Learning a Static Analyzer from Data

Pavol Bielik[(✉)], Veselin Raychev, and Martin Vechev

Department of Computer Science, ETH Zürich, Zürich, Switzerland
{pavol.bielik,veselin.raychev,martin.vechev}@inf.ethz.ch

Abstract. To be practically useful, modern static analyzers must precisely model the effect of both, statements in the programming language as well as frameworks used by the program under analysis. While important, manually addressing these challenges is difficult for at least two reasons: (i) the effects on the overall analysis can be non-trivial, and (ii) as the size and complexity of modern libraries increase, so is the number of cases the analysis must handle.

In this paper we present a new, automated approach for creating static analyzers: instead of manually providing the various inference rules of the analyzer, the key idea is to learn these rules from a dataset of programs. Our method consists of two ingredients: (i) a synthesis algorithm capable of learning a candidate analyzer from a given dataset, and (ii) a counter-example guided learning procedure which generates new programs beyond those in the initial dataset, critical for discovering corner cases and ensuring the learned analysis generalizes to unseen programs.

We implemented and instantiated our approach to the task of learning JavaScript static analysis rules for a subset of points-to analysis and for allocation sites analysis. These are challenging yet important problems that have received significant research attention. We show that our approach is effective: our system automatically discovered practical and useful inference rules for many cases that are tricky to manually identify and are missed by state-of-the-art, hand tuned analyzers.

1 Introduction

Static analysis is a fundamental method for automating program reasoning with a myriad of applications in verification, optimization and bug finding. While the theory of static analysis is well understood, building an analyzer for a practical language is a highly non-trivial task, even for experts. This is because one has to address several conflicting goals, including: (i) the analysis must be scalable enough to handle realistic programs, (ii) be precise enough to not report too many false positives, (iii) handle tricky corner cases and specifics of the particular language (e.g., JavaScript), (iv) decide how to precisely model the effect of the environment (e.g., built-in and third party functions), and other concerns. Addressing all of these manually, by-hand, is difficult and can easily result in suboptimal static analyzers, hindering their adoption in practice.

Problem Statement. The goal of this work is to help experts design robust static analyzers, faster, by automatically learning key parts of the analyzer from data.

We state our learning problem as follows: given a domain-specific language $\mathcal{L}$ for describing analysis rules (i.e., transfer functions, abstract transformers), a dataset $\mathcal{D}$ of programs in some programming language (e.g., JavaScript), and an abstraction function α that defines how concrete behaviors are abstracted, the goal is to learn an analyzer $pa \in \mathcal{L}$ (i.e., the analysis rules) such that programs in $\mathcal{D}$ are analyzed as precisely as possible, subject to α.

Key Challenges. There are two main challenges we address in learning static analyzers. First, static analyzers are typically described via rules (i.e., type inference rules, abstract transformers), designed by experts, while existing general machine learning techniques such as support vector machines and neural networks only produce weights over feature functions as output. If these existing techniques were applied to program analysis [25,29], the result would simply be a (linear) combination of existing rules and no new interesting rules would be discovered. Instead, we introduce domain-specific languages for describing the analysis rules, and then learn such analysis rules (which determine the analyzer) over these languages.

The second and more challenging problem we address is how to avoid learning a static analyzer that works well on some training data $\mathcal{D}$, but fails to generalize well to programs outside of $\mathcal{D}$ – a problem known in machine learning as *overfitting*. We show that standard techniques from statistical learning theory [23] such as *regularization* are insufficient for our purposes. The idea of regularization is that picking a simpler model minimizes the expected error rate on unseen data, but a simpler model also contradicts an important desired property of static analyzers to *correctly handle tricky corner cases*. We address this challenge via a counter-example guided learning procedure that leverages program semantics to generate new data (i.e., programs) for which the learned analysis produces wrong results and which are then used to further refine it. To the best of our knowledge, we are the first to replace model regularization with a counter-example guided procedure in a machine learning setting with large and noisy training datasets.

We implemented our method and instantiated it for the task of learning production rules of realistic analyses for JavaScript. We show that the learned rules for points-to and for allocation site analysis are indeed interesting and are missed by existing state-of-the-art, hand crafted analyzers (e.g., Facebook's Flow [5]) and TAJS (e.g., [17]).

Our main contributions are:

– A method for learning static analysis rules from a dataset of programs. To ensure that the analysis *generalizes* beyond the training data we carefully generate counter-examples to the currently learned analyzer using an oracle.
– A decision-tree-based algorithm for learning analysis rules from data that *learns to overapproximate* when the dataset cannot be handled precisely.

– An end-to-end implementation of our approach and an evaluation on the challenging problem of learning tricky JavaScript analysis rules. We show that our method produces interesting analyzers which generalize well to new data (i.e. are sound and precise) and handle many tricky corner cases.

2 Our Approach

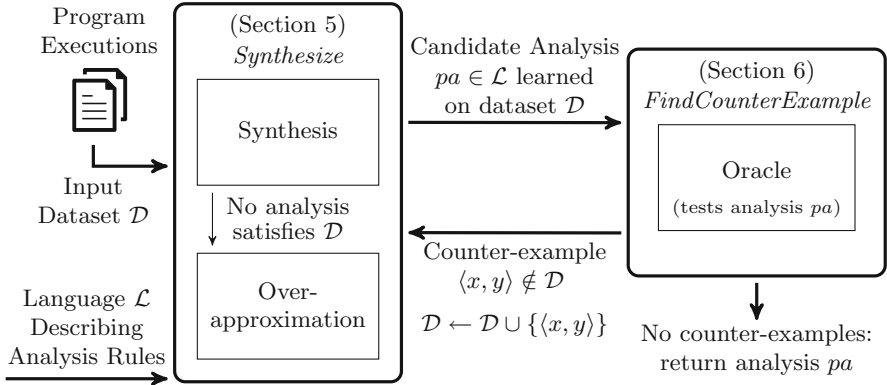We begin by describing components of our learning approach as shown in Fig. 1.

Fig. 1. Overview of our approach to learning static analysis rules from data consisting of three components – a language $\mathcal{L}$ for describing the rules, a learning algorithm and an oracle – that interact in a counter-example based refinement loop.

Obtaining Training Data $\mathcal{D}$. Our learning approach uses dataset of examples $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^{N}$ consisting of pairs $\langle x^j, y^j \rangle$ where x^j is a program and y^j is the desired output of the analysis when applied to x^j. In general, obtaining such labeled training data for machine learning purposes is a tedious task. In our setting, however, this process can be automated because: (i) in static analysis, there is a well understood notion of soundness, namely, the analyzer must *approximate* (in the sense of lattice ordering) the concrete program behaviors, and (ii) thus, we can simply run a large amount of programs in a given programming language with some inputs, and obtain a subset of the concrete semantics for each program. We note that our learning method is independent of how the labels are obtained. For example, the labels y^j can be obtained by running static or dynamic analyzers on the programs x^j in $\mathcal{D}$ or they can be provided manually.

Synthesizer and Language $\mathcal{L}$. To express interesting rules of a static analyzer, we use a loop-free domain-specific language $\mathcal{L}$ with branches. The synthesizer then takes as input the dataset $\mathcal{D}$ with a language $\mathcal{L}$ and produces a candidate program analysis $pa \in \mathcal{L}$ which correctly handles the pairs in $\mathcal{D}$. The synthesizer we propose phrases the problem of learning a static analysis over $\mathcal{L}$ as a problem in learning decision trees over $\mathcal{L}$. These components are described in Sect. 5.

Oracle. Our goal is to discover a program analysis that not only behaves as described by the pairs in the dataset $\mathcal{D}$, but one that generalizes to programs beyond those in $\mathcal{D}$. To address this challenge, we introduce the oracle component (*FindCounterExample*) and connect it with the synthesizer. This component takes as input the learned analysis *pa* and tries to find another program x for which *pa* fails to produce the desired result y. This counter-example $\langle x, y \rangle$ is then fed back to the synthesizer which uses it to generate a new candidate analyzer as illustrated in Fig. 1. To produce a counter-example, the oracle must have a way to quickly and effectively test a (candidate) static analyzer. In Sect. 6, we present two techniques that make the testing process more effective by leveraging the current set $\mathcal{D}$ as well as current candidate analysis *pa* (these techniques for testing a static analyzer are of interest beyond learning considered in our work).

Counter-Example Guided Learning. To learn a static analyzer *pa*, the synthesizer and the oracle are linked together in a counter-example guided loop. This type of iterative search is frequently used in program synthesis [31], though its instantiation heavily depends on the particular application task at hand. In our setting, the examples in $\mathcal{D}$ are programs (and not say program states) and we also deal with notions of (analysis) approximation. This also means that we cannot directly leverage off-the-shelf components (e.g., SMT solvers) or existing synthesis approaches. Importantly, the counter-example guided approach employed here is of interest to machine learning as it addresses the problem of overfitting with techniques beyond those typically used (e.g., regularization [23], which is insufficient here as it does not consider samples not in the training dataset).

Practical Applicability. We implemented our approach and instantiated it to the task of learning rules for points-to and allocation site analysis for JavaScript code. This is a practical and relevant problem because of the tricky language semantics and wide use of libraries. Interestingly, our system learned inference rules missed by manually crafted state-of-the-art tools, e.g., Facebook's Flow [5].

3   Overview

This section provides an intuitive explanation of our approach on a simple points-to analysis for JavaScript. Assume we are learning the analysis from one training data sample given in Fig. 2 (a). It consists of variables `a`, `b` and `b` is assigned an object s_0. Our goal is to learn that `a` may also point to that same object s_0.

Points-to analysis is typically done by applying inference rules until fixpoint. An example of an inference rule modeling the effect of assignment is:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad \texttt{Assignment}(v_1, v_2)}{\texttt{VarPointsTo}(v_1, h)} \ [\textsc{Assign}]$$

This rule essentially says that if variable v_2 is assigned to v_1 and v_2 may point to an object h, then the variable v_1 may also point to this object h.

```
var b = {}; // empty object s₀
a = b;
```

Expected points-to set
$\mathcal{D} = \{(\mathtt{a} \to \{s_0\})\}$

```
├ VarDeclaration:b
  ├ ObjectExpression:{}
├ Assignment
  ├ Identifier:a
  ├ Identifier:b
```

(a) Training data   (b) Abstract syntax tree (AST) representation of (a)

$f_{desired}(x) ::=$

y **if there is** $\mathtt{Assignment}(x, y)$

y **if there is** $\mathtt{VarDeclaration{:}x}(y)$

$\bot$ **otherwise**

$f_{overfit}(x) ::=$

y **if** y **is** $\mathtt{VarDeclaration{:}y}$ **preceding** x

y **if there is** $\mathtt{VarDeclaration{:}x}(y)$

$\bot$ **otherwise**

(c) Learned functions to resolve points-to queries from (a)
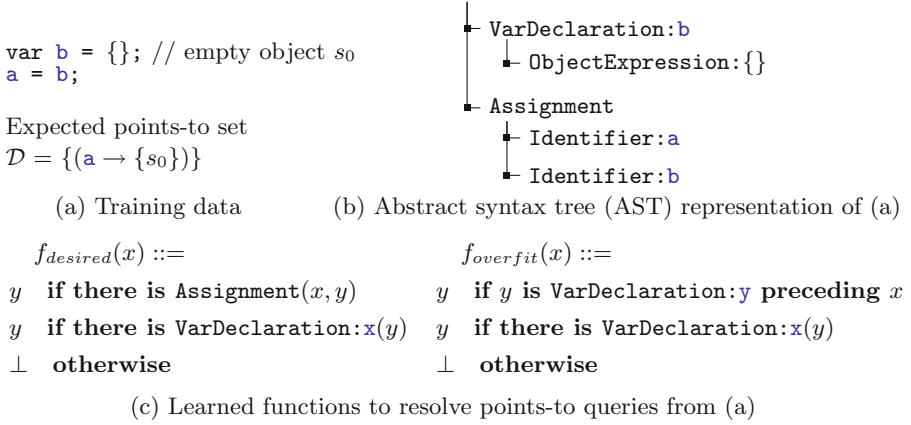
Fig. 2. Example data for learning points-to analysis.

Domain specific language (DSL) for analysis rules. Consider the following general shape of inference rules:

$$\frac{\mathtt{VarPointsTo}(v_2, h) \qquad v_2 = f(v_1)}{\mathtt{VarPointsTo}(v_1, h)} \ [\text{General}]$$

Here, the function f takes a program element (a variable) and returns another program element or $\bot$. The rule says: use the function f to find a variable v_2 whose points-to set will be used to determine what v_1 points to. The Assign rule is an instance of the General rule that can be implemented by traversing the AST and checking if the parent node of x is of type $\mathtt{Assignment}$ and if x is its first child. In this case, the right sibling of x is returned. Otherwise f returns $\bot$.

Problem Statement. The problem of learning a points-to analysis can now be stated as follows: find an analysis $pa \in \mathcal{L}$ such that when analyzing the programs in the training data $\mathcal{D}$, the resulting points-to set is as outlined in $\mathcal{D}$.

The Overfitting Problem. Consider Fig. 2(b) which shows the AST of our example. In addition to Assign, we need to handle the case of variable initialization (first line in the program). Note that the dataset $\mathcal{D}$ does not uniquely determine the best function f. In fact, instead of the desired one $f_{desired}$, other functions can be returned such as $f_{overfit}$ shown in Fig. 2(c). This function inspects the statement prior to an assignment instead of at the assignment itself and yet it succeeds to produce the correct analysis result on our dataset $\mathcal{D}$. However, this is due to the specific syntactic arrangement of statements in the training data $\mathcal{D}$ and may not generalize to other programs, beyond those in $\mathcal{D}$.

Our Solution. To address the problem of overfitting to $\mathcal{D}$, we propose a counter-example guided procedure that biases the learning towards semantically meaningful analyses. That is, the oracle tests the current analyzer and tries to find

a counter-example on which the analysis fails. Our strategy to generating candidate programs is to modify the programs in $\mathcal{D}$ in ways that can change both the syntax and the semantics of those programs. As a result, any analysis that depends on such properties would be penalized in the next iteration of *Synthesize*. As we show in the evaluation, our approach results in a much faster oracle than if we had generated programs blindly. This is critical as faster ways of finding counter-examples increase the size of the search space we can explore, enabling us to discover interesting analyzers in reasonable time.

For example, a possible way to exclude $f_{overfit}$ is to insert an unnecessary statement (e.g., `var c = 1`) before the assignment `a = b` in Fig. 2(a). Here, the analysis defined by $f_{overfit}$ produces an incorrect points-to set for variable `a` (as it points-to the value 1 of variable `c`). Once this sample is added to $\mathcal{D}$, $f_{overfit}$ is penalized as it produces incorrect results and the next iteration will produce a different analysis until eventually the desired analysis $f_{desired}$ is returned.

Soundness of the Approach. Our method produces an analyzer that is guaranteed to be sound w.r.t to all of the examples in $\mathcal{D}$. Even if the analyzer cannot exactly satisfy all examples in $\mathcal{D}$, the synthesis procedure always returns an *over-approximation* of the desired outputs. That is, when it cannot match the target output exactly, *Synthesize* learns to approximate (e.g., can return $\top$ in some cases). A formal argument together with a discussion on these points is provided in Sect. 5. However, our method is not guaranteed to be sound for all programs in the programming language. We see the problem of certifying the analyzer as orthogonal and complementary to our work: our method can be used to predict an analyzer which is likely correct, generalize well, and to sift through millions of possibilities quickly, while a follow-up effort can examine this analyzer and decide whether to accept it or even fully verify it. Here, an advantage of our method is that the learned analyzer is expressed as a program, which can be easily examined by an expert, as opposed to standard machine learning models where interpreting the result is nearly impossible and therefore difficult to verify with standard methods.

4   Checking Analyzer Soundness

In this section, following [4], we briefly discuss what it means for a (learned) analyzer to be sound. The concrete semantics of a program p include all of p's concrete behaviors and are captured by a function $[\![p]\!] : \mathbb{N} \to \wp(\mathcal{C})$. This function associates a set of possible concrete states in $\mathcal{C}$ with each position in the program p, where a position can be a program counter or a node in the program's AST.

A static analysis pa of a program p computes an abstract representation of the program's concrete behaviors, captured by a function $pa(p) : \mathbb{N} \to \mathcal{A}$ where $(\mathcal{A}, \sqsubseteq)$ is typically an abstract domain, usually a lattice of abstract facts equipped with an ordering $\sqsubseteq$ between facts. An abstraction function $\alpha : \wp(\mathcal{C}) \to \mathcal{A}$ then establishes a connection between the concrete behaviors and the abstract facts. It defines how a set of concrete states in $\mathcal{C}$ is abstracted into an abstract element

in $\mathcal{A}$. The function is naturally lifted to work point-wise on a set of positions in $\mathbb{N}$ (used in the definition below).

Definition 1 (Analysis Soundness). *A static analysis pa is sound if:*

$$\forall p \in \mathcal{T_L}.\ \alpha(\llbracket p \rrbracket) \sqsubseteq pa(p) \tag{1}$$

Here $\mathcal{T_L}$ denotes the set of all possible programs in the target programming language ($\mathcal{T_L}$). That is, a static analysis is sound if it over-approximates the concrete behaviors of the program according to the particular lattice ordering.

4.1 Checking Soundness

One approach for checking the soundness of an analyzer is to try and automatically verify the analyzer itself, that is, to prove the analyzer satisfies Definition 1 via sophisticated reasoning (e.g., as the one found in [10]). Unfortunately, such automated verifiers do not currently exist (though, coming up with one is an interesting research challenge) and even if they did exist, it is prohibitively expensive to place such a verifier in the middle of a counter-example learning loop where one has to discard thousands of candidate analyzers quickly. Thus, the soundness definition that we use in our approach is as follows:

Definition 2 (Analysis Soundness on a Dataset and Test Inputs). *A static analysis pa is sound w.r.t a dataset of programs P and test inputs ti if:*

$$\forall p \in P.\ \alpha(\llbracket p \rrbracket_{ti}) \sqsubseteq pa(p) \tag{2}$$

The restrictions over Definition 1 are: the use of a set $P \subseteq \mathcal{T_L}$ instead of $\mathcal{T_L}$ and $\llbracket p \rrbracket_{ti}$ instead of $\llbracket p \rrbracket$. Here, $\llbracket p \rrbracket_{ti} \subseteq \llbracket p \rrbracket$ denotes a subset of a program p's behaviors obtained after running the program on some set of test inputs *ti*.

The advantage of this definition is that we can automate its checking. We run the program p on its test inputs *ti* to obtain $\llbracket p \rrbracket_{ti}$ (a finite set of executions) and then apply the function α on the resulting set. To obtain $pa(p)$, we run the analyzer *pa* on p; finally, we compare the two results via the inclusion operator $\sqsubseteq$.

5 Learning Analysis Rules

We now present our approach for learning static analysis rules from examples.

5.1 Preliminaries

Let $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^N$ be a dataset of programs from a target language $\mathcal{T_L}$ together with outputs that a program analysis should satisfy. That is, $x^j \in \mathcal{T_L}$ and y^j are the outputs to be satisfied by the learned program analysis.

Definition 3 (Analysis Soundness on Examples). *We say that a static analysis* $pa \in \mathcal{L}$ *is sound on* $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^N$ *if:*

$$\forall j \in 1 \dots N \ . \ \ y^j \sqsubseteq pa(x^j) \tag{3}$$

This definition is based on Definition 2, except that the result of the analysis is provided in $\mathcal{D}$ and need not be computed by running programs on test inputs.

Note that the definition above does not mention the precision of the analysis pa but is only concerned with soundness. To search for an analysis that is both sound, precise and avoids obvious, but useless solutions (e.g., always returns $\top$ element of the lattice $(\mathcal{A}, \sqsubseteq)$), we define a precision metric.

Precision Metric. First, we define a function $r : \mathcal{T}_\mathcal{L} \times \mathcal{A} \times \mathcal{L} \rightarrow \mathbb{R}$ that takes a program in the target language, its desired program analysis output and a program analysis and indicates if the result of the analysis is exactly as desired:

$$r(x, y, pa) \ \ = \ \ \textbf{if } (y \neq pa(x)) \textbf{ then } 1 \textbf{ else } 0 \tag{4}$$

We define a function *cost* to compute precision on the full dataset $\mathcal{D}$ as follows:

$$cost(\mathcal{D}, pa) = \sum_{\langle x,y \rangle \in \mathcal{D}} r(x, y, pa) \tag{5}$$

Using the precision metric in Eq. 5, we can state the following lemma:

Lemma 1. *For a program analysis* $pa \in \mathcal{L}$ *and a dataset* $\mathcal{D}$, *if* $cost(\mathcal{D}, pa) = 0$, *then the analysis is sound according to Definition 3.*

Proof: The proof is direct. Because $cost(\mathcal{D}, pa) = 0$ and r is positive, then for every $\langle x, y \rangle \in \mathcal{D}$, $r(x, y, pa) = 0$. This means that $y = pa(x)$ and so $y \sqsubseteq pa(x)$, which is as defined in Definition 3.   □

5.2   Problem Formulation

Given a language $\mathcal{L}$ that describes analysis inference rules (i.e., abstract transformers) and a dataset $\mathcal{D}$ of programs with the desired analysis results, the *Synthesize* procedure should return a program analysis $pa \in \mathcal{L}$ such that:

1. pa is sound on the examples in $\mathcal{D}$ (Definition 3), and
2. $cost(\mathcal{D}, pa)$ is minimized.

The above statement essentially says that we would like to obtain a sound analysis which also minimizes the over-approximation that it makes. As the space of possible analyzers can be prohibitively large, we discuss a restriction on the language $\mathcal{L}$ and give a procedure that efficiently searches for an analyzer such that soundness is enforced and *cost* is (approximately) minimized.

5.3   Language Template for Describing Analysis Rules

A template of the language $\mathcal{L}$ for describing analysis rules is shown in Fig. 3(a). The template is simple and contains actions and guards that are to be instantiated later. The statements in the language are either an action or a conditional `if-then-else` statements that can be applied recursively.
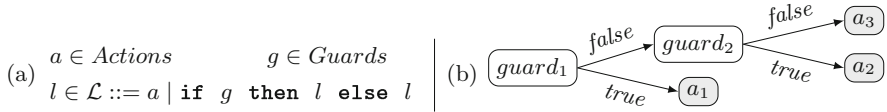
(a)
$$a \in Actions \qquad g \in Guards$$
$$l \in \mathcal{L} ::= a \mid \texttt{if } g \texttt{ then } l \texttt{ else } l$$

(b)

Fig. 3. (a) Syntax of a template language $\mathcal{L}$ with branches for expressing analysis rules. (b) Example of a function from the $\mathcal{L}$ language shown as a decision tree.

An analysis rule of a static analyzer is expressed as a function built from statements in $\mathcal{L}$. As usual, the function is executed until a fixed point [4]. The semantics of the `if` statements in *pa* is standard: guards are predicates (side-effect free) that inspect the program being analyzed and depending on their truth value, the corresponding branch of the `if` statement is taken. The reason such `if` statements are interesting is because they can express analysis rules such as the ones of our running example in Fig. 2.

We provide a formal semantics and detailed description of how the language $\mathcal{L}$ is instantiated for learning points-to and allocation site analysis in an extended version of this paper [1].

5.4   ID3 Learning for a Program Analyzer

A key challenge in learning program analyzers is that the search space of possible programs over $\mathcal{L}$ is massive as the number of possible combinations of branches and subprograms is too large. However, we note that elements of $\mathcal{L}$ can be represented as trees where internal nodes are guards of `if` statements and the leafs are actions as shown in Fig. 3(b). Using this observation we can phrase the problem of learning an analyzer in $\mathcal{L}$ as the problem of learning a decision tree, allowing us to adapt existing decision tree algorithms to our setting.

Towards that, we extend the ID3 [27] algorithm to handle action programs in the leafs and to enforce soundness of the resulting analysis $pa \in \mathcal{L}$. Similarly to ID3, our algorithm is a greedy procedure that builds the decision tree in a top-down fashion and locally maximizes a metric called information gain.

Our learning shown in Algorithm 1 uses three helper functions that we define next. First, the *genAction* function returns best analysis a_{best} for a dataset $\mathcal{D}$:

$$a_{best} = genAction(\mathcal{D}) = \underset{a \in Actions}{\arg\min}\; cost(\mathcal{D}, a) \tag{6}$$

That is, *genAction* returns the most precise program analysis consisting only of *Actions* (as we will see later, an action is just a sequence of statements, without branches). If a_{best} is such that $cost(\mathcal{D}, a_{best}) = 0$, the analysis is both precise and sound (from Lemma 1), which satisfies our requirements stated in Sect. 5.2 and we simply return it. Otherwise, we continue by generating an `if` statement.

Generating Branches. The ID3 decision tree learning algorithm generates branches based on an information gain metric. To define this metric, we first use a standard definition of entropy. Let the vector $\boldsymbol{w} = \langle w_1, ..., w_k \rangle$ consist of elements from a set C. Then the entropy H on $\boldsymbol{w}$ is:

$$H(\boldsymbol{w}) = -\sum_{c \in C} \frac{count(c, \boldsymbol{w})}{k} \log_2 \left(\frac{count(c, \boldsymbol{w})}{k} \right) \tag{7}$$

where $count(c, \boldsymbol{w}) = | \{ i \in 1 \ldots k \mid w_i = c \} |$.

For a dataset $d \subseteq \mathcal{D}$, let $d = \{ x_i, y_i \}_{i=1}^{|d|}$. Then, we define the following vector:

$$\boldsymbol{w}_d^{a_{best}} = \langle r(x_i, y_i, a_{best}) \mid i \in 1 \ldots |d| \rangle \tag{8}$$

That is, for every program in d, we record if a_{best} is a precise analysis (via the function r defined previously). Let $g \in Guards$ be a predicate that is to be evaluated on a program x. Let $\mathcal{D}^g = \{ \langle x, y \rangle \in \mathcal{D} \mid g(x) \}$ and $\mathcal{D}^{\neg g} = \mathcal{D} \setminus \mathcal{D}^g$.

The information gain on a set of examples $\mathcal{D}$ for analysis a_{best} and predicate guard g is then defined as:

$$IG^{a_{best}}(\mathcal{D}, g) = H(\boldsymbol{w}_\mathcal{D}^{a_{best}}) - \frac{|\mathcal{D}^g|}{|\mathcal{D}|} H(\boldsymbol{w}_{\mathcal{D}^g}^{a_{best}}) - \frac{|\mathcal{D}^{\neg g}|}{|\mathcal{D}|} H(\boldsymbol{w}_{\mathcal{D}^{\neg g}}^{a_{best}}) \tag{9}$$

For a given predicate g, what the information gain quantifies is how many bits of information about the analysis precision will be saved if instead of using the imprecise analysis a_{best} directly, we split the dataset with a predicate g. Using the information gain metric we define *genBranch* as follows:

$$g_{best} = genBranch(a_{best}, \mathcal{D}) = \arg\max_{g \in Guards}{}^\perp IG^{a_{best}}(\mathcal{D}, g) \tag{10}$$

Here, $\arg\max^\perp$ is defined to return $\perp$ if the maximized information gain is 0, or otherwise to return the guard g which maximizes the information gain.

Back to Algorithm 1, if *genBranch* returns a predicate with positive information gain, we split the dataset with this predicate and call *Synthesize* recursively on the two parts. In the end, we return an `if` statement on the predicate g and the two recursively synthesized analysis pieces.

Approximation. If the information gain is 0 (i.e. $g_{best} = \perp$), we could not find any suitable predicate to split the dataset and the analysis a_{best} has non-zero cost. In this case, we define a function *approximate* that returns an approximate, but sound program analysis – in our implementation we return analysis that loses precision by simply returning $\top$, which is always a sound analysis.

def *Synthesize*($\mathcal{D}$)

> **Input:** Dataset $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^{N}$
> **Output:** Program $pa \in \mathcal{L}$
> $a_{best} \leftarrow genAction(\mathcal{D})$
> **if** $cost(\mathcal{D}, a_{best}) = 0$ **then return** a_{best};
> $g_{best} \leftarrow genBranch(a_{best}, \mathcal{D})$
> **if** $g_{best} = \bot$ **then return** $approximate(\mathcal{D})$;
> $p_1 \leftarrow Synthesize(\{\langle x, y \rangle \in \mathcal{D} \mid g_{best}(x)\})$
> $p_2 \leftarrow Synthesize(\{\langle x, y \rangle \in \mathcal{D} \mid \neg g_{best}(x)\})$
> **return** `if` g_{best} `then` p_1 `else` p_2

Algorithm 1: Learning algorithm for programs from language $\mathcal{L}$.

Note that this approximation does not return $\top$ for the entire analysis but only for few of the branches in the decision tree for which the synthesis procedure fails to produce a good program using both *genAction* and *getBranch*.

In terms of guarantees, for Algorithm 1, we can state the following lemma.

Lemma 2. *The analysis pa $\in \mathcal{L}$ returned by Synthesize is sound according to Definition 3.*

The proof of this lemma simply follows the definition of the algorithm and uses induction for the recursion. For our induction base, we have already shown that in case $cost(\mathcal{D}, a_{best}) = 0$, the analysis is sound. By construction, the analysis is also sound if *approximate* is called. In our induction step we use the fact that analyses p_1 and p_2 from the recursion are sound and must only show that the composed analysis `if` g_{best} `then` p_1 `else` p_2 is also sound.

6   The Oracle: Testing an Analyzer

A key component of our approach is an oracle that can quickly test whether the current candidate analyzer is sound, and if not, to find a counter-example. The oracle takes as an input a candidate analyzer *pa* and the current dataset $\mathcal{D}$ used to learn *pa* and outputs a counter-example program on which *pa* is unsound. More formally, if $P_{\mathcal{D}} = \{x \mid \langle x, y \rangle \in \mathcal{D}\}$, our goal is to find a counter-example program $p \in \mathcal{T}_{\mathcal{L}}$ such that $p \notin P_{\mathcal{D}}$ and the soundness condition in Definition 2 is violated for the given analysis *pa* and program *p*. That is, our oracle must generate new programs beyond those already present in $P_{\mathcal{D}}$.

Key Challenge. A key problem the oracle must address is to *quickly* find a counter-example in the search space of all possible programs. As we show in Sect. 7, finding such a counter-example by blindly generating new programs does not work as the search space of programs in $\mathcal{T}_{\mathcal{L}}$ is massive (or even infinite).

Speeding up the Search. We address this challenge by designing a general purpose oracle that *prioritizes* the search in $\mathcal{T}_{\mathcal{L}}$ based on ideas inspired by state-of-the-art testing techniques [11,22]. In particular, we generate new programs by

performing modifications of the programs in $P_{\mathcal{D}}$. These modifications are carefully selected by exploiting the structure of the current analysis pa in two ways: (i) to select a program in $\mathcal{T}_{\mathcal{L}}$ and the position in that program to modify, and (ii) to determine what modification to perform at this position.

6.1 Choosing Modification Positions

Given a program $x \in P_{\mathcal{D}}$ and analysis pa, we prioritize positions that are *read* while executing the program analysis pa and changing them would trigger different *execution path* in the analyzer pa itself (not the analyzed program). Determining these positions is done by instrumenting the program analyzer and recording the relevant instructions affecting the branches the analyzer takes.

For example, for Fig. 2(a), we defined the analysis by the function $f_{overfit}$. For this function, only a subset of all AST nodes determine which of the three cases in the definition of $f_{overfit}$ will be used to compute the result of the analysis. Thus, we choose the modification position to be one of these AST nodes.

6.2 Defining Relevant Program Modifications

We now define two approaches for generating interesting program modifications that are potential counter-examples for the learned program analysis pa.

Modification via Equivalence Modulo (EMA) Abstraction. The goal of EMA technique is to ensure that the candidate analysis pa is robust to certain types of program transformations. To achieve this, we transform the statement at the selected program position in a semantically-preserving way, producing a set of new programs. That is, while the transformation is semantic-preserving, it is also one that should not affect the result of the analysis pa.

More formally, an EMA transformation is a function $F_{ema} : \mathcal{T}_{\mathcal{L}} \times \mathbb{N} \to \wp(\mathcal{T}_{\mathcal{L}})$ which takes as input a program p and a position in the program, and produces a set of programs that are a transformation of p at position n. If the analysis pa is sound, then these functions (transformations) have the following property:

$$\forall p' \in F_{ema}(p,n).pa(p) = pa(p') \tag{11}$$

The intuition behind such transformations is to ensure stability by exploring *local program modifications*. If the oracle detects the above property is violated, the current analysis pa is unsound and the counter-example program p' is reported. Examples of applicable transformations are dead code insertion, variable names renaming or constant modification, although transformations to use can vary depending on the kind of analysis being learned. For instance, inserting dead code that reuses existing program identifiers can affect flow-insensitive analysis, but should not affect a flow-sensitive analysis. The EMA property is similar to notion of algorithmic stability used in machine learning where the output of a classifier should be stable under small perturbations of the input as well as the concept of equivalence modulo inputs used to validate compilers [22].

Modification via Global Jumps. The previous modifications always generated semantic-preserving transformations. However, to ensure better generalization we are also interested in exploring changes to programs in $P_{\mathcal{D}}$ that may not be semantic preserving, defined via a function $F_{gj} : \mathcal{T}_{\mathcal{L}} \times \mathbb{N} \to \wp(\mathcal{T}_{\mathcal{L}})$. The goal is to discover a new program which exhibits behaviors not seen by any of the programs in $P_{\mathcal{D}}$ and is not considered by the currently learned analyzer pa.

Overall, as shown in Sect. 7, our approach for generating programs to test the analysis pa via the functions F_{gj} and F_{ema} is an order of magnitude more efficient at finding counter-examples than naively modifying the programs in $P_{\mathcal{D}}$.

7   Implementation and Evaluation

In this section we provide an implementation of our approach shown in Fig. 1 as well as a detailed experimental evaluation instantiated to two challenging analysis problems for JavaScript: learning points-to analysis rules and learning allocation site rules. In our experiments, we show that:

- The approach can learn practical program analysis rules for tricky cases involving JavaScript's built-in objects. These rules can be incorporated into existing analyzers that currently handle such cases only partially.
- The counter-example based learning is critical for ensuring that the learned analysis generalizes well and does not overfit to the training dataset.
- Our oracle can effectively find counter-examples (orders of magnitude faster than random search).

These experiments were performed on a 28 core machine with 2.60 Ghz Intel(R) Xeon(R) CPU E5-2690 v4 CPU, running Ubuntu 16.04. In our implementation we parallelized both the learning and the search for the counter-examples.

Training Dataset. We use the official ECMAScript (ECMA-262) conformance suite (https://github.com/tc39/test262) – the largest and most comprehensive test suite available for JavaScript containing over 20 000 test cases. As the suite also includes the latest version of the standard, all existing implementations typically support only a subset of the testcases. In particular, the `NodeJS` interpreter v4.2.6 used in our evaluation can execute (i.e., does not throw a syntax error) 15 675 tests which we use as the training dataset for learning.

Program Modifications. We list the program modifications used to instantiate the oracle in Table 1. The semantic preserving program modifications that should not change the result of analyses considered in our work F_{ema} are: inserted dead code, renamed variables and user functions, renamed parameters as well as generated side-effect free expressions (e.g., declaring new variables). Note that these mutations are very general and should apply to almost arbitrary property. To explore new program behaviours by potentially changing program semantics we use program modifications F_{gj} that change values of constants (strings and numbers), add method arguments and parameters.

Table 1. Program modifications used to instantiate the oracle (Sect. 6) that generates counter-examples for points-to analysis and allocation site analysis.

Program modifications	
F_{ema}	F_{gj}
Adding dead code	Adding method arguments
Renaming variables	Adding method parameters
Renaming user functions	Changing constants
Side-effect free expressions	

7.1   Learning Points-to Analysis Rules for JavaScript

We now evaluate the effectiveness of our approach on the task of learning a points-to analysis for the JavaScript built-in APIs that affect the binding of `this` object. This is useful because existing analyzers currently either model `this` only partially [5,12] (i.e., cover only a subset of `Function.prototype` API behaviors) or not at all [16,24], resulting in potentially unsound results.

   We illustrate some of the complexity for determining the objects to which `this` points-to within the same method in Fig. 4. Here, `this` points-to different objects depending on how the method is invoked and what values are passed in as arguments. In addition to the values shown in the example, other values may be seen during runtime if other APIs are invoked, or the method `isBig` is used as an object method or as a global method.

```
global.length = 4;            // this points to global
var dat = [5, 3, 9, 1];       dat.filter(isBig); // [5, 9]
function isBig(value) {       // this points to boxed 42
  return value >=             dat.filter(isBig, 42); // []
      this.length;            // this points to dat object
}                             dat.filter(isBig, dat); // [5, 9]
```

Fig. 4. JavaScript code snippet illustrating subset of different objects to which `this` can point to depending on the context method `isBig` is invoked in.

Language $\mathcal{L}$. To learn points-to analysis, we use a domain-specific language $\mathcal{L}_{pt}$ with if statements (to synthesize branches for corner cases) and instructions to traverse the JavaScript AST in order to provide the specific analysis of each case. A detailed list of the instructions with their semantics is provided in [1].

Learned Analyzer. A summary of our learned analyzer is shown in Table 2. For each API we collected all its usages in the ECMA-262 conformance suite, ranging from only 6 to more than 600, and used them as initial training dataset for the learning. In all cases, a significant amount of counter-examples were needed to refine the analysis and prevent overfitting to the initial dataset. On average,

Table 2. Dataset size, number of counter-examples found and the size of the learned points-to analysis for JavaScript APIs that affect the points-to set of `this`.

Function name	Dataset size	Counter-examples found	Analysis size*
Function.prototype			
call()	26	372	97 (18)
apply()	6	182	54 (10)
Array.prototype			
map()	315	64	36 (6)
some()	229	82	36 (6)
forEach()	604	177	35 (5)
every()	338	31	36 (6)
filter()	408	76	38 (6)
find()	53	73	36 (6)
findIndex()	51	96	28 (6)
Array			
from()	32	160	57 (7)
JSON			
stringify()	18	55	9 (2)

* Number of instructions in $\mathcal{L}_{pt}$ (Number of `if` branches)

for each API, the learning finished in 14 min, out of which 4 min were used to synthesize the program analysis and 10 min used in the search for counter-examples (cumulatively across all refinement iterations). The longest learning time was 57 min for the `Function.prototype.call` API for which we also learn the most complex analysis – containing 97 instructions in $\mathcal{L}_{pt}$. We note that even though the APIs in `Array.prototype` have very similar semantics, the learned programs vary slightly. This is caused by the fact that different number and types of examples were available as the initial training dataset which also means that the oracle had to find different types of counter-examples. We provide an example of the learned analysis in [1].

7.2 Learning Allocation Site Analysis for JavaScript

We also evaluate the effectiveness of our approach on a second analysis task – learning allocation sites in JavaScript. This is an analysis that is used internally by many existing analyzers. The analysis computes which statements or expressions in a given language result in an allocation of a new heap object.

We illustrate the expected output and some of the complexities of allocation site analysis on a example shown in Fig. 5. In JavaScript, there are various ways for how an object can be allocated including creating new object without calling a constructor explicitly (for example by creating new array or object expression inline), creating new object by calling a constructor explicitly using `new`,

```
var obj = {a: 7};
var arr = [1, 2, 3, 4];
if (obj.a == arr.slice(0,2)) { ... }          Allocation Sites
var n = new Number(7);                       (new object allocated)
var obj2 = new Object(obj);
try {  ...  } catch (err) {  ...  }
```

Fig. 5. Illustration of program locations (underlined) for which the allocation site analysis should report that a new object is allocated.

creating a new object by calling a method or new objects created by throwing an exception. In addition, some of the cases might further depend on actual values passed as arguments. For example, calling a `new Object(obj)` constructor with `obj` as an argument does not create a new object but returns the `obj` passed as argument instead. The goal of the analysis is to determine all such program locations (as shown in Fig. 5) at which a new object is allocated.

Consider the following simple, but unsound and imprecise allocation site analysis:

$$f_{alloc}(x) = \begin{cases} true & \textbf{if there is } \texttt{Argument:x} \textbf{ or } \texttt{NewExpression:x} \\ false & \textbf{otherwise} \end{cases}$$

which states that a location x is an allocation site if it is either an argument or a new expression. This analysis is imprecise because there are other ways to allocate an object (e.g., when creating arrays, strings, boxed values or by calling a function). It is also unsound, because the JavaScript compiler might not create a new object even when `NewExpression` is called (e.g., `new Object(obj)` returns the same object as the given *obj*).

Instead of defining tricky corner cases by hand, we use our approach to learn this analyzer automatically from data. We instantiate the approach in a very similar way compared to learning points-to analysis by adjusting the language and how the labels in the training dataset are obtained (details provided in [1]). For this task, we obtain 134 721 input/output examples from the training data, which are further expanded with additional 905 counter-examples found during 99 refinement iterations of the learning algorithm. For this (much higher than in the other analyzer) number of examples the synthesis time was 184 min while the total time required to find counter-examples was 7 h.

The learned program is relatively complex and contains 135 learned branches, including the tricky case where `NewExpression` does not allocate a new object. Compared to the trivial, but wrong analysis f_{alloc}, the synthesized analysis marks over twice as many locations in the code as allocation sites (≈ 21K vs ≈ 45K).

7.3   Analysis Generalization

We study how well the learned analyzer for points-to analysis works for unseen data. First, we manually inspected the learned analyzer at the first iteration

of the *Synthesize* procedure (without any counter-examples generated). We did that to check if we overfit to the initial dataset and found that indeed, the initial analysis would *not* generalize to some programs outside the provided dataset. This happened because the learned rules conditioned on unrelated regularities found in the data (such as variable names or fixed positions of certain function parameters). Our oracle, and the counter-example learning procedure, however, eliminate such kinds of non-semantic analyses by introducing additional function arguments and statements in the test cases.

Overfitting to the initial dataset was also caused by the large search space of possible programs in the DSL for the analysis. However, we decided not to restrict the language, because a more expressive language means more automation. Also, we did not need to provide upfront partial analysis in the form of a sketch [31].

Oracle Effectiveness for Finding Counter-Examples. We evaluate the effectiveness of our oracle to find counter-examples by comparing it to a random ("black box") oracle that applies all possible modifications to a randomly selected program from the training dataset. For both oracles we measure the average number of programs explored before a counter-example is found and summarize the results in Table 3. In the table, we observe two cases: (i) early in the analysis loop when the analysis is imprecise and finding a counter-example is *easy*, and (ii) later in the loop when *hard* corner cases are not yet covered by the analysis. In both cases, our *oracle guided by analysis* is orders of magnitude more efficient.

Table 3. The effect of using the learned analysis to guide the counter-example search.

Difficulty	Programs explored until first counter-example is found	
	"Black box"	Guided by analysis
Easy ($\approx 60\%$)	146	13
Hard ($\approx 40\%$)	> 3000	130

Is Counter-Example Refinement Loop Needed? Finally, we compare the effect of learning with a refinement loop to "one-shot" learning without the loop, but with more data provided up-front. For this experiment, we automatically generate a huge dataset $\mathcal{D}_{huge}$ by applying all possible program modifications (as defined by the oracle) on all programs in $\mathcal{D}$. For comparison, let the dataset obtained at the end of the counter-example based algorithm on $\mathcal{D}$ be $\mathcal{D}_{ce}$. The size of $\mathcal{D}_{ce}$ is two orders of magnitude smaller than $\mathcal{D}_{huge}$.

An analysis that generalizes well should be sound and precise on both datasets $\mathcal{D}_{ce}$ and $\mathcal{D}_{huge}$, but since we use one of the datasets for training, we use the other one to validate the resulting analyzer. For the analysis that is learned using counter-examples (from $\mathcal{D}_{ce}$), the precision is around 99.9% with the remaining 0.01% of results approximated to the top element in the lattice (that is, it does not produce a trivially sound, but useless result). However, evaluating the analysis learned from $\mathcal{D}_{huge}$ on $\mathcal{D}_{ce}$ has precision of only 70.1% with

the remaining 29.1% of the cases being *unsound*. This means that $\mathcal{D}_{ce}$ indeed contains interesting cases critical to analysis soundness and precision.

Summary. Overall, our evaluation shows that the learning approach presented in our work can learn static analysis rules that handle various cases such as the ones that arise in JavaScript built-in APIs. The learned rules generalize to cases beyond the training data and can be inspected and integrated into existing static analyzers that miss some of these corner cases. We provide an example of both learned analyses in the extended version of this paper [1].

8   Related Work

Synthesis from Examples. Similar to our work, synthesis from examples typically starts with a domain-specific language (DSL) which captures a hypothesis space of possible programs together with a set of examples the program must satisfy and optionally an oracle to provide additional data points in the form of counter-examples using CEGIS-like techniques [31]. Examples of this direction include discovery of bit manipulation programs [19], string processing in spreadsheets [13], functional programs [7], or data structure specifications [9]. A recent work has shown how to generalize the setting to large and noisy datasets [28].

Other recent works [15,18] synthesize models for library code by collecting program traces which are then used as a specification. The key differences with our approach are that we (i) use large dataset covering hundreds of cases and (ii) we synthesize analysis that generalizes beyond the provided dataset.

Program Analysis and Machine Learning. Recently, several works have used machine learning in the domain of program analysis for task such as probabilistic type prediction [20,29], reducing the false positives of an analysis [25], or as a way to speed up the analysis [2,14,26] by learning various strategies used by the analysis. A key difference compared to our work is that we present a method to learn the static analysis rules which can then be applied in an iterative manner. This is a more complex task than [20,29] which do not learn rules that can infer program specific properties and [2,14,25,26] which assume the rules are already provided and typically learn a classifier on top of them.

Learning Invariants. In an orthogonal effort there has also been work on learning program invariants using dynamic executions. For recent representative examples of this direction, see [8,21,30]. The focus of all of these works is rather different: they work on a per-program basis, exercising the program, obtaining observations and finally attempting to learn the invariants. Counter-example guided abstraction refinement (CEGAR) [3] is a classic approach for learning an abstraction (typically via refinement). Unlike our work, these approaches do not learn the actual program analysis and work on a per-program basis.

Scalable Program Analysis. Another line of work considers scaling program analysis in hard to analyse domains such as JavaScript at the expense of analysis soundness [6,24]. These works are orthogonal to us and follow the traditional way of designing the static analysis components by hand, but in the future they can also benefit from automatically learned rules by techniques such as ours.

9   Conclusion and Future Work

We presented a new approach for learning static analyzers from examples. Our approach takes as input a language for describing analysis rules, an abstraction function and an initial dataset of programs. Then, we introduce a counter-example guided search to iteratively add new programs that the learned analyzer should consider. These programs aim to capture corner cases of the programming language being analyzed. The counter-example search is made feasible thanks to an oracle able to quickly generate candidate example programs for the analyzer.

We implemented our approach and applied it to the setting of learning a points-to and allocation site analysis for JavaScript. This is a very challenging problem for learning yet one that is of practical importance. We show that our learning approach was able to discover new analysis rules which cover corner cases missed by prior, manually crafted analyzers for JavaScript.

We believe this is an interesting research direction with several possible future work items including learning to model the interfaces of large libraries w.r.t to a given analysis, learning the rules for other analyzers (e.g., type analysis), or learning an analysis that is semantically similar to analyses written by hand.

References

1. Bielik, P., Raychev, V., Vechev, M.T.: Learning a static analyzer from data. CoRR, abs/1611.01752 (2016)
2. Cha, S., Jeong, S., Oh, H.: Learning a strategy for choosing widening thresholds from a large codebase. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 25–41. Springer, Cham (2016). doi:10.1007/978-3-319-47958-3_2
3. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
5. Facebook: Facebook Flow: Static typechecker for JavaScript (2016). https://github.com/facebook/flow
6. Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 752–761 (2013)
7. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 229–239 (2015)

8. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 499–512 (2016)

9. Gehr, T., Dimitrov, D., Vechev, M.: Learning commutativity specifications. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 307–323. Springer, Cham (2015). doi:10.1007/978-3-319-21690-4_18

10. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 261–273. ACM (2015)

11. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)

12. Guarnieri, S., Livshits, B.: GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 151–168 (2009)

13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 317–330 (2011)

14. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 237–256. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53413-7_12

15. Heule, S., Sridharan, M., Chandra, S.: Mimic: computing models for opaque code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 710–720 (2015)

16. Jang, D., Choe, K.-M.: Points-to analysis for JavaScript. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 1930–1937 (2009)

17. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03237-0_17

18. Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J.S., Solar-Lezama, A.: Synthesizing framework models for symbolic execution. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 156–167 (2016)

19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, vol. 1, pp. 215–224 (2010)

20. Katz, O., El-Yaniv, R., Yahav, E.: Estimating types in binaries using predictive modeling. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 313–326 (2016)

21. Kowalewski, S., Philippou, A. (eds.): TACAS 2009. LNCS, vol. 5505. Springer, Heidelberg (2009)

22. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 216–226 (2014)

23. Von Luxburg, U., Schoelkopf, B.: Statistical learning theory: models, concepts, and results. In: Inductive Logic, pp. 651–706 (2011)

24. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 499–509. ACM, New York (2013)

25. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 462–473 (2015)
26. Oh, H., Yang, H., Yi, K.: Learning a strategy for adapting a program analysis via Bayesian optimisation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pp. 572–588 (2015)
27. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
28. Raychev, V., Bielik, P., Vechev, M., Krause, A.: Learning programs from noisy data. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 761–774 (2016)
29. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 111–124 (2015)
30. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 388–411. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_21
31. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, pp. 404–415 (2006)