\text{HasPrevNodeValue}, t, n)$ can be alternatively written as

$$wr(\text{HasPrevNodeValue}, t, n) = \begin{cases} true, & \exists\, n' < n \wedge wr(\text{WriteType}, t, n) = wr(\text{WriteType}, t, n') \\ false, & else \end{cases}.$$

Given these definitions, a write operation $op \in Write$ is computed as

$$\frac{op \in Write \quad ctx' = wr(op, t, n)}{\langle op :: s, t, n, ctx \rangle \to \langle s, t, n, ctx' \rangle}. \tag{Write}$$

That is, we detach the next operation op that is a *Write* from a program $op :: s$. As op is a write instruction, the context is modified to ctx' leaving $s \in L_{alloc}$ as remaining operations. The abstract syntax tree t and the position within the tree n stay unchanged.

Moves Similar to *Write*, a move is a function

$$mv : Move \times AST \times X \to X.$$

The write operation is replaced by a move operation $m \in Move$ and instead of a context, mv returns a new position $n' \in X$ within t. Following Bielik et al. [1], we define the semantics of move operations:

- $mv(\text{Up}, t, n) = n'$ moves the current position to the parent node n' within t. If n has no parent, $\perp$ is returned.

- $mv(\text{Left}, t, n) = n'$ returns the left sibling n' of n or $\bot$ if n has no left sibling.
- $mv(\text{Right}, t, n) = n'$ returns the right sibling n' of n or $\bot$ if n has no right sibling.
- $mv(\text{DownFirst}, t, n) = n'$ moves to the first child n' of n or returns $\bot$ if n has no children.
- $mv(\text{DownLast}, t, n) = n'$ moves to the last child n' of n or returns $\bot$ if n has no children.
- $mv(\text{PrevNodeValue}, t, n) = n'$ finds the closest node n' with the same node value as n within the same function or in the global program scope. More formally, the function searches for n' such that $n' = \max\{\hat{n} \mid \hat{n} < n\}$ and $wr(\text{WriteValue}, t, n) = wr(\text{WriteValue}, t, n')$ within the same program scope. If no such position n' exists, $\bot$ is returned.
- $mv(\text{PrevNodeType}, t, n) = n'$ is very similar to PrevNodeValue, but instead finds the closest node n' with the same node type as n such that $n' = \max\{\hat{n} \mid \hat{n} < n\}$ and $wr(\text{WriteType}, t, n) = wr(\text{WriteType}, t, n')$. If no prior node with the same value exists within the given scope, again $\bot$ is returned.

The computation of a move is twofold as moves may be successfully returning a new position n' or fail an return $\bot$. A successful move alters the position of the program state from n to n' as

$$\frac{op \in Move \quad n' = mv(op, t, n) \quad n' \neq \bot}{\langle op :: s, t, n, ctx\rangle \to \langle s, t, n', ctx\rangle}. \tag{Move}$$

If a move fails, the program is set to the empty instruction ϵ and the position to $n' = \bot$:

$$\frac{op \in Move \quad n' = mv(op, t, n) \quad n' = \bot}{\langle op :: s, t, n, ctx\rangle \to \langle \epsilon, t, n', ctx\rangle}. \tag{Move-Fail}$$

Actions The *Actions* of an analysis correspond to the leaves of the respective decision tree and can thus be viewed as return statements. The action NewAlloc returns $\{true\}$ as an analysis result, NoAlloc returns $\{false\}$. Lastly, if a branch of an analysis has to be approximated, the action Approx returns $\{true, false\}$.

If-then-else-statements Similar to Bielik et al. [1], we define the formal semantics of if-then-else statements as

$$\frac{\begin{array}{c} op \in \textbf{if } g = ctx \textbf{ then } l_{true} \textbf{ else } l_{false} \\ \langle g, t, n, \emptyset\rangle \to \langle \epsilon, t, n', ctx'\rangle \quad ctx = ctx' \end{array}}{\langle op, t, n, ctx\rangle \to \langle l_{true}, t, n, ctx\rangle} \tag{If-True}$$

and

$$\frac{op \in \textbf{if } g = ctx \textbf{ then } l_{true} \textbf{ else } l_{false} \quad \langle g, t, n, \emptyset \rangle \rightarrow \langle \epsilon, t, n', ctx' \rangle \quad ctx \neq ctx'}{\langle op, t, n, ctx \rangle \rightarrow \langle l_{false}, t, n, ctx \rangle} \quad \text{(If-False)}$$

. The condition checks whether a guard g evaluates to a given context ctx. If the result of the guard is equal to the expected context, the first case *If-True* is applied and the program continues the execution with the instructions l_{true}. Otherwise, *If-False* triggers the execution of the instructions l_{false}.

As a guard is a sequence of *Move* operations and one *Write* operation, the rules for moves and writes apply to the evaluation of a guard. The moves set the current position within the given AST to a node of interest and the write operation determines the value of a guard that is then compared to the context. The subsequent example illustrates how a guard works. At first, we rewrite the candidate analysis of Figure 4.2d on page 14 using the syntax of L_{alloc}. The resulting analysis is depicted in Figure 5.2.

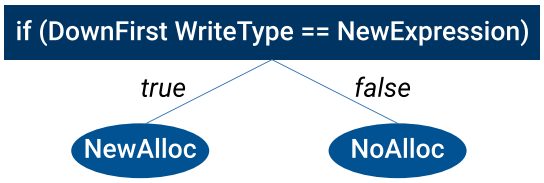


Figure 5.2: Candidate analysis of the example in Figure 4.2 rewritten with the syntax of L_{alloc}.

The guard consists of one move operation DownFirst and one write operation WriteType. The expected context $ctx := NewExpression$ is equal to the AST node type $NewExpression$. The starting position is set to the node $n := VarDeclaration : obj$ of the AST t depicted in Figure 4.2b. Furthermore, the execution context $ctx' := \emptyset$ is initially empty. This initialization results in a starting state $\sigma = \langle \text{DownFirst} :: \text{WriteType}, t, VarDeclaration : obj, \emptyset \rangle$. The computation of the operations of the guard are shown in Figure 5.3 and result in the final state $\sigma' = \langle \epsilon, t, NewExpression : Date, NewExpression \rangle$. The final execution context ctx' of σ' is compared to the expected context ctx of the guard. Both ctx and ctx' are equal to $NewExpression$. Thus, $ctx' = ctx$ holds and the operations l_{true} are executed. In the example, l_{true} consists of the action NewAlloc.

$$\langle \text{DownFirst} :: \text{WriteType}, t, VarDeclaration : obj, \emptyset \rangle$$

$$\xrightarrow{mv(\text{DownFirst},t,VarDeclaration:obj)=NewExpression:Date}$$

$$\langle \text{WriteType}, t, NewExpression : Date, \emptyset \rangle$$

$$\xrightarrow{wr(\text{WriteType},t,NewExpression:Date)=NewExpression}$$

$$\langle \epsilon, t, NewExpression : Date, NewExpression \rangle$$

Figure 5.3: Steps to compute the guard of the analysis depicted in Figure 5.2 for the abstract syntax tree t of Figure 4.2b.

5.2.2 Implementation

As the domain-specific language L_{alloc} operates on ASTs, we use the *ESTree*[1] specification as tree representation. To convert JavaScript code to its ESTree AST representation, we utilize the parser *Acorn*[2]. The implementation of L_{alloc} consists of two modules: the language functionality and an interpreter that is able to execute programs from L_{alloc}.

The language functionality is implemented according to the specification of the semantics in Section 5.2.1. Each *Move* and *Write* is represented by a function that implements the traversal over the AST or the collection of writable information.

To execute a program from L_{alloc}, the interpreter runs through the program instructions recursively. For an action, the interpreter either returns a boolean value *true* or $false$ for NewAlloc and NoAlloc or a specific string in case of approximation. To evaluate an if-then-else-statement, the sequence of move and write operations defined by the guard is executed and the written value is evaluated against the expected context. Based on the evaluation, the execution is either continued with the operations of the then-branch l_{true} or the else-branch l_{false}.

5.3 Data Collection

The supervised approach to learn a decision tree from data requires labeled training samples. A training sample $\langle x^j, y^j \rangle = \langle\langle t, n \rangle, a \rangle \in D$ for the allocation site analysis consists of a program t converted to an AST, a node n within t as position, and a label a that is either *true* if the given position is an allocation site or $false$ otherwise. As manually labeling data is tedious or even infeasible for large amounts of data, we implemented an approach to automatically derive training samples by executing instrumented JavaScript programs. Our approach is inspired by Bielik et al.[1]. It first instruments the source code of programs, runs the executed programs to create a trace of information, and then derives training samples

[1] https://github.com/estree/estree
[2] https://github.com/acornjs/acorn

from the created trace.

Source code instrumentation In our implementation, a position n of a training sample is a node of type *Identifier* as defined by ESTree. Identifiers, for example, can be names of variables or functions. The instrumentation injects custom code into each program. This code contains a function that assigns a unique ID to each JavaScript object. Furthermore, we inject a function that is used to wrap Identifiers, to write information about the wrapped Identifier to the trace, and to finally return the value of the Identifier. The information written includes the unique ID if the value represented by the Identifier is of type object or function. Otherwise, we note that an Identifier is neither of type object nor of type function. The trace is written to a file for each executed program.

For the actual instrumentation, we need to individually decide for an AST node how a respective Identifier is wrapped by the injected function based on the type of the node. For example, an Identifier that names a variable in a variable declaration cannot be directly wrapped by a function since this would result in an invalid syntax. We therefore wrap these Identifiers naming variable declarations in a separate statement that directly follows the variable declaration. On the other hand, Identifiers tied to binary expressions, for instance, are wrapped directly. Our implementation supports all JavaScript language features up to and including ECMAScript 2019[3].

Derivation of training data By executing the instrumented programs, we create a trace for each program that is used to derive training samples. Each entry in the trace corresponds to one Identifier in the source code. For each Identifier, we only select its first appearance in the trace to, for example, account for loop iterations and function calls. From each of these selected Identifiers, we derive one training sample. To decide on the label of a sample, we execute the following steps, which we adapted from the approach of Bielik et al. [1]:

1. If an entry is neither an object nor a function, it is no allocation site and thus labeled with $false$. This applies, for example, to primitive data types such as numbers or strings.
2. If the entry is an object or a function, we check whether the unique ID has been seen before within the trace.
   a) If the ID has been seen before, the object already appeared earlier within the source code, is not considered an allocation site, and thus receives the label $false$.
   b) Otherwise, the entry corresponds to the first appearance of an object and is therefore labeled with *true* as a new allocation site.

Given an entry of a trace for a program, we create a training sample $\langle\langle t, n\rangle, a\rangle$ with the AST t of the program, the Identifier as node n, and the computed label a.

[3] https://262.ecma-international.org/10.0/

5.4 Program Synthesis

We implemented the synthesis according to Algorithm 1, presented in Section 4.1.3. The synthesis receives a dataset D that is generated as described in the previous section as input. The core of the implementation consist of the functions *genAction* and *genBranch*. In the following, we describe the implementation of these two functions and additionally the special case of approximation.

genAction To find the best action a_{best}, the implementation calculates the cost for the two actions NewAlloc and NoAlloc on the dataset D. Subsequently, the action with the lowest cost is returned.

genBranch For branch generation, we need to find a guard that yields a high information gain. In the first step, our tool preselects potential guards that are further investigated. As guards according to the syntax of L_{alloc} consist of a theoretically infinite amount of move operations, we need to narrow the search space for potential guards to trade off between a high information gain and the runtime of the synthesis. Bielik et al. [1] propose to evaluate programs up to size six consisting of five move operations and one write operation as guards. We tried different guard constellations ranging from size four (three moves and one write) up to size six (five moves and one write). To evaluate a specific setting, we look into the information gain of guards and the required amount of branches of a learned candidate analysis. A low amount of branches indicates that we are able to find guards with a suitable information gain that recursively split the dataset into two comparably large subsets. Therefore, we aim for a setting that minimizes the amount of branches. Based on our test of different settings for guards, we decide to use a preselection of $1,000$ uniformly sampled guards from all possible guards up to size six. A preselection that uses more or different sized guards does not have a positive impact on the amount of analysis branches and the information gain.

In the next step, we determine the expected context ctx of guards that is the right hand side of the condition. For this, our implementation runs each of the 1,000 preselected guards on the current dataset and identifies guard-context combinations that appear most often. Each guard-context combination consists of a guard with up to six operations and an expected context as right hand side of the condition.

In the final step, our tool calculates the information gain for a select amount of guard-context combinations. Bielik et al. [1] use the ten guard-context combinations that appear most often and calculate the information gain for each of these guards. However, for our dataset, this selection approach proves to be too fragile and results in many approximated branches. For these branches, the information gain for all ten selected guards is zero. We therefore adapted the approach to select the 40 guard-context combinations that appear most often and calculate their information gain. If none of the 40 guards yield a positive information gain, the procedure is repeated up to ten times so that up to 400 guards are evaluated.

With this selection procedure, we assure that only few branches have to be approximated. The reasoning behind the selected values of 40 guards per chunk and up to ten repetitions is the balance between the runtime of the learning procedure and the quality of the selected guards. In general, a bigger chunk size evaluates the information gain of more guards. Thus, with more analyzed guards, a guard with a higher information gain may be found. However, calculating the information gain is computationally expensive. We therefore empirically settled for a chunk size of 40 guards to find a guard with suitable information gain in a reasonable amount of time. Only if none of these initial 40 guards provides any information gain, we analyze up to 360 additional guards. In up to ten iterations, we find a suitable guard with positive information gain in most cases and rarely need to approximate analysis branches.

approximate There usually are still a few branches where approximation is required due to noisy data. In this case, our implementation returns the action Approx.

5.5 Oracle

The oracle creates new training samples given a candidate analysis *pa* and a dataset D. Our implementation follows Section 4.2 and, as a first step, determines positions for modifications within the programs in D. Then, it executes selected modifications to find new samples that are incorrectly labeled by *pa*.

5.5.1 Determine Modification Positions

To determine the positions for program modification, we run the analysis *pa* for all training samples $\langle\langle t, n\rangle, a\rangle$ from our dataset D. During the execution for a training sample, we collect all visited nodes from the AST t of this sample. Hence, we generate a set of nodes to modify for each training sample in D. From these sets, we remove the node n which itself should not be modified.

5.5.2 Selection of Modifications

We select the modifications in line with Bielik et al. [1]. Table 5.1 gives an overview of both the semantic-preserving modifications F_{sp} and the modifications that might alter the semantics F_{sa}. We again do not have any information about the authors' implementation of these modifications which might therefore diverge from our implementation.
We consider the following for the two types of modifications in our implementation:

Semantic-preserving modifications From Section 4.2.2 we know that the label of modified training samples a is not influenced by the modification and stays the same. Moreover, the node n within the sample remains unchanged after the modification. The applied changes are thus only reflected in the code and the corresponding AST t. To make sure that we do not unintentionally provide structural patterns within the created

Table 5.1: Selected program modifications for the oracle that preserve program semantics F_{sp} and that may alter semantics F_{sa}.

Program Modifications	
F_{sp}	F_{sa}
Insertion of dead code	Addition of function arguments
Renaming of variables	Addition of function parameters
Renaming of functions	Changing of constant values
Addition of expressions	

modifications, the selection of the specific modifications is randomized. If, for example, variables are always renamed to the same identifier, the analysis might pick up this regularity and create a rule from it. Instead, we use random identifiers of variable length to rename variables and functions. For both the insertion of dead code and the addition of expressions, we provide a selection of code snippets to randomly choose from.

Modifications that may not be semantic preserving Even though these modifications could alter the program semantics, we keep the label and the node of a modified sample unchanged. Thus, we implement the modifications such that the analysis result remains the same. We made the deliberate decision to not make any changes to the label of a training sample to avoid forging training samples with a predefined label. We again use identifiers of variable length and composition as additional function parameters. Additional function arguments are also either generated identifiers or constant values that are not objects and thus cannot directly influence the label of a training sample. The value of constants is changed to a random number, string, or boolean.

Each of the program modifications is applied to all suitable nodes that we previously determined. If one of these nodes, for instance, is a variable or function declaration, we can rename it. To select new training samples from the created modifications, we check if the analysis *pa* produces the correct result *true* or $false$ for each modified sample. We then select the samples that are incorrectly classified by *pa* to further refine the analysis.

5.6 Line-Based Analysis for Allocation Sites

To put the learned rules for the allocation site analysis into practice, we implemented a static code analysis that examines lines of source code for allocation sites. This analysis can for example be used to find bottlenecks in memory-intensive applications or to prevent potential memory leaks early on during development.

The line-based analysis works on lines of individual code files. For each line of code, the analysis determines whether the line is considered an allocation site. To distinguish between

allocation sites and non-allocation sites, we use the learned analysis rules. These rules are applied to each Identifier of the ESTree AST representation of the code file. If a line of code includes at least one Identifier that is considered an allocation site according to the analysis rules, it is altogether labeled as an allocation site. Otherwise, if a line of code either does not include an Identifier or no Identifier is rated as allocation site, the line is not considered an allocation site.

Regarding sensitivities, the line-based analysis is not context or path sensitive. We do neither differentiate between executions based on additional context information nor between different execution paths. However, the analysis is flow sensitive and partly depends on the order of statements in a given code file. The underlying analysis rules, for example, consider whether a specific Identifier has been seen and initialized before to determine whether the Identifier is an allocation site.

6 Evaluation

We evaluate our approach and implementation answering six research questions. To learn and test analysis rules for allocation sites, we generate a dataset as described in Section 5.3. Our evaluation thereby goes beyond the initial evaluation of Bielik et al. [1] who evaluated the learned analysis rules for the allocation site analysis by manually inspecting selected rules. Overall, the evaluation of Bielik et al. for the allocation site analysis is rather sparse. We thus conduct our own evaluation by examining accuracy, precision, and recall of synthesized analyses. In addition to the generated dataset, we use validation data extracted from code of open source projects to evaluate analysis rules. In some research questions of our evaluation, we refer to the results of Bielik et al. for comparison. However, we cannot conduct a detailed comparison due to a lack of data as Bielik et al. do not provide metrics such as accuracy, precision, or recall for the allocation site analysis.

6.1 Research Questions

This section describes the research questions that are answered in the evaluation.

RQ 1: To what degree does the learned analysis approximate results?

The first question aims to clarify if we are able to learn analysis rules that are not only sound, but also precise for the dataset used for training. That is, we want to analyze whether a learned analysis is able to properly map the training samples to their respective labels or if many analysis branches have to be approximated at the cost of precision.

RQ 2: How well does the learned analysis generalize for a train-test split of a given dataset?

In the next step, we examine the generalization of a learned analysis for a test set that is a subset of the generated dataset. The goal of this RQ is to evaluate the analysis generalization for data that is not used for training but is structurally similar to the training data.

RQ 3: To what extent does the oracle improve the analysis generalization for a train-test split of a given dataset?

Building on RQ 2, we aim to investigate in RQ 3 whether the oracle improves the generalization of learned analysis rules for test data that is part of the generated dataset but not used for the learning procedure.

RQ 4: How well does the learned analysis generalize to data beyond the given dataset?

RQ 4 analyzes the generalization of a learned analysis for data beyond the generated dataset. That is, the analysis is learned from the generated dataset, but unlike RQ 2, the data to evaluate the analysis is not part of this dataset. With this RQ, we want to find out if the learned analysis rules are applicable to a practical static analysis.

RQ 5: To what extent does the oracle improve analysis generalization beyond the given dataset?

In RQ 5, we separately asses the effect of the oracle on the generalization of a learned analysis on data that is not a subset of the generated dataset.

RQ 6: What is the impact of the size of the training dataset on the quality of the learned analysis?

Finally, we want to find out whether more training data leads to an improved analysis quality in RQ 6. This question seeks to determine whether more data to synthesize an analysis improves the generalization for test samples form the generated dataset and for samples beyond that dataset. This question is relevant because the generation of training data is costly and it is therefore beneficial to use as little data as possible to obtain a high quality analysis.

6.2 Study Objects

For the evaluation, we use two different datasets. The first is a dataset of training samples D_T for the allocation site analysis that is generated according to our data collection procedure described in Section 5.3. The generated dataset is used to train and test analyses. Second, we collect validation data D_V from open source projects and manually crafted validation samples to further evaluate analysis rules on data beyond the generated dataset.

Generated dataset D_T for training and testing We derive the dataset D_T that is used to train and test analysis rules from the *ECMAScript Test Suite*[1]. We therefore instrument the source code and subsequently execute each test case. To generate the dataset, we mainly use the ECMAScript Test Suite for three reasons:

1. The ECMAScript Test Suite was also used by Bielik et al. [1] to generate a dataset. As we aim to reproduce the results, we want to create a comparable dataset.
2. The test suite comprehensively covers built-in JavaScript functionality and related edge cases that should be covered by analysis rules.
3. Our approach to generate training data requires executable programs. Therefore, we benefit from the many small and easily executable programs included in the ECMAScript Test Suite.

[1] https://github.com/tc39/test262

We instrument a total of 31,318 test files and derive 125,130 training samples. The detailed composition of D_T is displayed in Table 6.1.

Table 6.1: Composition of the generated dataset D_T used to learn and test rules for the allocation site analysis.

Composition of D_T		
Allocation Sites	Non-Allocation Sites	Total Samples
42,052	83,078	125,130

Validation data D_V Besides the generated dataset D_T, we create additional data for analysis validation D_V from open source code and manually crafted code snippets. We derive validation data from the code of the two open source JavaScript projects *Axios*[2] and *Strapi*[3]. Axios is a promise-based HTTP client both for the browser and Node.js. Strapi is a Content Management System (CMS) to make content easily accessible via custom APIs. Both projects are actively maintained and popular with 88.5 and 40.3 thousand stars on Github. Thus, the two projects contain code that is highly relevant for an examination.

We manually evaluate randomly selected code files from both projects. Within each of the files, we mark lines of code either as allocation sites or as non-allocation sites by hand. Whether or not a line is considered an allocation site follows the definition in Section 5.6: a line of code is rated as allocation site if we at least consider one Identifier within the line to be an allocation site. Thus, we can use the validation data to evaluate the line-based analysis and to thereby examine the underlying analysis rules. In addition to the data from open source projects, we manually craft samples to specifically validate analysis rules for built-in JavaScript functionality and edge cases. Table 6.2 provides an overview of the open source projects and the structure of the validation data. We analyzed 341 lines of code (LOC) from Axios and Strapi. Furthermore, we added 30 LOC of manually crafted samples. Table 6.2 additionally displays the number of LOC that include at least one Identifier. As the underlying analysis rules of the line-based analysis target Identifiers, we evaluate the accuracy, precision, and recall of the analysis only on the 146 LOC of D_V that include Identifiers. Thus the calculation, for instance, does not consider empty lines or lines that only contain a curly bracket as correctly classified non-allocation sites to avoid artificially increasing the accuracy.

[2]https://github.com/axios/axios
[3]https://github.com/strapi/strapi

Table 6.2: Composition of the additional validation data D_V to assess analysis rules. The data is extracted from open source projects and augmented with manually created data points. The table shows the project names, the total number of analyzed lines of code (LOC), the number of LOC that contain at least one identifier as well as the number of allocation and non-allocation sites.

	Validation Data			
Project	LOC Total	LOC with Identifiers	Allocation Sites	Non-Allocation Sites
Axios	196	79	29	167
Strapi	145	41	16	129
Manually Crafted	30	26	14	16
Combined	371	146	59	312

6.3 Study Procedure

We now describe our approach to answer the research questions introduced in Section 6.1. The setup used for the evaluation is summarized in Table 6.3. For RQ 1, RQ 2, and RQ 3, we use train-test splits of the dataset D_T to evaluate different analysis properties. We split D_T or selected subsets of D_T into two distinct sets D_T^{train} and D_T^{test}. D_T^{train} is used as training data to synthesize the analysis and contains 80% of the selected samples. The test set D_T^{test} consists of the remaining 20% of the data and is used to examine various analysis properties depending on the research question. The samples for D_T^{train} and D_T^{test} are selected uniformly at random. For RQ 4 and RQ 5, we synthesize the analysis from the complete dataset D_T. We then evaluate the learned analysis on the validation set D_V. To answer RQ 6, we use both train-test splits of D_T and the validation set D_V.

RQ 1: To what degree does the learned analysis approximate results?

To evaluate whether we are able to learn a precise analysis for the data that is used in the synthesis procedure, we analyze the number of approximated analysis branches. A branch is considered approximated if its leaf contains the action Approx. Each branch of an analysis that results in approximation reduces the precision of the analysis for the training data as at least one sample is mapped to the approximated branch. Furthermore, we evaluate the number of samples that end up in approximated branches. For this, we use train-test splits of D_T to see how many test samples from D_T^{test} are approximated. We examine the number of approximated branches for analyses synthesized from training sets D_T^{train} of various sizes from $10,000$ to $100,104$ training samples. The number of approximated samples is then evaluated on the respective test sets D_T^{test}.

Table 6.3: Summary of the setup for the evaluation. Sets of training data D_T^{train} and test data D_T^{test} are non-overlapping subsets of D_T. From validation set D_V, we only use code lines that include at least one Identifier.

	Study Setup	
RQ	**Training Data**	**Evaluation Data**
1	D_T^{train}: 10,000 to 100,104 samples	D_T^{test}: 2,500 to 25,026 samples
2	D_T^{train}: 100,104 samples	D_T^{test}: 25,026 samples
3	D_T^{train}: 10,000 to 40,000 samples + 5,000 to 20,000 samples from 5 oracle iterations	D_T^{test}: 2,500 to 10,000 samples
4	D_T: 125,130 samples	D_V: 146 LOC with Identifiers
5	D_T: 125,130 samples + 50,000 samples from 5 oracle iterations	D_V: 146 LOC with Identifiers
6	D_T^{train}: 10,000 to 100,104 samples + 5,000 to 20,000 samples from 5 oracle iterations	D_T^{test}: 2,500 to 25,026 samples + D_V: 146 LOC with Identifiers

RQ 2: How well does the learned analysis generalize for a train-test split of a given dataset?

For the second research question, we aim to create the best possible analysis and therefore use the maximum amount of training data from D_T while still retaining some data for testing. We split the complete dataset D_T of 125,130 samples into datasets D_T^{train} (80%) to train and D_T^{test} (20%) to evaluate the analysis. We do not refine the learned candidate analysis using the oracle, but directly examine the analysis synthesized from D_T^{train}. We calculate the metrics accuracy, precision, and recall of the analysis synthesized from D_T^{train} for the corresponding test set D_T^{test}. As the samples of D_T^{test} are not used to train the analysis, we use these three metrics to draw conclusions about analysis generalization for data included in D_T.

RQ 3: To what extent does the oracle improve the analysis generalization for a train-test split of a given dataset?

In difference to RQ 2, we train and evaluate multiple analyses on smaller subsets of D_T split into D_T^{train} and D_T^{test}. We rely on these smaller subsets to be able to perform more oracle iterations. While we train and refine an analysis on the complete dataset D_T to answer RQs 4 and 5, we cannot use this analysis to answer this RQ as no data for testing remains. Synthesizing analyses for large amounts of data is computationally expensive and requires costly hardware. A further effort to synthesize and refine an analysis consisting of 80% of the complete dataset is thus beyond the scope of this thesis.

The initial candidate analyses used to answer this RQ are refined over five oracle iterations. To prevent the size of D_T^{train} from increasing too much with each iteration, we limit the maximum number of newly added samples to 10% of the initial size of D_T^{train}. If the limit

is reached, the maximum number of samples is randomly selected from all the samples generated by the oracle.

RQ 4: How well does the learned analysis generalize to data beyond the given dataset?

To answer RQ 4, we use the line-based analysis and the validation set D_V. We first synthesize analysis rules from all available training samples D_T without oracle refinement. These rules are then used by the line-based analysis. We execute the line-based analysis on code of the two open source projects Axios and Strapi as well as manually crafted code. Subsequently, we compare the results from the execution with the pre-labeled dataset D_V. For the comparison, we calculate the accuracy, precision, and recall of the line-based analysis individually for each project and combined for all projects. To calculate accuracy, precision, and recall, we only consider lines of code that at least include one Identifier and are thus not trivially considered non-allocation sites.

RQ 5: To what extent does the oracle improve analysis generalization beyond the given dataset?

In difference to the procedure described in RQ 4, we refine the synthesized analysis in five oracle iterations. To again avoid an exploding size of the training data after each iteration, we limit the number of newly added samples within an oracle iteration to $10,000$. After each iteration, we record the intermediate analysis. We then evaluate all recorded analyses as part of the line-based analysis as detailed in the procedure of RQ 4.

RQ 6: What is the impact of the size of the training dataset on the quality of the learned analysis?

For RQ 6, we investigate how accuracy, precision, and recall of a learned analysis change with an increasing size of the training dataset. We again train and evaluate multiple analyses on randomly sampled subsets of D_T that are split into training data D_T^{train} and test data D_T^{test}. The size of the subsets range from $12,500$ samples ($10,000$ training samples | $2,500$ test samples) to $125,130$ ($100,104$ training samples | $25,026$ test samples).

Besides the test sets D_T^{test}, we examine the analyses' quality with the help of D_V. For this, we use each of the learned analyses as rules in the line-based analysis and compare the resulting accuracy, precision, and recall on D_V. We again only consider lines of code that include Identifiers.

6.4 Results

We provide the results for the research questions following the procedures described in the previous section.

RQ 1: To what degree does the learned analysis approximate results?

Table 6.4 summarizes the results for RQ 1. The absolute number of approximated branches ranges from 7 for an analysis trained with 20,000 samples to 35 for 125,130 training samples. This corresponds to a percentage of 0.64% to 1.22% and an arithmetic mean of 0.98% of approximated branches. The evaluation of analyses on test data results in 19 to 138 approximated test samples which is between 0.27% and 0.96% of the overall amount of test data. On average, 0.54% of the samples of the test set are approximated.

These results show that comparably little approximation is required as on average 0.98% of branches and 0.54% of samples are approximated. However, the number of approximated samples is not negligible as it may have a significant impact on the precision or recall. For an approximated sample, we cannot clearly state whether it is an allocation site. Thus, approximated samples are counted either as false positive or false negative predictions. The results in RQ 2 show that precision drops by 1.57 percentage points in the worst case when considering all approximated samples as false positives. In this case, the recall remains the same. On the other hand, if we rate all approximated samples as false negatives, the recall decreases by 1.58 percentage points. In summary, we conclude that we can learn an analysis that is precise for its training data and does only approximate a small number of branches and samples. Nevertheless, we need to consider the approximated samples as incorrect predictions in practice since they negatively impact the quality of the analysis rules.

Table 6.4: Amount and percentage of approximated samples of D_T^{test} for analyses synthesized from different sets of training data D_T^{train}. In addition, the absolute and relative amount of approximated analysis branches is displayed.

			Approximated Samples and Branches		
Total Data	D_T^{train}	D_T^{test}	Approximated Test Samples	Analysis Branches	Approximated Branches
12,500	10,000	2,500	24 (0.96%)	745	8 (1.07%)
25,000	20,000	5,000	19 (0.38%)	1,096	7 (0.64%)
50,000	40,000	10,000	27 (0.27%)	1,827	18 (0.99%)
125,130	100,104	25,026	138 (0.55%)	2,870	35 (1.22%)

RQ 2: How well does the learned analysis generalize for a train-test split of a given dataset?

As Table 6.5 reveals, the accuracy of the analysis synthesized from D_T^{train} and evaluated on D_T^{test} is 98.02%. The calculation of the accuracy also considers the 138 approximated samples as incorrectly classified by the analysis. The precision of the analysis is 97.75% and the recall is 98.02%. For both precision and recall, we do not consider the approximated samples as we cannot explicitly classify them as either false negatives or false positives. If we rate all of the 138 approximated samples as false positives, the precision drops to a lower bound 96.18%. Otherwise, if we classify the approximated samples as false negatives, the recall decreases by 1.58 percentage points to its lower bound of 96.44%.

Table 6.5: Accuracy, precision, and recall of the analysis synthesized from D_T^{train} and evaluated on D_T^{test}.

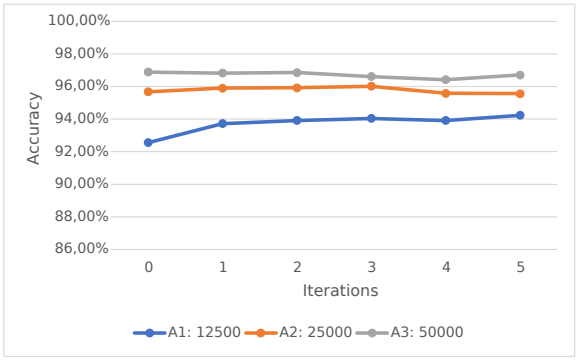
Analysis generalization for D_T					
Total Data	D_T^{train}	D_T^{test}	Accuracy	Precision	Recall
125,130	100,104	25,026	98.02%	97.75%	98.02%

The high precision of at least 96.18% shows that we can reliably detect allocation sites. In addition, the high recall with a lower bound of 96.44% confirms that non-allocation sites are mostly classified correctly. We still have to consider that 2 to 4 false positives within 100 samples might be too many, for example if the analysis rules are used as part of a larger static analyzer. In this case, the missing precision of different analysis components could add up and lead to a static analysis that is too imprecise for practical application. Altogether we infer from the comparably high values for accuracy, precision, and recall that the synthesis procedure derives analysis rules that generalize well for structurally similar data.

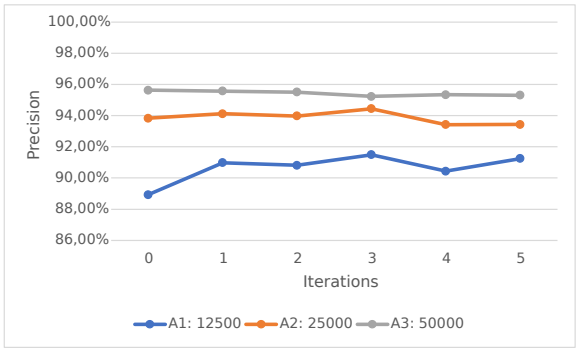
RQ 3: To what extent does the oracle improve the analysis generalization for a train-test split of a given dataset?

The line charts in Figure 6.1 show the development of the accuracy (Figure 6.1a), precision (Figure 6.1b) and recall (Figure 6.1c) for three analyses A_1 (12,500 samples), A_2 (25,000 samples), and A_3 (50,000 samples) over five oracle iterations. All three analyses are trained on a subset of the generated dataset D_T. The first synthesis, iteration 0, uses 80% of the 12,500, 25,000, and 50,000 samples, respectively, as training data and the remaining 20% for the evaluation. In each iteration, the oracle adds the maximum of 10% of the initial training samples to the training data. For example, for A_1, 1,000 samples are added in each iteration.Thus, in the final iteration, A_1 is synthesized from a total of 15,000 training samples consisting of the initial 10,000 samples and an additional 5,000 samples created by the oracle.

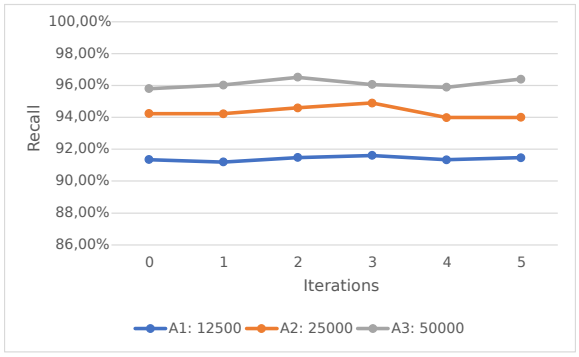
The variation of accuracy, precision, and recall over the course of five oracle iterations is



(a) Accuracy



(b) Precision



(c) Recall

Figure 6.1: Development of the accuracy, precision, and recall for analyses A_1, A_2, and A_3 synthesized and evaluated on datasets of $12,500$, $25,000$, and $50,000$ samples for 5 oracle iterations. Iteration 0 denotes the evaluation of the analysis after the initial synthesis. To make the small differences between the three analyses visible, the scale of the y-axis of the diagrams does start at 86% instead of 0.

small. For A_1, the accuracy ranges from 92.56% in the initial iteration 0 to 94.24% in the final iteration 5. The precision is between 88.93% (iteration 0) and 91.49% (iteration 3). The lowest value for the recall is 91.20% (iteration 1) and the highest value is 91.60% (iteration 3). Analysis A_2 has an accuracy of 95.56% (iteration 5) to 96.02% (iteration 3), the precision ranges from 93.42% (iteration 4) to 94.44% (iteration 3), and the recall is between 93.99% (iterations 4 and 5) and 94.90% (iteration 3). Finally, the accuracy of A_3 is between 96.42% (iteration 4) and 96.89% (iteration 0), precision between 95.23% (iteration 3) and 95.62% (iteration 0), and the recall ranges from 95.80% (iteration 0) to 96.52% (iteration 2).

The results of our experiments do not indicate a consistent improvement of the analysis generalization for analyses A_1, A_2, and A_3 that are refined with the oracle. The iteration with the maximum and minimum value for accuracy, precision, and recall varies for the three analyses. For analysis A_1, the accuracy is lowest before the first oracle iteration at 92.56% and highest after five iterations at 94.24% which indicates an improvement of the accuracy with an increasing number of iterations. For analysis A_3, however, the accuracy reaches its minimum of 96.42% in iteration 4 and its maximum of 96.89% before even executing the oracle once in iteration 0. Besides these variations, the difference between maximum and minimum values of accuracy, precision, and recall for subsequent oracle iterations is comparably small and ranges from 0.39 to 2.56 percentage points with an arithmetic mean of 0.96 percentage points. Looking at all three analyses A_1, A_2, and A_3, we cannot identify a consistent improvement of either accuracy, precision, or recall over five oracle iterations.

A possible explanation why the oracle does not improve the analysis generalization is the increasing number of analysis branches and thus analysis rules with each iteration. For analyses A_1, A_2, and A_3, the number of branches after iteration 5 is significantly higher than after the first synthesis in iteration 0. Figure 6.2 depicts the increase of analysis branches over the five iterations.

The goal of the oracle is to avoid analysis branches that do not generalize well beyond the training data. Thus, we would expect the oracle to consolidate analysis branches that are based on properties specific to the training data. However, instead of a branch consolidation, the increasing number of analysis branches hints towards the addition of new rules for the samples created by the oracle. Therefore, the oracle does not lead to more generic, but rather to more specific rules that might finally decrease accuracy, precision, and recall of an analysis.

RQ 4: How well does the learned analysis generalize to data beyond the given dataset?

The line-based analysis that uses rules learned from the dataset D_T has an accuracy of 76.03%, a precision of 72.22%, and a recall of 66.10% combined for all projects and the manually crafted samples from D_V. Table 6.6 provides a detailed overview of the accuracy, precision, and recall individually for each project, combined for the open source projects Axios and Strapi, and cumulatively for all projects. For the project Strapi, the line-based analysis has the overall highest accuracy of 80.49% and the highest recall of 75.00%. The maximum precision is 88.89% for the manually crafted code samples. The accuracy and recall are lowest for the manual samples at 73.08% and 57.14%, respectively. The analysis has the lowest precision for

Axios at 65.52%.

Figure 6.2: Development of the number of analysis branches for the three analyses A_1, A_2, and A_3 over five oracle iterations.

Table 6.6: Accuracy, precision, and recall of the line-based analysis on validation set D_V. The underlying analysis rules are synthesized from D_T without oracle refinement.

Analysis generalization for D_V			
Project	**Accuracy**	**Precision**	**Recall**
Axios	74.68%	65.52%	65.52%
Strapi	80.49%	75.00%	75.00%
Manually Crafted	73.08%	88.89%	57.14%
Axios + Strapi	76.67%	68.89%	68.89%
Total	**76.03%**	**72.22%**	**66.10%**

Our results show that accuracy, precision, and recall significantly drop for the evaluation of samples from validation set D_V in comparison to structurally similar samples from the generated dataset D_T. For the open source projects Axios and Strapi, 76.67% of the lines that include at least one Identifier are accurately classified by the line-based analysis. In comparison, as we saw in RQ 2, the accuracy for data from a test set from D_T was 98.02%. With 68.89%, precision and recall are even lower than the accuracy due to many false positives and false negatives produced by the line-based analysis. In Figure 6.3, we provide examples of false positives and false negatives taken from the evaluated code of Axios and Strapi. The example in Figure 6.3a shows a false positive as the variable declaration of *PORT* is considered an allocation site even if it is initialized with a number. A number is a primitive data type in JavaScript and does not allocate new heap memory. We expect the analysis rules to be precise for this type of variable declaration. In contrast, in the example in Figure 6.3b,

the variable *url* is initialized by calling a user-defined function *getAbsoluteAdminUrl* which returns a string which again is a primitive data type. While this example is a false positive, it is a shortcoming of the line-based analysis which does not evaluate the actual body of the called function and we thus cannot expect this case to be covered by the analysis rules.

```
const PORT = argv.p || 3000;
```

(a) False positive: variable initialization with a number.

```
const url =
getAbsoluteAdminUrl(strapi.config);
```

(b) False positive: constant initialization via a function that returns a string.

```
async function pingDashboard(url,
    multipleTime = false) {...}
```

(c) False negative: function declaration.

```
var fs = require('fs');
```

(d) False negative: module import.

Figure 6.3: Code lines from Axios and Strapi that were incorrectly classified as either false positives or false negatives.

In the example of Figure 6.3c, a new function *pingDashboard* is declared. We consider this line to be a false negative as declaring a function allocates new heap memory, but it is rated as non-allocation site by the line-based analysis. Similarly, the code shown in Figure 6.3d implements a module import that leads to an allocation of new heap memory, but is not considered an allocation site by the line-based analysis. We would expect both these false negative cases to be covered by the learned analysis rules.

With an accuracy of 73.08%, the evaluation of the line-based analysis on manually crafted code yields a high precision of 88.89% at the cost of recall which is at 57.14%. In terms of edge cases, the analysis incorrectly considers the call to the object constructor with an argument of type object as seen in Figure 4.4 on page 20 as an allocation site. This edge case is covered by our training data and should thus not be missed. In contrast, the rules shown by Bielik et al. [1] do consider this edge case. Analogously, the analysis partially handles built-in JavaScript functions that are included in the ECMAScript Test Suite incorrectly. False negative predictions, for instance, contain calls to built-in functions like *Object.entries* or *Array.filter*. Both these functions create a new array and thus allocate new heap memory, but are not consistently rated as allocation sites. The incorrect classification of edge cases and built-in functionality that is covered by training data indicates an overfitting of the learned analysis rules to the training data.

RQ 5: To what extent does the oracle improve analysis generalization beyond the given dataset?

The line charts in Figure 6.4 summarize the development of the accuracy, precision, and recall of the line-based analysis for Axios and Strapi (Figure 6.4a), the manually crafted code samples (Figure 6.4b), and for all projects combined (Figure 6.4c). For the code of the open source projects Axios and Strapi, the accuracy of the line-based analysis reaches a minimum

of 70.00% (iteration 3) and a maximum of 80.00% (iteration 4). Precision and recall range from 58.49% (iteration 3) to 71.43% (iteration 4) and 68.89% (iterations 0, 3 and 5) to 77.78% (iterations 1 and 4), respectively.

Furthermore, the line-based analysis has an accuracy between 57.69% (iteration 5) and 76.92% (iteration 1) for the manually crafted samples. The precision is lowest at 61.54% (iteration 5) and highest at 88.89% (iteration 0). The recall varies from 57.14% (iterations 0 and 5) to 85.71% (iterations 1, 2 and 4).

In total, the accuracy is between 68.49% (iteration 5) and 78.77% (iteration 1), the precision ranges from 60.00% (iteration 5) to 72.22% (iteration 0), and the recall is between 66.10% (iterations 0 and 5) and 79,66% (iterations 1 and 4).

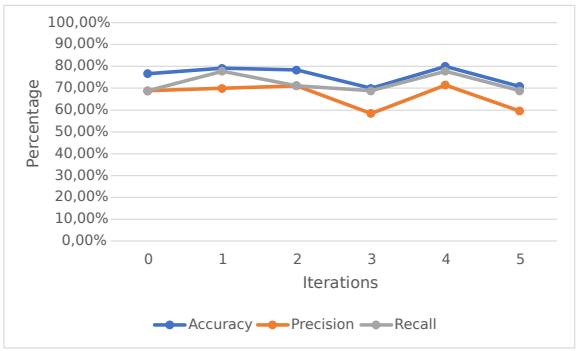
Similar to RQ 3, there is no consistent improvement of the analysis generalization with an increasing number of oracle iterations. On the contrary, we see that accuracy and precision of the line-based analysis evaluated on all samples of validation data D_V decrease by 7.53 and 12.22 percentage points, respectively, from iteration 0 to iteration 5. At the same time, the number of analysis branches of the underlying analysis rules increases from 3,274 branches in iteration 0 to 8,698 branches in the final iteration 5. In comparison, Bielik et al. only report 135 branches for their analysis trained with 134,721 training samples. Instead of becoming more generic, our analysis rules seem to become more specific with each oracle iteration. This increased specificity might lead to more overfitting of the analysis to the training data and thus to a decrease in generalization.

In comparison to RQ 4, the oracle neither improves precision nor recall for edge cases and built-in functionality. The precision of the line-based analysis decreases by 27.35 percentage points over the five oracle iterations while the recall remains the same.

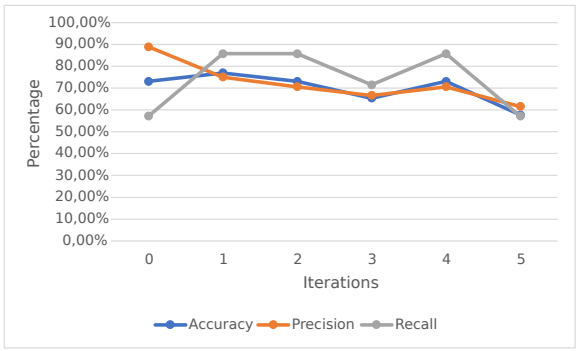
RQ 6: What is the impact of the size of the training dataset on the quality of the learned analysis?

We first look at the results of analyses that are learned from training data D_T^{train} and evaluated on test data D_T^{test}. The line charts in Figure 6.5 reveal that the accuracy, precision, and recall increase with the number of training samples for both the rules synthesized with (Figure 6.5b) and without (Figure 6.5a) oracle refinement. Without oracle refinement, the accuracy for an analysis synthesized from 10,000 training samples and evaluated on 2,500 test samples is 92.56% and increases to 98.02% for 100,104 training and 25,026 test samples. Similarly, the precision increases from 88.93% to 97.75% and the recall from 91.35% to 98.02%.

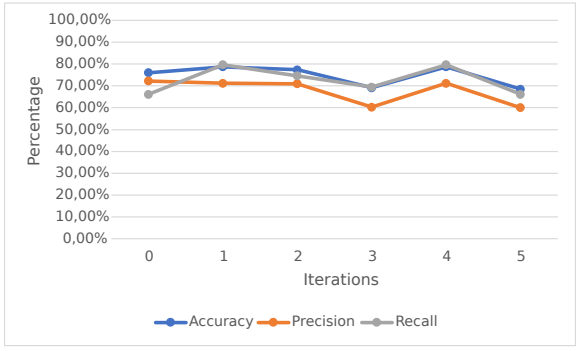
For the case with oracle refinement over five iterations, we evaluate analyses synthesized from 10,000 to 40,000 samples of training data on 2,500 to 10,000 test samples. Analogous to the case without oracle refinement, the accuracy of 94.24%, precision of 91.25%, and recall of 91.47% for 10,000 training samples (2,500 test samples) increases to 96.71%, 95.31%, and 96.40% for 40,000 training samples (10,000 test samples). The results of the evaluation of the analyses on test data D_T^{test} thus show that accuracy, precision, and recall steadily improve with an increasing amount of training data D_T^{train} for both analyses with and without oracle refinement. From this improvement, we deduce that the number of training samples positively impacts analysis quality for structurally similar data from the dataset D_T.



(a) Axios and Strapi

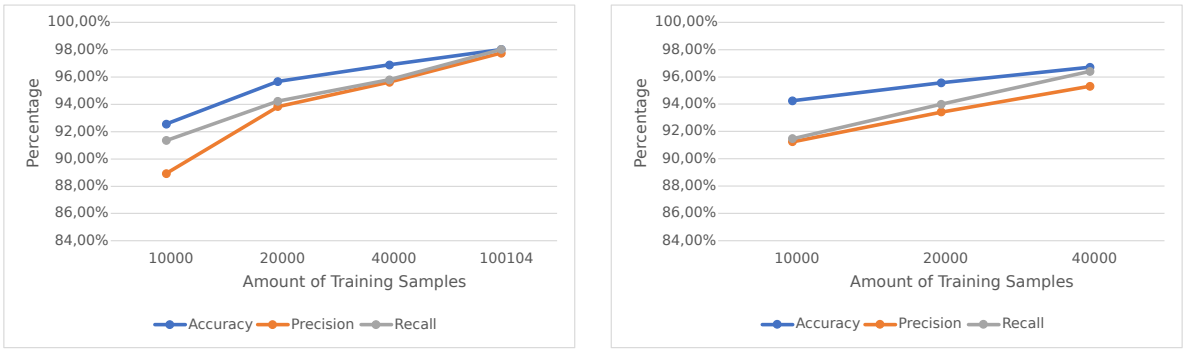


(b) Manually Crafted



(c) Total

Figure 6.4: Development of accuracy, precision and recall of the line-based analysis using analysis rules refined over five oracle iterations. The three metrics are calculated for the code excerpts from Axios and Strapi, the manually crafted samples, and for all three projects combined.



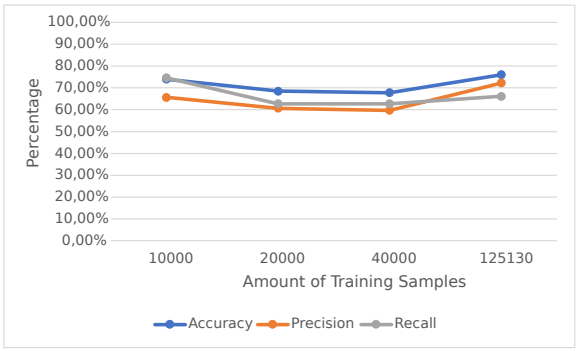
(a) Analysis that is synthesized without oracle refinement.

(b) Analysis that is refined over five oracle iterations.

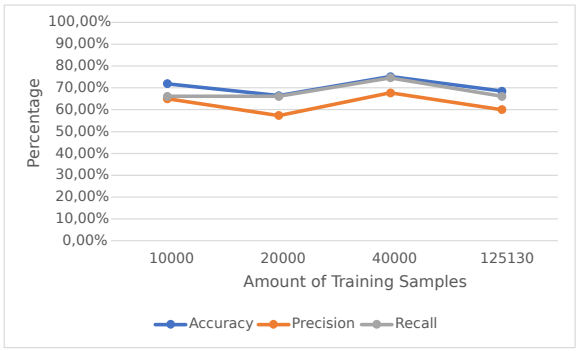
Figure 6.5: Comparison of the metrics accuracy, precision, and recall of synthesized analysis rules based on a train-test split. The analyses are synthesized from different amounts of training data D_T^{train} with and without oracle refinement. To evaluate the analyses, the corresponding test sets D_T^{test} are used. The y-axis deliberately starts at 84% to better differentiate visually between the accuracy, precision and recall.

Besides the evaluation on test sets D_T^{test}, we examine the analysis rules on the dataset D_V as part of the line-based analysis. The charts in Figure 6.6 summarize the results for line-based analyses with the underlying rules synthesized from 10,000 to 125,130 training samples. Without oracle refinement (Figure 6.6a), the accuracy is lowest for an analysis based on 40,000 training samples with 67.81% and highest for 125,130 samples with 76.03%. The precision varies between 59.68% for 40,000 training samples and 72.22% for 125,130 samples. The analyses with rules synthesized from 20,000 and 40,000 training samples provide the lowest recall of 62.71%. The highest recall is 74.58% for 10,000 training samples. For the case with five oracle iterations (Figure 6.6b), accuracy is between 66.44% (20,000 training samples) and 75.17% (40,000 training samples). The precision ranges from 57.35% (20,000 training samples) to 67.69% (40,000 training samples) and the recall from 66.10% (10,000, 20,000, and 125,130 training samples) to 74.58% (40,000 training samples).

The picture is therefore different for the evaluation of the analyses on the validation data D_V. For the analyses without oracle refinement, accuracy, precision, and recall decrease for the evaluation of an analysis trained with 10,000 training samples to an analysis trained with 20,000 samples. While accuracy and precision are highest for the maximum number of 125,130 training samples, the recall peaks at only 10,000 training samples. For the analyses with oracle refinement, accuracy and precision drop by 3.42 and 5.00 percentage points with an increase of the initial amount of training data from 10,000 to 125,130 samples. The recall remains the same. In summary, there is no clear evidence that an increase in training samples from 10,000 up to 125,130 leads to a better generalization of the learned analyses beyond data from D_T.



(a) Line-based analysis based on rules that are synthesized without oracle refinement.



(b) Line-based analysis based on rules that are synthesized and refined over five oracle iterations.

Figure 6.6: Accuracy, precision, and recall for line-based analyses evaluated on the validation set D_V. The analysis rules for the line-based analyses are synthesized from different amounts of training data with and without oracle refinement.

6.5 Discussion

In this section, we discuss the results of the research questions. In RQ 1, we show that only few analysis branches and test samples are approximated. We conclude from these results that we are able to learn an analysis that is not only sound but reasonably precise for the training data. This is an important insight as it confirms that our implementation produces potentially interesting analysis rules that are not just sound due to trivial approximation.

RQ 2 indeed yields promising results for data that is structurally similar to the training data. Without oracle refinement, the analysis rules learned from 100,104 training samples correctly predicts the allocations site labels of 98.02% of the 25,026 test samples. However, we could not confirm the high accuracy in the evaluation of the line-based analysis in RQ 4 on validation data from open source projects. Accuracy (76.67%), precision (68.89%), and recall (68.89%) of the line-based analysis significantly drop for our evaluation on open source code. In addition, edge cases are missed and built-in functions are partly handled incorrectly. Thus, the performance of the line-based analysis trained without oracle for both the open source code and the manually crafted cases is not sufficient for a production-ready static analysis.

If we refine the analyses over five oracle iterations, accuracy, precision, and recall do not improve consistently and significantly both for the evaluation of the analysis on test data in RQ 3 and validation data in RQ 5. The oracle thus misses its goal of improving analysis generalization. With each oracle iteration, new analysis branches are created rather than consolidated. Thus, contrary to the expectations, the specificity of the analysis seems to increase with each oracle iteration. We conclude that the oracle does not improve the generalization of the analysis rules.

The results of RQ 6 indicate that more training data leads to an improvement of analysis generalization for test data from D_T. Again, this statement does not hold for the line-based analysis that is evaluated on the validation data from D_V. For the line-based analysis, more

training data even leads to worse analysis results in some cases which further indicates overfitting to the training data.

In summary, the implementation of the line-based analysis has low accuracy, precision, and recall. In addition, it misses edge cases and partly labels built-in functionality incorrectly. It therefore stands to reason that the analysis overfits to the training data. Thus, our implementation of the line-based analysis does not prove to be applicable in practice.

6.6 Threats to Validity

In the final section of the evaluation, we discuss topics that could potentially threaten the validity of our results. We cover threats that concern the validity of our results and might endanger the generalizability of our findings. We further consider threats that influence the comparability of our results with the results of Bielik et al. [1].

6.6.1 Implementation Errors

Our custom implementations might include bugs which finally influence the results of the evaluation. Both errors in the implementation of the approach and the generation of training data could impact the properties of the learned analysis rules. We rely on automatic tests to make sure that the core functionality of the learning procedure is properly implemented. For instance, we test whether the move and write operations of the language specification L_{alloc} or the modifications performed by the oracle adhere to their expected behavior. In addition, we manually check random samples of the generated training data D_T to assure that the data correctly maps code locations to the proper label to denote allocation sites. Nevertheless, we might have missed bugs in the implementation and cannot manually evaluate all training samples. Thus, an element of risk still remains.

6.6.2 Selection of Validation Data

The two selected projects Axios and Strapi both stem from the same JavaScript open source ecosystem. We thus cannot be sure that the results would not differ for validation data selected from projects with different characteristics such as closed-source projects from a corporate environment. To mitigate this threat, we select popular projects that are actively maintained and contain a reasonable number of contributors. Additionally, we manually craft validation samples which might be biased and thus artificially improve the evaluation results. We therefore report the results for the open source projects and these manually crafted samples separately.

6.6.3 Amount of Validation Data

The dataset D_V used to evaluate the line-based analysis has a relatively small number of samples. From the total number of 371 LOC as samples, 146 include at least one Identifier and are thus considered for the evaluation. We do not assume that the small number of

validation samples affects our conclusion that the analysis rules are not applicable in practice. We expect to see the same classification errors for larger datasets and therefore consider it to be unlikely that more validation data significantly improves the analysis results. On the other hand, validity is still at risk, as we cannot rule out the possibility that the results deteriorate with a larger validation set. However, worse results lead to the same conclusion that the learned analysis rules are not applicable in practice.

6.6.4 Comparability With the Results of Bielik et al.

We do neither have access to the source code nor to the training dataset of Bielik et al. [1]. Thus, the implementation details of the approach of Bielik et al. to learn analysis rules from code and the procedure to generate training data are subject to our interpretation.

We for example use a different algorithm to select guards for the if-then-else statement of the analysis rules. In addition, the implementation of the fundamentals might differ. This concerns, for instance, the implementation of the write and move operations of L_{alloc} as well as the selection of the underlying AST specification.

As for the dataset, we mine our dataset from the ECMAScript Test Suite analogously to Bielik et al. However, we derive training samples from a current state of the test suite using our custom implementation. Therefore, both the source of the training data and the implementation differ and the datasets are thus likely to diverge.

7 Future Work

This chapter discusses potential future improvements and extensions to this work. For improvements, we cover the application of different generalization techniques and the use of more specific training data. The approach could further be expanded to problems other than allocation site analysis and to different programming languages.

7.1 Application of Different Generalization Techniques

As seen in our evaluation, the oracle did not consistently improve the generalization of the analysis. Therefore, different techniques to decrease the complexity of the learned analysis rules could be applied. A generic approach to improve the generalization of decision trees is *pruning* which aims to remove statistically insignificant branches from the tree [36].

A technique tied to the learning approach could be to restrain the domain-specific language L_{Alloc}. For example, the composition of guards could be constrained to a more specific sequence of move and write operations. Such constraints could help to avoid guards that reflect overly specific properties of the training data.

7.2 Use of Specific Training Data

Instead of using training data that tries to cover all notions of allocation sites, more specific sets of data could be used to cover project or case-specific requirements. For project-specific use cases, training data could be generated from the test suite of a project. The rules learned from this training data might better capture specific features of the project's code, such as user-defined functions.

To cover case-specific requirements like edge cases or functions from libraries, a specific dataset that only contains these specific cases could be used to learn rules that individually cover the corresponding requirements. These case-specific rules could then, for example, be used as part of a larger analysis to consider more cases and to thus improve the analysis quality.

7.3 Application to Different Problems

In this thesis, we applied the approach to learn rules for static code analyses to the use case of detecting allocation sites. However, the approach as described in Chapter 4 is not tied to this specific use case. A first step could be the reproduction of the results of Bielik et. al [1] for a

points-to analysis to statically compute a set of objects to which a variable may hold a pointer at runtime. A further potential area of application is the inference of types for dynamically typed languages.

7.4 Application to Different Programming Languages

Besides applying the learning approach to a different problem, it can also be applied to a different programming language. Each language comes with different characteristics and a different ecosystem. These different characteristics yield diverse edge cases and the ecosystems provide functionality specific to the language. Therefore, it might be beneficial to learn rules for an allocation site analysis for other programming languages besides JavaScript.

8 Conclusion

Manually crafting rules for static code analyses that consider both corner cases and the program-specific environment is a hard task. We replicate the approach of Bielik et al. [1] to learn analysis rules from code and provide a more detailed evaluation than Bielik et al. that aims to examine the practical applicability of the learned analysis rules. The approach consists of two components: the synthesis and the oracle. The synthesis uses decision tree learning to derive analysis rules from training data. To reduce the overfitting of the learned analysis rules to the training data, the oracle efficiently creates new training samples that are fed back to the synthesis to refine the rules.

We implemented this approach to learn rules for an allocation site analysis for JavaScript. An allocation site analysis is a static code analysis to detect source code locations that lead to the allocation of new heap memory. To learn rules for the allocation site analysis, we generated training data by instrumenting and executing programs from the ECMAScript Test Suite. Subsequently, we evaluated our implementation and the analysis rules generated with our implementation. Our results show that only few branches need to be approximated due to noisy data and we can thus learn a precise analysis for the training data. We then investigated how well the learned analysis rules with and without oracle refinement are applicable to test data from our generated dataset. The evaluation yielded a high accuracy, precision, and recall for the analysis rules with and without oracle refinement examined on test data. However, the oracle did not lead to a consistent improvement of the analysis quality.

To evaluate the generalization of the analysis rules beyond the generated dataset, we integrated the synthesized analysis rules in a static code analysis that we call line-based analysis. This analysis determines whether a line of code is considered an allocation site. According to our results, the line-based analysis built on top of the learned analysis rules did not perform well for code from open source projects. Furthermore, important edge cases were missed and built-in JavaScript functionality was partly misjudged. Finally, our experiments show that the analysis quality improves with an increasing number of training samples for the evaluation on the generated dataset. However, an increase in training data does not lead to improved evaluation results on data extracted from open source code.

In summary, our implementation yielded analysis rules for the allocation site analysis that are accurate for data that is structurally similar to the training data. Accuracy, precision, and recall dropped significantly for the evaluation data from open source code which leads us to conclude that our generated analysis rules are not applicable to a production-ready static code analysis. We identified a possible cause in the complexity of the analysis rules. This complexity is reflected in the high number of branches of an analysis. Instead of the expected reduction in complexity, the oracle further increased the number of analysis branches. In comparison to the findings of Bielik et al. [1], our analysis rules contain up to almost 65 times

more branches and are thus significantly more complex. While the evaluation of Bielik et al. [1] does not provide metrics comparable to accuracy, precision, or recall for the allocation site analysis, they show that their analysis rules are able to detect edge cases that our rules miss. A next step could therefore aim at improving the generalization of the analysis rules beyond the training data, for example by applying different generalization techniques.

One difficulty we encountered during this thesis is the fine line between generating unbiased training data and indirectly defining analysis rules by assigning predefined labels to selected samples. This was especially evident during the implementation of the modifications for the oracle. Defining a fixed label for a modified sample based on the type of modification might help to increase the generalization of the analysis. However, as this fixed label follows a specific rule, it might be easier to directly implement this rule instead of learning it from code. It is therefore important to asses on a situational basis whether it is the greater effort to generate training data and subsequently learn the analysis rules or to directly craft the rules by hand.

Bibliography

[1] P. Bielik, V. Raychev, and M. Vechev. "Learning a static analyzer from data". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 233–253.

[2] W. S. Humphrey. "A personal commitment to software quality". In: *European Software Engineering Conference*. Springer. 1995, pp. 5–7.

[3] C. Jones. "Geriatric issues of aging software". In: *CrossTalk* 20.12 (2007), pp. 4–8.

[4] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. "What we have learned about fighting defects". In: *Proceedings eighth IEEE symposium on software metrics*. IEEE. 2002, pp. 249–258.

[5] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[6] P. Louridas. "Static code analysis". In: *Ieee Software* 23.4 (2006), pp. 58–61.

[7] A. Møller and M. I. Schwartzbach. *Static program analysis*. 2021.

[8] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. "Memento mori: Dynamic allocation-site-based optimizations". In: *ACM SIGPLAN Notices* 50.11 (2015), pp. 105–117.

[9] S. C. Johnson. "Lint, a C Program Checker". In: *Comp. Sci. Tech. Rep*. 1978, pp. 78–1273.

[10] Y. Smaragdakis and G. Balatsouras. "Pointer analysis". In: *Foundations and Trends in Programming Languages* 2.1 (2015), pp. 1–69.

[11] H. Shahriar and M. Zulkernine. "Classification of static analysis-based buffer overflow detectors". In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*. IEEE. 2010, pp. 94–101.

[12] S. Krüger. "CogniCrypt-the secure integration of cryptographic software." PhD thesis. University of Paderborn, Germany, 2020.

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. "A novel neural source code representation based on abstract syntax tree". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 783–794.

[14] Y. Wang and H. Li. "Code completion by modeling flattened abstract syntax trees as graphs". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 16. 2021, pp. 14015–14023.

[15] A. L. Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[16] A. V. Joshi. "Introduction to AI and ML". In: *Machine Learning and Artificial Intelligence*. Springer, 2020, pp. 3–7.

[17] V. Kotu and B. Deshpande. *Predictive analytics and data mining: concepts and practice with rapidminer*. Morgan Kaufmann, 2014.

[18] S. J. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Third edition, Global edition. Prentice Hall series in artificial intelligence. Boston: Pearson, 2016. ISBN: 1292153970.

[19] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, et al. "Supervised machine learning: A review of classification techniques". In: *Emerging artificial intelligence applications in computer engineering* 160.1 (2007), pp. 3–24.

[20] S. K. Murthy. "Automatic construction of decision trees from data: A multi-disciplinary survey". In: *Data mining and knowledge discovery* 2.4 (1998), pp. 345–389.

[21] J. R. Quinlan. "Induction of decision trees". In: *Machine learning* 1.1 (1986), pp. 81–106.

[22] D. M. Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *arXiv preprint arXiv:2010.16061* (2020).

[23] B. Juba and H. S. Le. "Precision-recall versus accuracy and the role of large data sets". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 4039–4048.

[24] S. Rasthofer, S. Arzt, and E. Bodden. "A machine-learning approach for classifying and categorizing android sources and sinks." In: *NDSS*. Vol. 14. 2014, p. 1125.

[25] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer. "Applying machine learning techniques for detection of malicious code in network traffic". In: *Annual Conference on Artificial Intelligence*. Springer. 2007, pp. 44–50.

[26] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. "Learning a classifier for false positive error reports emitted by static code analysis tools". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2017, pp. 35–42.

[27] E. A. Alikhashashneh, R. R. Raje, and J. H. Hill. "Using machine learning techniques to classify and predict static code analysis tool warnings". In: *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. IEEE. 2018, pp. 1–8.

[28] L. Büch and A. Andrzejak. "Learning-based recursive aggregation of abstract syntax trees for code clone detection". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 95–104.

[29] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter. "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool". In: *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE. 2019, pp. 288–299.

[30] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev. "Scalable taint specification inference with big code". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 760–774.

[31] V. Raychev, M. Vechev, and A. Krause. “Predicting program properties from" big code"”. In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 111–124.

[32] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev. “Inferring crypto API rules from code changes”. In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 450–464.

[33] Y. Wang, W.-d. Cai, and P.-c. Wei. “A deep learning approach for detecting malicious JavaScript code”. In: *Security and Communication Networks* 9.11 (2016), pp. 1520–1534.

[34] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. “JSAI: A static analysis platform for JavaScript”. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 2014, pp. 121–132.

[35] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. “SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript”. In: *International Workshop on Foundations of Object-Oriented Languages (FOOL)*. Vol. 10. Citeseer. 2012.

[36] J. Mingers. “An empirical comparison of pruning methods for decision tree induction”. In: *Machine learning* 4.2 (1989), pp. 227–243.