Unsupervised Learning of API Aliasing Specifications

Jan Eberhardt
DeepCode AG, Switzerland
jan@deepcode.ai

Samuel Steffen
ETH Zurich, Switzerland
samuel.steffen@inf.ethz.ch

Veselin Raychev
DeepCode AG, Switzerland
veselin@deepcode.ai

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

Abstract

Real world applications make heavy use of powerful libraries and frameworks, posing a significant challenge for static analysis as the library implementation may be very complex or unavailable. Thus, obtaining specifications that summarize the behaviors of the library is important as it enables static analyzers to precisely track the effects of APIs on the client program, without requiring the actual API implementation.

In this work, we propose a novel method for discovering aliasing specifications of APIs by learning from a large dataset of programs. Unlike prior work, our method does not require manual annotation, access to the library's source code or ability to run its APIs. Instead, it learns specifications in a *fully unsupervised* manner, by statically observing usages of APIs in the dataset. The core idea is to learn a probabilistic model of interactions between API methods and aliasing objects, enabling identification of additional likely aliasing relations, and to then infer aliasing specifications of APIs that explain these relations. The learned specifications are then used to augment an API-aware points-to analysis.

We implemented our approach in a tool called *USpec* and used it to automatically learn aliasing specifications from millions of source code files. *USpec* learned over 2000 specifications of various Java and Python APIs, in the process improving the results of the points-to analysis and its clients.

CCS Concepts • **Theory of computation → Program specifications**; • **Computing methodologies →** *Unsupervised learning*.

Keywords big code, unsupervised machine learning, specification, pointer analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314640

ACM Reference Format:
Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised Learning of API Aliasing Specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314640

1 Introduction

The widespread use of complex libraries and frameworks in modern programs poses a significant challenge for static analysis [26]. This challenge persists because of two difficulties. First, analyzing a program by including the source code of all libraries used by the program is typically infeasible. Second, while it would be ideal for static analyzers to know clean specifications summarizing library semantics, manually crafting such specifications is very costly due to the sheer number, diversity and intricacies of modern libraries.

Indeed, many works have addressed the challenge of automatically inferring specifications [2, 5, 7, 15, 17, 19, 23, 28, 33], including descriptions of points-to effects [6]. In particular, obtaining quality API aliasing specifications is crucial as points-to analysis is a core component of state-of-the-art static reasoning systems and its results are often used downstream, *e.g.,* for tasks such as vulnerability finding [18]. Thus, any method improving the results of points-to and alias analysis is likely to have a positive effect on the entire pipeline.

While techniques vary, current approaches require either the library's code [12, 20] or black-box access to its APIs and ability to execute dynamic traces on that API [6].

Unsupervised learning of aliasing specifications In this work, we present a novel method that automatically infers useful API aliasing specifications by learning from a large corpus of programs that use these APIs. In contrast to previous work, our approach only relies on statically observing *usages* of APIs, and does not require dynamically running tests against the API or knowing its implementation.

Our method is based on two key insights. First, since interactions between objects and APIs often follow similar patterns in real world code, we train a probabilistic model ϕ on a large dataset of programs so to capture these interactions. Second, we use the model ϕ to identify additional likely aliasing relations, namely those that lead to API interactions

that are more likely according to ϕ. We then extrapolate from these relations in order to infer API aliasing specifications. The resulting specifications induce additional aliasing, beyond the high probability relations already identified by ϕ. As the inferred specifications are interpretable, they can be directly examined by an expert or used to improve the results of classic points-to analyzers [3].

We implemented our method in a tool called *USpec (unsupervised specification learning)* and evaluated it for both Java and Python. Using millions of source code files from public repositories, *USpec* was able to automatically infer more than 2000 API aliasing specifications, spanning non-standard libraries such as the Jackson JSON library[1] and Android specific APIs such as SparseArray[2]. Integrating the inferred specifications into an API-aware may-alias analysis improved both its results as well as those of client analyses.

Main contributions Our main contributions are:

- A method for creating a probabilistic model of API-object interactions learned over a large dataset of programs interacting with APIs (§3–4).
- An approach for inferring practical API aliasing specifications based on the probabilistic model (§5).
- An augmented API-aware may-alias analysis which integrates the learned aliasing API specifications (§6).
- A complete implementation and evaluation of our method in a tool called *USpec*. We extensively evaluated *USpec* on both Java and Python, indicating that it can find interesting and practically useful aliasing specifications beyond the reach of prior work (§7).

2 Overview

We now provide an overview of our approach for learning likely API aliasing specifications, illustrated in Fig. 1.

Input Our method takes as input a large dataset of programs which make calls to API methods. Such a dataset can be obtained from public open-source code repositories (*e.g.,* GitHub). In the figure, we show a sample input program that includes several API method calls. For example, the program calls methods `put` and `get` of the Java `HashMap` API.

Extracting API unaware event graphs Each input program is processed using a static context-sensitive points-to analysis to obtain the *event graph* of the program (§3). In particular, this analysis is *unaware* of aliasing relations induced by API methods and hence, to maintain precision, assumes that return values of API calls do not alias with other objects.

Nodes of the constructed event graph represent usages of (abstract) objects in API methods, while edges capture aliasing relations. For instance, the event graph for the code snippet in Fig. 1 includes a node of the form $\langle$`getFile`, ret$\rangle$ indicating the return value of the call to `getFile`, and a node of the form $\langle$`put`, 2$\rangle$ indicating the second argument of the call to `put`. These two nodes are connected by a directed edge to indicate that the return value of `getFile` (the object pointed to by `f`) is used as a second argument to `put`.

Learning a probabilistic event graph model In the next step, we use the large set of event graphs obtained previously to learn a probabilistic model over event graph edges (§4). That is, we train a machine learning model which, given a pair of events and a feature describing the local surroundings of the events in the graph, returns the probability that the two events are connected by an edge.

Key insight: finding additional aliasing relations One of the key insights of our work is that this model also assigns high probability to some event pairs *not* connected by an edge. This is possible if the way in which a pair of objects is used in API methods suggests that these objects should alias. For example, assume that in the training data set, the function `getName` is often called using the return value of `getFile` as a receiver, such as in `db.getFile().getName()` (not shown in Fig. 1). As a result, the model might assign a high probability to the potential edge `a` $\rightarrow$ `b` (*i.e.,* `a` and `b` are considered to likely alias) in the following snippet

```
a = database.getFile();  ...  c = b.getName();
```

Similarly, the model might assign high probability to a potential edge `f` $\rightarrow$ `x` for the code snippet in Fig. 1, though it does not exist in the event graph. Due to `put` and `get`, aliasing indeed does occur, hence such an edge would be desirable.

Such a model is already helpful and can be used to infer new aliasing relations. However, this only makes sense for adding the few edges with highest confidence as typically, the proportion of non-aliasing variables is very large. Even in this case, it is likely the model will sometimes be confident that an inexistent edge exists, leading to a high false positive rate (§7.2). We believe this is a general problem of machine learning models predicting anomalous events. For example, similar issues occur for other tasks [1, 21], where the prediction accuracy on variable selection is > 85%, but as variable misuse bugs are rare, the true positive rate on the actual bug finding task is much worse. To address this problem, we focus on a restricted set of edges induced by a set of candidate specifications, thereby changing the distribution to one where most edges are likely to exist. The probabilistic model is used to only score those edges, allowing us to obtain the API specifications best explaining the model.

Learning likely API aliasing specifications To learn API specifications, we consider a particular hypothesis class captured by two general patterns: RetSame(s) describes that method s returns the same value when being called with equal arguments multiple times, and RetArg(t, s, x) describes that method t returns the x-th argument of a preceding call

[1] https://github.com/FasterXML/jackson
[2] https://developer.android.com/reference/android/util/SparseArray

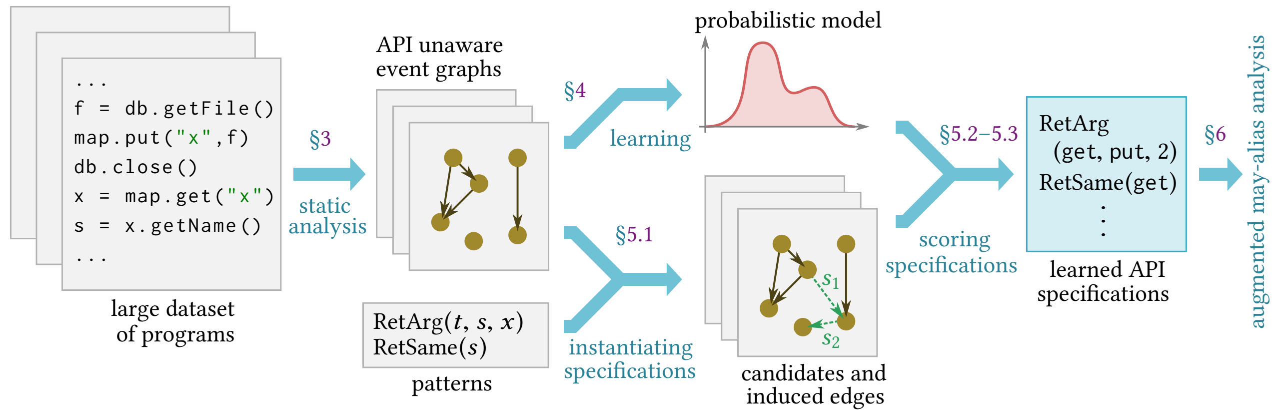


Figure 1. Overview of our approach for learning likely API aliasing specifications from a large dataset of programs.

to s if all other arguments are identical. As we will see, these patterns allow us to learn powerful aliasing specifications.

As a first step, we find all places in the event graph where the patterns above match, resulting in a set of candidate specifications (§5.1). For example, in the snippet of Fig. 1, the calls to `put` and `get` match the RetArg pattern and yield the candidate specification RetArg(`get`, `put`, 2). Naturally, each such candidate introduces new aliasing relations and edges to the event graph. For example, the specification above induces the edge `f` → `x`. At the end of this step, we obtain a set of new induced edges in the graph together with the originating candidate specifications (dashed edges in Fig. 1).

In the next step, we use the previously trained probabilistic model to score candidate specifications (§5.2). For each candidate, we query the model for its induced edges and obtain a score for that specification. For example, the specification RetArg(`get`, `put`, 2) might be assigned a high score as its induced edge for the shown code snippet is assigned a high probability. Using a threshold on the scores, we finally select the final specifications from the set of candidates (§5.3).

In §6, we show how to augment an Andersen-style points-to analysis [3] with these specifications in order to obtain an API aware may-alias analysis.

Examples of learned API specifications Applying our approach to millions of source code files (§7.1) allowed us to infer around 2000 API aliasing specifications, many of which are interesting and not immediately obvious. For example:

RetSame(`android.view.ViewGroup.findViewById`)
Calling findViewById with same id returns aliasing objects

RetSame(`com.fasterxml.jackson.databind.JsonNode.path`)
Calling path with same argument returns aliasing objects

RetArg(`get`, `put`, 2) for `android.util.SparseArray`
Calling get returns second arg. of preceeding put with equal key

Learning these specifications by means of static analysis and summarization [12, 26] of a library's implementation is beyond the reach of current fully automated techniques. For example, `android.util.SparseArray` is internally implemented with multiple arrays and binary search. Similarly, dictionary collections are implemented either as hash tables or trees. Analyzing these will require a non-trivial combination of numerical domains [29] and shape analysis [27].

3 Constructing Event Graphs

In this section, we present the notion of an *event graph*—an abstraction of object-API interactions. Event graphs will be used later in order to build a probabilistic model over edges.

3.1 Events and Concrete Histories

Our event graph model is based on the concept of object *histories* introduced by [25]. We next summarize this concept.

Consider an object-oriented programming language whose state $\langle L, \rho, \pi \rangle$ consists of (i) a set L of allocated objects, (ii) a function ρ assigning values to local variables, and (iii) a function π assigning values to fields of objects in L. The concrete semantics of the programming language defines transitions between program states according to the usual rules.

Let P be a program performing calls to external API libraries (*e.g.,* to interfaces whose implementations are not available) at different *call sites.* A call site m comprises the respective method call statement in P and its calling context (a sequence of method calls reaching the call site). The fully qualified method name and signature of the function called at m is the *method identifier* for m, denoted $\text{id}(m)$. We use $\text{nargs}(m)$ to denote the number of arguments of $\text{id}(m)$.

Let o be an object allocated in P and $\mathcal{M}$ be the set of all call sites in P. One can track the interactions of o with API methods by recording *events.* An event is a pair $\langle m, x \rangle$ of a call site $m \in \mathcal{M}$ and a *position* $x \in \text{Pos}$, where $\text{Pos} := \mathbb{N} \cup \{\text{ret}\}$ for a special value "ret". Here, x takes (i) a value in $\{1, \ldots, \text{nargs}(m)\}$ if o is passed as the x-th argument to m, (ii) the value 0 if o is the receiver of m, or (iii) the value "ret" if m returns o. The sequence of events for o is the *history* of o. The set of histories is $\mathcal{H} := (\mathcal{M} \times \text{Pos})^*$.

```
Map<String, File> map = new HashMap<>();
map.put("key": s1, someApi.getFile(): o1);
String name = map.get("key": s2): o2.getName();
```

`map`	$\{ (\langle \text{newMap}, \text{ret}\rangle, \langle \text{put}, 0\rangle, \langle \text{get}, 0\rangle) \}$
s_1 `"key"`	$\{ (\langle \text{lc}_1, \text{ret}\rangle, \langle \text{put}, 1\rangle) \}$
o_1	$\{ (\langle \texttt{getFile}, \text{ret}\rangle, \langle \text{put}, 2\rangle) \}$
s_2 `"key"`	$\{ (\langle \text{lc}_2, \text{ret}\rangle, \langle \text{get}, 1\rangle) \}$
o_2	$\{ (\langle \text{get}, \text{ret}\rangle, \langle \texttt{getName}, 0\rangle) \}$
`name`	$\{ (\langle \texttt{getName}, \text{ret}\rangle) \}$

Figure 2. Example code snippet (top) and abstract histories (bottom). The gray annotations identifying literals and return values are not part of the source code.

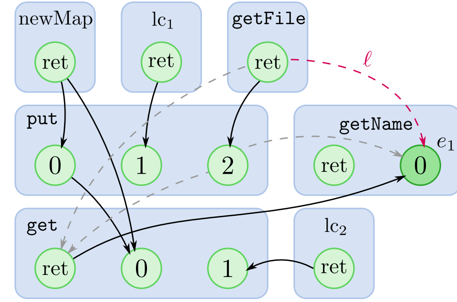


Figure 3. Solid arrows: event graph for code snippet in Fig. 2. Dashed arrows: added edges after identifying that $o_1 = o_2$ due to an aliasing specification of the `HashMap` API.

To capture the events an object participates in, we introduce the function $\text{his}\colon L \to \mathcal{H}$ assinging histories to objects. Then, at each call site, the histories of all involved objects are extended accordingly. Note that a call site m may generate potentially multiple events: $\langle m, 0\rangle$ for the receiver object, $\langle m, \text{ret}\rangle$ for the returned object (if any), and events for any function arguments. In contrast to [25], we introduce an event of the form $\langle \text{newT}, \text{ret}\rangle$ at allocation statements for objects of type T (*e.g.,* at `t = new T()`). For each occurrence of a literal in the program (*e.g.,* string literals), we introduce a literal construction event of the form $\langle \text{lc}_i, \text{ret}\rangle$.

3.2 Points-to Analysis and Abstract Histories

We now discuss our starting aliasing assumptions on the points-to analysis as well as the resulting abstract histories.

Modeling API effects on aliasing Using a static points-to analysis such as [3], the potentially infinite set of dynamically allocated objects can be partitioned into a bounded set of *abstract objects* [25]. A key point here is to decide how to model API invocations. On one side, we can conservatively assume that return values of API method calls alias with all other objects. Unfortunately, while sound, such an analysis is too imprecise in practice. Alternatively, we can assume that the returned value always corresponds to a fresh object, and hence this value never aliases with any other abstract object. Such an assumption is unsound, however, it leads to much better precision in practice. In our work, we start with the latter assumption and then selectively add aliasing relations for which we have sufficient confidence, thereby maintaining precision of the analysis while increasing points-to coverage.

Abstract histories Once we have performed the points-to analysis under the assumption that return values of APIs never alias with any other object, we can use the resulting abstract objects to construct their *abstract histories* [25].

An abstract history naturally lifts the concept of a concrete history and has the form $\text{his}\colon L \to \mathcal{P}(\mathcal{H})$, where each abstract object is assigned a set of concrete histories $h \subseteq \mathcal{H}$. Then, at a call site involving an object, *all* concrete histories in the abstract history of the object are extended by the appropriate event. In practice, to bound the length of a history, we perform single loop unrolling. At control-flow joins, abstract histories are joined via set union.

Example of abstract histories Consider the example code snippet in Fig. 2 (top). The snippet involves six abstract objects, including two string literals (s_1, s_2) and two objects returned by API methods (o_1, o_2). Note that as discussed above, we assume that objects o_1 and o_2 are different.

The histories of the abstract objects are shown in Fig. 2 (bottom), where we use method names to indicate the unique call sites of the methods. For example, the object `map` is used as a receiver of `put` and `get` after it was constructed. String literals s_i are constructed in literal constructors lc_i.

3.3 Event Graph

We now define the event graph of a program. The abstract histories of all events in a program P induce a directed graph $G_P = (V, E)$, where V is the set of events in P and E encodes the ordering of events for the same abstract object. More precisely, E contains a directed edge (e_1, e_2) iff e_1 and e_2 occur in the same history of an object o and for all histories of o where both events are present, e_1 occurs *before* e_2. We refer to G_P as the *event graph* of P.

Fig. 3 (solid arrows) shows the event graph for the code snippet in Fig. 2. Here, the nodes represent events while a rectangular region represents a call site with all events stemming from that site. For example, the shaded node represents the event $e_1 = \langle \texttt{getName}, 0\rangle$.

The edges in G_P essentially model information flow between call sites in P. Note that by construction, the edges of an event graph form a transitive closure. If the points-to analysis is precise enough, the resulting event graph is typically robust to common code refactorings such as renamings, extractions and inlinings, as long as the same APIs are called.

Next, we inspect how an event graph describes points-to sets and thereby represents may-alias information.

Allocation events Let $\text{parents}_G(e) = \{e' \mid (e', e) \in E\}$ denote the set of parents of an event $e \in V$. Note that $\text{parents}_G(e) = \emptyset$ for an event e iff a new abstract object is allocated at e, in which case e is called an *allocation event*. We define the function $\text{alloc}_G : V \to \mathcal{P}(V)$ assigning to each event the set of allocation events of the respective object:

$$\text{alloc}_G(e) := \left\{ e' \;\middle|\; \begin{array}{c} \exists m.\ e' = \langle m, \text{ret}\rangle \ \wedge\ \text{parents}_G(e') = \emptyset \\ \wedge\ e' \in \text{parents}_G(e) \cup \{e\} \end{array} \right\}.$$

The set $\text{alloc}_G(e)$ thus represents the points-to set associated with event e. As usual, two events e, e' are considered to *may-alias* iff $\text{alloc}_G(e) \cap \text{alloc}_G(e') \neq \emptyset$.

For instance, in the event graph G in Fig. 3, $\text{alloc}_G(e_1) = \{\langle \texttt{get}, \text{ret}\rangle\} = \text{alloc}_G(\langle \texttt{get}, \text{ret}\rangle)$. This describes that the receiver of `getName` may alias with the return value of `get`.

Introducing aliasing with specifications Recall that the analysis of §3.1 assumes that a new abstract object is returned for each API method call. However, the semantics of APIs might well introduce aliasing relations. For example, in the case of `HashMap`, calling `get(k)` after `put(k,o)` returns `o`.

Consider again the code snippet of Fig. 2. An analysis aware of this API specification would identify that o_1 and o_2 are in fact the same object and thus their abstract histories would be merged into the following abstract history:

$$\{(\langle \texttt{getFile}, \text{ret}\rangle, \langle \texttt{put}, 2\rangle, \langle \texttt{get}, \text{ret}\rangle, \langle \texttt{getName}, 0\rangle)\}.$$

As a result, more edges will be added to the event graph (see all dashed arrows in Fig. 3 including ℓ), which would increase precision. Our goal will be to discover such API aliasing specifications.

In the next section, we introduce a probabilistic model over event graph edges which enables us to identify likely aliasing of events. Then, in §5, we use this probabilistic model in order to discover likely API specifications.

4 Probabilistic Event Graph Model

We now introduce our probabilistic model of event graph edges. This model is typically learned from a large dataset of available programs and aims to capture the likelihood that an aliasing relation between two events exists. In §4.1, we describe a machine learning model that returns the probability of an edge being present between two events, given a feature describing the surroundings of the events in the graph. We show how to obtain training data for this model in §4.2. In §4.3, we discuss how such a model can give rise to new event graph edges inducing additional aliasing relations—an idea we use when learning API aliasing specifications in §5.

4.1 Event Pair Classification

Next, we describe our machine learning model classifying pairs of events into two classes (0 or 1) representing whether or not two events are connected by an edge.

Features We define a *path* $p = (e_1, \ldots, e_k)$ to be a sequence of events such that each two consecutive events e_i, e_{i+1} are connected by an edge in E, that is, $(e_i, e_{i+1}) \in E$. Here, $k = |p|$ is the *length* of the path. Let $\textsc{Paths}_G$ be the set of all paths in an event graph G. The set of all paths of length at most k that include an event $e \in V$ is defined as the *context* of e:

$$\text{ctx}_{G,k}(e) := \{p \mid p \in \textsc{Paths}_G \ \wedge\ e \in p \ \wedge\ |p| \leq k\}.$$

For example, in Fig. 3, $\text{ctx}_{G,2}(e_1) = \{(\langle \texttt{get}, \text{ret}\rangle, e_1)\}$.

The feature $\text{ftr}(e_1, e_2)$ of an event pair (e_1, e_2) is defined as the following tuple, where $e_i = \langle m_i, x_i\rangle$:

$$\text{ftr}(e_1, e_2) = (x_1, x_2, \text{ctx}_{G,2}(e_1), \text{ctx}_{G,2}(e_2), \gamma(e_1, e_2)).$$

Here, $\gamma(e_1, e_2)$ represents additional information that (i) captures types of method arguments in m_1 and m_2, and (ii) relates m_1 and m_2 to guarding control-flow conditions. The features ftr were empirically shown to perform well for the task considered in this work (see §7.2).

Model We now define our machine learning model ϕ, where $\phi(\text{ftr}(e_1, e_2))$ returns the probability that $(e_1, e_2) \in E$. In our work, we use an individual logistic regression model for each pair of argument positions. More precisely, it is

$$\phi(\text{ftr}(e_1, e_2)) = \phi((x_1, x_2, c_1, c_2, d)) = \psi_{(x_1, x_2)}(c_1, c_2, d),$$

where $\psi_{(x_1, x_2)}$ is a logistic regression model (see §7.1).

4.2 Obtaining Training Data

In order to obtain positive training samples, we extract all edges from a large set of event graphs. For each extracted edge (e_1, e_2), we record the sample $(\text{ftr}(e_1, e_2), 1)$. Here, we modify $\text{ftr}(e_1, e_2)$ to ensure that there is no path between e_1 and e_2 in the union of their contexts. That is, we may remove some paths from both contexts, including the edge between e_1 and e_2. This ensures our model does not simply learn to predict the transitive closure.

Negative training samples are obtained from non-existent edges. Since most pairs of events are *not* connected by an edge, we use a subsampling approach to obtain sets of positive and negative samples with similar size. More precisely, to obtain negative samples of the form $(\text{ftr}(e_1, e_2), 0)$, we sample pairs of events (e_1, e_2) that occur in the same calling context but are not connected by an edge.

The collected samples are used to train the model ϕ by training the individual logistic regression models.

4.3 Identifying New Edges

A key idea of this work is that the above model can be used to identify *new* edges in event graphs. More specifically, a pair of events (e_1, e_2) in a graph $G = (V, E)$ might be assigned a high probability by ϕ even though $(e_1, e_2) \notin E$. This means that G would be "better explainable" by the probabilistic model if E included the edge. In this case, (e_1, e_2) is called an *edge candidate*. Such edge candidates particularly arise if the way a pair of objects is used in API methods strongly

Table 1. Specification patterns and their intuitive meaning. Here, s and t represent API methods instantiated later to obtain concrete specifications, and $x \in \mathrm{Pos} \setminus \{\mathrm{ret}, 0\}$.

$\mathrm{RetSame}(s)$	Calling s multiple times with equal arguments and receiver may return the same object.
$\mathrm{RetArg}(t, s, x)$	Calling t may return the x-th argument of a preceding call of s on the same receiver where all other arguments are equal.

suggests the two objects should in fact alias. That is, adding an edge candidate to the event graph, and thereby merging histories of previously unrelated objects, will lead to histories that are "better explained" by the probabilistic model.

For example, consider the edge ℓ in Fig. 3 whose addition to the event graph would result in merging the histories of objects o_1 and o_2. The edge ℓ describes that the object allocated by a call to `getFile` is later used as a receiver of `getName`. If this API usage pattern has been observed many times in the training data, ℓ may be assigned a high probability by ϕ, suggesting objects o_1 and o_2 should alias.

While introducing additional aliasing relations using edge candidates such as ℓ allows for improved precision, it does not yet provide us with the actual API specifications that explain why the edges were introduced. Next, we will show how to infer such deterministic and structured API specifications by building on the probabilistic model.

5 Learning API Aliasing Specifications

We now show how the probabilistic event graph model from §4 is used to learn aliasing specifications of API methods. We first instrodùce the class of considered specifications in §5.1. Then, we show how the probabilistic model is used to score candidates (§5.2) and to select good specifications (§5.3). Finally, in §5.4, we discuss how the set of inferred specifications can be extended to ensure additional consistency.

5.1 Hypothesis Class of API Specifications

We now introduce the hypothesis class of API specifications considered in our work. First, we introduce a set of *specification patterns*, which define our search space. These patterns are later instantiated with concrete methods so to obtain candidate specifications.

Specification patterns In Tab. 1, we show the two types of specification patterns and their intuitive meaning. The variables s and t (source and target) represent method identifiers of the same API and will be instantiated later to obtain concrete specifications. The variable x is a function argument position: $x \in \mathrm{Pos} \setminus \{\mathrm{ret}, 0\}$.

The pattern $\mathrm{RetSame}(s)$ expresses that calling the method s multiple times on the same receiver API object with equal arguments may return the same object. This models API methods that read some internal state without changing it. Note that not all methods behave this way (*e.g.,* the `next()` function of a Java `Iterator`).

The pattern $\mathrm{RetArg}(t, s, x)$ states that if a call to s is eventually followed by a call to t on the same receiver object, t may return the argument at position x in s if (i) the arguments at positions $i = 1, \ldots, x - 1$ in s are equal to the arguments at positions i in t, and (ii) the arguments at positions $j = x + 1, \ldots, \mathrm{nargs}(s)$ in s are equal to the arguments at positions $j - 1$ in t. This models the case where s stores some information which is later retrieved by t. For example, RetArg(`HashMap.get`, `HashMap.put`, 2) expresses that calling `get("key")` after `put("key",x)` might return `x`.

Values In order to define equality of arguments, we introduce the function val_G assigning a set of values to each event in an event graph G. For a literal construction event $e = \langle \mathrm{lc}_i, \mathrm{ret} \rangle$, we define $\mathrm{val}_G(e) = \{v_i\}$ to be the singleton set containing the value v_i of the constructed literal. Similarly, for an object construction event $e = \langle \mathrm{newT}, \mathrm{ret} \rangle$, we define $\mathrm{val}_G(e) = \{v\}$ with v being a unique identifier of the allocated object. For all other events e, we define:

$$\mathrm{val}_G(e) = \{v \mid \exists e' \neq e.\ v \in \mathrm{val}_G(e') \ \wedge\ e' \in \mathrm{alloc}_G(e)\}.$$

Note that for any event $e = \langle m, \mathrm{ret} \rangle$ where m is an API method call, $\mathrm{alloc}_G(e) = \{e\}$ and therefore $\mathrm{val}_G(e) = \emptyset$. This models the fact that we do not know which object or literal is returned by m. We use $\mathcal{V}$ to denote the set of values.

For example, in Fig. 3, $\mathrm{val}_G(\langle \mathrm{put}, 1 \rangle) = \{$`"key"`$\}$ as lc_1 constructs the string literal `"key"`, while $\mathrm{val}_G(e_1) = \emptyset$.

Matching and instantiating patterns We define a predicate $\mathrm{equal}_G(m_1, x_1, m_2, x_2)$ over call sites $m_1, m_2 \in \mathcal{M}$ and argument positions $x_1, x_2 \in \mathrm{Pos} \setminus \{\mathrm{ret}, 0\}$ as follows:

$$\mathrm{equal}_G(m_1, x_1, m_2, x_2) \iff \mathrm{val}_G(\langle m_1, x_1 \rangle) \ \cap\ \mathrm{val}_G(\langle m_2, x_2 \rangle) \ \neq \emptyset.$$

This expresses that the values for the x_1-th argument in m_1 and the x_2-th argument in m_2 are not distinct, *i.e.,* the two arguments may actually be the same object or literal value.

We define a pair of call sites (m_1, m_2) occurring in an event graph G to *match* a specification pattern R if m_1 and m_2 satisfy the conditions expressed in R. Formally, (m_1, m_2) matches the pattern $\mathrm{RetSame}(s)$ in $G = (V, E)$ iff

(C1) $\mathrm{id}(m_1) = \mathrm{id}(m_2)$ (same method name and signature)
(C2) $\mathrm{alloc}_G(\langle m_1, 0 \rangle) = \mathrm{alloc}_G(\langle m_2, 0 \rangle)$ (same receiver)
(C3) $(\langle m_2, 0 \rangle, \langle m_1, 0 \rangle) \in E$ (m_2 is called before m_1)
(C4) $\forall i \in \{1, \ldots, \mathrm{nargs}(m_1)\}.\ \mathrm{equal}_G(m_1, i, m_2, i)$

The pair (m_1, m_2) matches the pattern $\mathrm{RetArg}(t, s, x)$ in G for some $x \in \mathrm{Pos} \setminus \{\mathrm{ret}, 0\}$ iff conditions *(C2)* and *(C3)* above are satisfied and

(C1′) $\mathrm{nargs}(m_2) = \mathrm{nargs}(m_1) + 1$

(C4′) The arguments satisfy

$$\forall i \in \{1, \ldots, x-1\}.\ \text{equal}_G(m_1, i, m_2, i)$$
$$\forall j \in \{x+1, \ldots, \text{nargs}(m_2)\}.\ \text{equal}_G(m_1, j-1, m_2, j)$$

If a pair of call sites (m_1, m_2) matches a pattern R, s (and t) can be instantiated with $\text{id}(m_2)$ (and $\text{id}(m_1)$, respectively) to obtain a *candidate specification*, denoted $\text{inst}(R, m_1, m_2)$.

Example Consider the example code snippet in Fig. 2 and its event graph in Fig. 3. The call site pair (`get`, `put`) matches

$$\text{RetArg}(t, s, 2) \quad (1)$$

and we can instantiate the following candidate specification:

$$\text{RetArg}(\texttt{HashMap.get}, \texttt{HashMap.put}, 2). \quad (2)$$

Induced edges Candidate specifications provide aliasing information about API methods. Next, we define the *induced edges* of a pattern match, which capture the aliasing relation expressed by the instantiated specification. We will later use induced edges to score candidate specifications.

Consider an event graph $G = (V, E)$. We define $\text{child}_G(e)$ for $e \in V$ to be the set of children of e in G:

$$\forall e, e' \in V. \quad e \in \text{child}_G(e') \iff e' \in \text{parents}_G(e).$$

A pair (m_1, m_2) matching $R = \text{RetArg}(t, s, x)$ induces the set of all edges from any allocation event of $\langle m_2, x\rangle$ to any child of $\langle m_1, \text{ret}\rangle$:

$$\text{induced}(R, G, (m_1, m_2)) = \{(e_1, e_2) \mid e_1 \in \text{alloc}_G(\langle m_2, x\rangle),\ e_2 \in \text{child}_G(\langle m_1, \text{ret}\rangle)\}.$$

A pair (m_1, m_2) matching $R = \text{RetSame}(s)$ induces the set of all edges from any child of $\langle m_2, \text{ret}\rangle$ to any child of $\langle m_1, \text{ret}\rangle$:

$$\text{induced}(R, G, (m_1, m_2)) = \{(e_1, e_2) \mid e_1 \in \text{child}_G(\langle m_2, \text{ret}\rangle),\ e_2 \in \text{child}_G(\langle m_1, \text{ret}\rangle)\}.$$

Example Matching pattern (1) with the pair (`get`, `put`) induces the dashed edge ℓ in Fig. 3. The edge ℓ states that the receiver of `getName` may alias with the return value of `getFile`. Accepting a specification for a matching call site pair results in a history merge which not only adds the induced edges but potentially also other edges to the event graph. In the example of Fig. 3, all dashed edges are added when merging the histories for objects o_1 and o_2.

5.2 Finding and Scoring Candidate Specifications

Next, we show how candidate specifications are extracted from the input dataset and how they can be scored in order to select the desired specifications.

Edge confidence In §4.3, we discussed how the model ϕ also assigns high probabilities to some non-existing event graph edges. A core idea of our work is that we can issue queries for induced edges to obtain scores of pattern matches. That is, if a pattern matching a call site pair induces an edge,

Algorithm 1 Extracting candidate specifications.

1: **for** each event graph $G = (V, E)$ of input dataset **do**
2: $\quad \mathcal{A}_G \leftarrow \{(m_1, m_2) \mid (\langle m_2, 0\rangle, \langle m_1, 0\rangle \in E)\}$
3: $\quad$ **for** all $(m_1, m_2) \in \mathcal{A}_G$ **do**
4: $\quad\quad$ **if** (m_1, m_2) matches a pattern R **then**
5: $\quad\quad\quad S \leftarrow \text{inst}(R, m_1, m_2)$ ▷ candidate spec
6: $\quad\quad\quad \{\ell\} \leftarrow \text{induced}(R, G, (m_1, m_2))$
7: $\quad\quad\quad p \leftarrow \phi(\text{ftr}(\ell))$ ▷ edge confidence
8: $\quad\quad\quad$ append p to the list Γ_S

```
map.put("key", "value");
String value = map.get("key");
```

Figure 4. Snippet with low confidence for correct spec.

we query ϕ for the probability of that edge. If the probability is high, we may want to accept the candidate specification.

Consider the pattern match (1) and its induced edge ℓ in Fig. 3. We can query $\phi(\text{ftr}(\ell))$ (see §4.1) to obtain a probability p for ℓ. As seen earlier in §4.3, adding ℓ to the event graph will merge the histories of objects o_1 and o_2. Thus, p is intuitively a measure of the probabilistic model's confidence that o_1 is the same object as o_2. We call p the *edge confidence* of ℓ.

Extracting and scoring candidates We now show how to extract candidate specifications from the input dataset and how to score candidates based on edge probabilities.

Alg. 1 describes how candidate specifications are extracted. The algorithm returns for each candidate S a list Γ_S of edge confidences. Note that the same candidate might be instantiated by multiple pattern matches at different call sites. For each event graph, we identify the set $\mathcal{A}_G$ of call site pairs with identical receiver. In practice, it suffices to only consider pairs within bounded distance in the event graph (discussed in §7.1). Next, we check for each $(m_1, m_2) \in \mathcal{A}_G$ whether (m_1, m_2) matches any specification pattern. If yes, we instantiate the candidate specification S, construct the induced edge ℓ (we ignore cases inducing more than a single edge), and query ϕ to obtain the edge confidence of ℓ.

Our goal is to obtain a score $\text{score}(S)$ for each candidate S based on Γ_S. It is important to understand that it suffices for S to be treated as precise if only *some* values in Γ_S are high: a low edge confidence only suggests that an induced edge was not able to explain the particular flow of information in *that specific case*, but not that S is generally a bad choice. Note that we expect Γ_S to contain some low values as not all information flow can be explained by the probabilistic model. For instance, in Fig. 4, a match of (2) likely does not induce an edge with high confidence in ϕ since the latter (informally speaking) can not explain why `"value"` should be returned by `get`. However, the edge ℓ depicted in Fig. 3 (induced by the same specification) might be assigned high confidence as the merge of objects o_1 and o_2 can be explained by the API calls `getFile` and `getName` (see §4.3).

There are several ways to compute the score score(S) of S. Example scores include (i) the highest value in Γ_S, (ii) the 95-percentile of Γ_S, or (iii) the average of the k highest values in Γ_S. In our implementation, we use the last approach for $k = 10$ as we observed this to perform well empirically.

5.3 Selecting Specifications

Given score(S) for each collected candidate specification S, we retain S only if score(S) $\geq \tau$ for a threshold τ. We use $\mathcal{S}$ to denote the set of selected specifications. The threshold τ is a parameter of our system and can be adapted to tune the precision of $\mathcal{S}$. In 7.2 we show how different values for τ impact the precision and recall of the results.

Discussion On a high level, the specification patterns considered in our work capture behaviors of container-like APIs that store (RetArg) and read (RetSame) values to/from internal state of the same API object. We also experimented with different patterns, but the results were modest and hence we focused on the two that perform empirically well (see §7).

We note that our approach is fundamentally not restricted to these patterns or to specifications involving a single API object, class or library. We believe scoring other patterns (or *e.g.,* naming conventions) using our probabilistic model is an interesting future research direction.

5.4 Extending the Specification Set

The set $\mathcal{S}$ can be extended with further RetSame specifications so to make the resulting set more consistent. In particular, the target method t of a RetArg specification should return the same value when being called a second time on the same receiver with same arguments, because the RetArg specification also applies to the second call. This means that for every specification RetArg(t, s, x) in $\mathcal{S}$, we can add the specification RetSame(t) to $\mathcal{S}$ if not already contained in $\mathcal{S}$. This is a direct way of extending $\mathcal{S}$ with further likely specifications. After extending $\mathcal{S}$, it holds that

$$\forall t, s, x. \quad \text{RetArg}(t, s, x) \in \mathcal{S} \implies \text{RetSame}(t) \in \mathcal{S}. \quad (3)$$

In practice however, we observed that for most RetArg specifications in $\mathcal{S}$, the corresponding RetSame specification is already contained in $\mathcal{S}$ (that is, the extension described above adds only few specifications to $\mathcal{S}$).

Using the approach described in this section, we were able to infer 2100 specifications out of 3500 candidates over millions of source code files (see §7).

6 Pointer Analysis with API Aliasing Specs

We now show how to augment a standard Andersen-style points-to analysis [3] with the previously learned API aliasing specifications in $\mathcal{S}$.

In principle, to leverage $\mathcal{S}$, one could perform a points-to analysis of the program P (with the assumption that return values of APIs do not alias) and then extend its event graph G_P based on matches with specifications in $\mathcal{S}$. That is, we could check whether any possible pair of call sites in G_P with the same receiver matches a specification in $\mathcal{S}$. For each match, we would add the induced edge to G_P together with additional edges reflecting the corresponding history merge. The points-to set of an event e is $\text{alloc}_{G_P}(e)$.

Unfortunately, the set of call site pairs $\mathcal{A}_{G_P}$ defined in Alg. 1 is very large in practice (quadratic in the number of call sites). As we show in §7.1, at *learning time* we can reduce the size of $\mathcal{A}_{G_P}$ by only considering a subset of pairs without affecting the quality of the learned specifications. However, at analysis time, all pairs of events in $\mathcal{A}_{G_P}$ need to be considered as any pair of events can effect each other, which is too inefficient.

We address this issue by proposing an extension to the classic Andersen-style points-to analysis [3] such that the obtained abstract histories give rise to the same results as the above approach while being much more efficient. We next describe this adaptation.

6.1 Ghost Fields

In order to model information flow through APIs, we adopt the notion of *ghost fields* used in [6]. A ghost field is an artificial API object field that stores objects or literal values passed to the API. Ghost fields are used to abstract the semantics of an API by modeling container-like behavior. In our work, we use ghost fields to capture the effects of event graph edges induced by specifications.

The main idea is to define read and write operations on ghost fields at call sites based on specifications. For a specification of the form RetArg(t, s, x), we let method s write its x-th argument to the ghost field whose name is derived from the values of the remaining arguments, and we let t read that field. Similarly, for a specification RetSame(s), we let s read the ghost field whose name is derived from the arguments to s.

Example Consider again the code snippet of Fig. 2. We can model the flow of information proposed in specification (2) by (i) augmenting `map` with a ghost field named `"key"`, (ii) writing to that field in the `put` method, and (iii) reading that field again in the `get` method.

6.2 Reading and Writing Ghost Fields

We now describe the previously sketched idea more formally. In particular, we define how specifications in $\mathcal{S}$ induce read and write operations on ghost fields at call sites.

Let $\mathcal{I}$ be the set of method identifiers and let GHOSTS := $\mathcal{I} \times \mathcal{V}^*$ be the set of ghost field names. The first component of a ghost field name specifies the API method supposed to read the field. For example:

$$(\texttt{get}, \texttt{"the answer is"}, 42) \in \text{GHOSTS}.$$

$$\mathrm{ReadGh}_{\mathcal{S}}(m) := \Big\{(\mathrm{id}(m), v_1, \ldots, v_k) \mid \mathrm{RetSame}(\mathrm{id}(m)) \in \mathcal{S} \wedge v_1 \in \mathrm{val}_G(\langle m,1\rangle) \wedge \cdots \wedge v_k \in \mathrm{val}_G(\langle m,k\rangle)\Big\}$$

$$\mathrm{WriteGh}_{\mathcal{S}}(m) := \Big\{(v, f) \mid \exists t, x.\ \mathrm{RetArg}(t, \mathrm{id}(m), x) \in \mathcal{S} \wedge v \in \mathrm{val}_G(\langle m,x\rangle) \wedge f \in \mathcal{F}(m,x,t)\Big\}$$

$$\mathcal{F}(m,x,t) = \Big\{(t, v_1, \ldots, v_{k-1}) \mid v_1 \in \mathrm{val}_G(\langle m,1\rangle) \wedge \cdots \wedge v_{x-1} \in \mathrm{val}_G(\langle m,x-1\rangle) \wedge v_x \in \mathrm{val}_G(\langle m,x+1\rangle) \wedge \cdots \wedge v_{k-1} \in \mathrm{val}_G(\langle m,k\rangle)\Big\}$$

Figure 5. Definitions of ReadGh and WriteGh functions for a call site m in graph G with $k = \mathrm{nargs}(m)$.

We introduce two functions specifying operations on ghost fields at call sites. The function $\mathrm{ReadGh}_{\mathcal{S}} \colon \mathcal{M} \to \mathcal{P}(\textsc{Ghosts})$ assigns to each call site the set of ghost field names (if any) that may be read at the call site. The second function $\mathrm{WriteGh}_{\mathcal{S}} \colon \mathcal{M} \to \mathcal{P}(\mathcal{V} \times \textsc{Ghosts})$ assigns to each call site a set of pairs (v, f) indicating that the value v might be written to ghost field f of the receiver object at the call site.

In Fig. 5, we provide the formal definitions of these functions. The definition of $\mathrm{ReadGh}_{\mathcal{S}}(m)$ states that if there exists a matching RetSame specification in $\mathcal{S}$, m reads the ghost field named after the identifier of m and the values of the arguments. In particular, calling $\mathrm{id}(m)$ multiple times with the same arguments reads the ghost field with same name. The definition of $\mathrm{WriteGh}_{\mathcal{S}}(m)$ states that if there exists a RetArg specification in $\mathcal{S}$ whose source is equal to $\mathrm{id}(m)$, m writes all possible values of the argument at the x-th position to a ghost field with name f. The name f is built from the *target* method identifier and the values of the other arguments. Note that due to (3), we have that $\mathrm{RetSame}(t) \in \mathcal{S}$ in this case. Therefore, any following call of t on the same receiver will read the field f written by m.

For example, assume we have learned the specifications

$$\mathcal{S} = \{\mathrm{RetSame}(\texttt{get}), \mathrm{RetArg}(\texttt{get}, \texttt{put}, 2)\}. \tag{4}$$

and let m_1 and m_2 be the calls to `put` and `get` in Fig. 2, respectively. We then have $\mathrm{WriteGh}_{\mathcal{S}}(m_1) = \{(o_1, (\texttt{get}, \texttt{"key"})\}$, indicating that m_1 writes o_1 into ghost field $(\texttt{get}, \texttt{"key"})$ of `map`. Further, $\mathrm{ReadGh}_{\mathcal{S}}(m_2) = \{(\texttt{get}, \texttt{"key"})\}$, indicating that m_2 reads ghost field $(\texttt{get}, \texttt{"key"})$ of `map`.

6.3 Extending Andersen-Style Analysis

We now show how to extend an Andersen-style points-to analysis [3] based on the ReadGh and WriteGh functions.

Towards this, we extend the function π (discussed in §3.1) to also capture values of ghost fields. This is, we define $\pi(o, f)$ to be the set of objects pointed to by the ghost field $f \in \textsc{Ghosts}$ of object $o \in L$.

Table 2. Deduction rules of our points-to analysis for basic example statements. The bottom two rules are specific to our work while the others are based on [3].

Statement	Rule	Name	
`x = new T();`	$\dfrac{}{\{o\} \subseteq \rho(x)}$	`Alloc`	(o is a new object)
`x = y;`	$\dfrac{}{\rho(y) \subseteq \rho(x)}$	`Assign`	
`x.f = y;`	$\dfrac{o \in \rho(x)}{\rho(y) \subseteq \pi(o, f)}$	`FieldW`	
`x = y.f;`	$\dfrac{o \in \rho(y)}{\pi(o, f) \subseteq \rho(x)}$	`FieldR`	
`y.m(...);`	$\dfrac{o \in \rho(y) \quad (v, f) \in \mathrm{WriteGh}_{\mathcal{S}}(m)}{v \in \pi(o, f)}$	`GhostW`	
`x = y.m(...);`	$\dfrac{o \in \rho(y) \quad f \in \mathrm{ReadGh}_{\mathcal{S}}(m)}{\pi(o, f) \subseteq \rho(x)}$ (if $\pi(o, f) = \emptyset$, allocate an object $z \in \pi(o, f)$)	`GhostR`	

Deduction rules In Tab. 2, we show our extension of the inference rules employed by an Andersen-style [3] points-to analysis. The rules for ghost fields share similarities with the rules for dynamic property accesses proposed by Sridharan et al. [31]. To illustrate the core idea, we only show rules for basic statements. Rules for more complex statements and expressions are analogous. The first five rules are standard [3].

The rule `GhostW` states that executing `m` results in the points-to sets of the ghost fields f of all possible receivers `y` to include v, where v and f are defined by WriteGh. Here, the variable m represents the call site of `y.m(...)`.

The rule `GhostR` states that after assigning the return value of `m` to `x`, the points-to set of `x` must include all points-to sets of all ghost fields (as given by ReadGh) of all possible receivers `y`. It is important to note that $\pi(o, f)$ is empty if the ghost field has never been written before. In order to model the aliasing relation proposed by a specification RetSame(`m`), we need to ensure that two matching calls to `m` read the same abstract object. Hence, we allocate a new abstract object $z \in \pi(o, f)$ if $\pi(o, f) = \emptyset$.

6.4 Improving Points-to Coverage

We can further improve the results of our analysis in cases where values of API method arguments cannot be resolved.

For example, assume we have learned the specifications in (4) and consider the snippets in Fig. 6. As no specification about `api.foo()` is known, we cannot know its return value. In our current analysis, the latter is treated as an empty set. For instance, in Fig. 6a, we have $\mathrm{val}(\langle \texttt{put}, 1\rangle) = \emptyset$ and therefore $\mathrm{WriteGh}_{\mathcal{S}}(\texttt{put}) = \emptyset$ since no ghost field name can be constructed. Similarly, in Fig. 6b, $\mathrm{ReadGh}_{\mathcal{S}}(\texttt{get}) = \emptyset$ for the first call to `get`. However, we might want to include more

```
map.put(api.foo(), obj)
map.get("k1")
map.get("k2")
```

(a) Writing unknown ghost field

```
map.put("k", obj)
map.get(api.foo())
map.get("k")
```

(b) Reading unknown ghost field

Figure 6. Example of unresolvable API method call arguments in presence of an API `api` without specifications.

results in our analysis (at the risk of reducing precision) by modeling that `api.foo()` in the snippets may return any of `"k1"`, `"k2"`, `"k"`, or something else. That is, we want the return values of all calls to `get` in Fig. 6 to may-alias with `obj`. We achieve this by extending the functions ReadGh and WriteGh such that writes (resp. reads) of unknown ghost fields use a dedicated field $\top$ (resp. $\perp$), see App. A.

7 Evaluation

In this section, we evaluate our approach for learning API aliasing specifications. In §7.1, we present an implementation of our approach, called *USpec*, and introduce our input dataset. The subsequent evaluation consists of four parts. First, we inspect the quality and diversity of the inferred specifications (§7.2). Second, we investigate the quantitative effect of the inferred specifications on the augmented points-to analysis (§7.3). Third, we show how the augmented API-aware points-to analysis improves downstream client analyses (§7.4). Finally, we compare *USpec* to the most recent state-of-the-art method for inferring likely aliasing specifications based on dynamically executing the library APIs (§7.5).

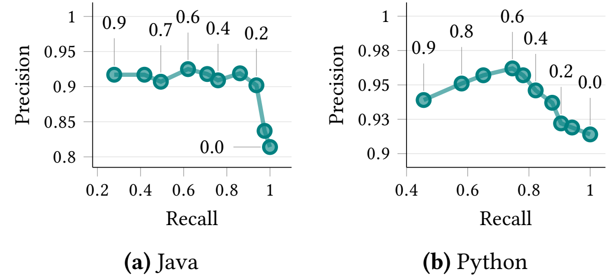
7.1 Implementation and Dataset

We implemented our approach for Java and Python, though our implementation is mostly language agnostic. Using event graphs as language independent program representations makes our approach applicable to most object-oriented programming languages. We believe that extending *USpec* to, for example, legacy C code—where there is no notion of classes and points-to analysis is inherently harder—is non-trivial.

Points-to analysis To obtain initial event graphs (§3) and for the augmented may-alias analysis (§6) we used a speed optimized Andersen-style flow- and context-sensitive interprocedural points-to analysis implemented in C++.

We note that *USpec*'s learning process is orthogonal to the initial points-to analysis (§3) and we experimentally observed that higher precision of this analysis yields better results. We believe that *USpec* can for example be used to improve aliasing information of dynamic analyses such as [6].

However, *USpec* can also be instantiated with a less precise initial analysis at the expense of little precision and recall: we experimented with a less precise *intra*procedural analysis and observed only a slight performance decline.



(a) Java **(b)** Python

Figure 7. Precision and recall of the selected specifications for different languages and thresholds τ (labeled datapoints).

Probabilistic model For the logistic regression model (§4), we used the off-the-shelf machine learning framework *Vowpal Wabbit.* [3] Each possible event graph path and each possible element in γ is encoded using a unique integer. A logistic regression feature (c_1, c_2, d) is then represented as a *set* of integers comprising the union of the encodings for all paths in c_1, c_2 and all elements in d. Such a set represents a sparse encoding of the feature in an over 100 million (resp. 40 million) dimensional feature space for Java (resp. Python).

We use a logistic regression model due to its simplicity and scalability. We also experimented with neural networks and word embeddings to take naming conventions into account. Though precision increased slightly, scalability suffered prohibitively. Recent works [11, 13] propose embeddings for source code, which could be readily used as features in the model. However, we did not further investigate code embeddings as the current features worked empirically well in our setting. Further, we believe that code embeddings face similar scalability problems as word embeddings.

Bounded candidate extraction When extracting candidate specifications from the input dataset (§5.2), we only include call site pairs (m_1, m_2) in $\mathcal{A}_G$ (Alg. 1) for which the distance of the receiver events $\langle m_1, 0\rangle$ and $\langle m_2, 0\rangle$ in the respective object history is at most 10. While we did not observe a negative effect on the inferred specifications, this improved performance of specification learning.

Dataset As input to *USpec*, we used a large dataset consisting of about 4 million (resp. about 1 million) Java (resp. Python) source code files from 33'251 (resp. 38'023) open-source GitHub repositories with at least 10 stars. We pruned our dataset to be free from project forks and file duplicates.

7.2 Quality of Learned Specifications

We now discuss the quality of the specifications learned by *USpec*. Particularly, we show how different thresholds τ (§5.3) impact precision and recall of the selected specifications. Further, we provide concrete examples of learned specifications, some of which are hard to capture using other tools.

[3] https://github.com/VowpalWabbit/vowpal_wabbit

Table 3. Some example specifications inferred using our implementation. The table indicates the API class, the number of matches in the training set, and the score for each specification. Many of the specifications are non-trivial.

API class	Specification	# matches	score
`java.util.HashMap`	RetArg(`get`, `put`, 2)	1'970	0.999
`java.security.KeyStore`	RetSame(`getKey`)	6	0.995
`java.sql.ResultSet`	RetSame(`getString`)	1'503	0.993
`android.util.SparseArray`	RetArg(`get`, `put`, 2)	40	0.948
`com.fasterxml.jackson.databind.JsonNode`	RetSame(`path`)	14	0.849
`android.view.ViewGroup`	RetSame(`findViewById`)	31	0.843
`org.antlr.runtime.tree.TreeAdaptor`	RetArg(`rulePostProcessing`, `addChild`, 2) **incorrect**	139	0.755
(python) `Dict`	RetArg(`SubscriptStore`, `SubscriptLoad`, 2)	28'414	0.999
(python) `List`	RetSame(`pop`) **incorrect**	254	0.983
(python) `configParser.SafeConfigParser`	RetArg(`get`, `set`, 3)	28	0.891

Precision versus recall In order to assess the quality of selected specifications, we ran the learning pipeline up to the point where the scored set of candidate specifications is obtained (§5.2). From these candidates, we randomly sampled 120 specifications and manually labeled each of them as being valid or invalid by inspecting the respective library documentation. Note that labeling may be hard. In cases of doubt, we conservatively labeled specifications as invalid.

Then, we used different thresholds τ to select specifications from the 120 candidates. For each threshold, we calculated the precision (*i.e.,* the fraction of valid specifications in the set of selected ones) and recall (*i.e.,* the fraction of selected candidates in the set of valid ones). We plot the precision vs. recall for different thresholds in Fig. 7. We note that the precision is high already for $\tau = 0$, indicating that many of the extracted candidates are already correct. Depending on the desired degree of points-to coverage and precision of the points-to analysis, the parameter τ can be tuned accordingly (higher recall increases coverage). For our remaining experiments, we set $\tau = 0.6$.

Characteristics of specifications Next, we ran the full specification learning pipeline (Fig. 1) using $\tau = 0.6$. Starting from already parsed source code in an intermediate binary representation, this took around 5 hours (resp. 2 hours) for Java (resp. Python) on a Xeon E5-2690 server with 28 cores and 512 GB of memory. This includes the time required for running the initial static analysis on all input projects and training the probabilistic model.

We note that unlike other approaches, *USpec* considers *all* libraries and API methods occurring in its input dataset and automatically learns specifications for these. Hence, the runtime of our system depends on the size of the input dataset, but not on the number of API classes.

In total, for Java (resp. Python) we extracted 1154 (resp. 2394) candidate specifications covering 536 (resp. 1488) API classes. From these, 621 (resp. 1438) specifications covering 313 (resp. 968) API classes were selected. A detailed breakdown of the number of selected specifications per library can be found in App. B.

In Tab. 3, we list several selected specifications together with the number of matches in the input programs and their score. The table also includes two incorrect specifications. We observe that some of the displayed specifications can not easily be obtained using existing tools. For instance, `java.sql.ResultSet` poses challenges to approaches relying on dynamic runs (see §7.5). Further, identification of most specifications learned by *USpec* is a challenge for static analyzers as the specifications include API classes with complex internal implementation: in a random sample of 30 API classes, we observed that all are non-trivial and make internal use of data structures. Many of the learned specifications are interesting and not immediately obvious. For instance, in 37% of the selected specifications (over all API classes), at least one of the source and target functions does not contain `get`, `put` or `set` in its name.

In summary, *USpec* is able to infer both precise and interesting aliasing specifications for a variety of API classes.

Alternative scoring functions We experimented with alternative scoring functions score$(\cdot)$ such as the number of matches, the number of repositories or files where a pattern matches, as well as linear combinations. We observed that the proposed scoring function performs best empirically. For instance, with $\tau = 0.6$ it results in precision 0.924 and recall 0.620 for Java (Fig. 7a). When instead using the number of matches as scoring function, higher precision can only be achieved at the price of strictly lower recall.

We note that scoring candidates is necessary. As discussed in §2, directly accepting all aliasing relations with high edge confidence (*i.e.,* without scoring candidate specifications) leads to a high number of false positives: we manually observed that for a confidence threshold of 0.5, around 1 out of 4 predicted additional event graph edges are incorrect. Further, if RetSame is assumed for all API functions, the false positive rates increase by almost a factor of two for

Table 4. Comparison of our API-aware aliasing analysis with an API unaware analysis for 100 call sites yielding different results. The table shows the the number of call sites where *USpec* increased points-to coverage and was more / less precise.

	increased points-to coverage while being precise		less precise because of wrong specification		less precise due to coverage increasing approach of §6.4		less precise (other)	
Java	86	≈ 1 per 320 loc	4	≈ 1 per 6'892 loc	8	≈ 1 per 3'446 loc	2	≈ 1 per 13'784 loc
Python	81	≈ 1 per 80 loc	0	-	16	≈ 1 per 405 loc	3	≈ 1 per 2'161 loc

both languages. Specifications like RetSame(`nextInt`) for `java.util.SecureRandom` (which is incorrect) are successfully filtered out by scoring based on the probabilistic model.

7.3 Effects on Points-to Analysis

Next, we study the effect of the learned specifications on points-to coverage and precision of points-to analysis. We compared our API-aware may-alias analysis (§6) using the specifications learned in §7.2 to a baseline analysis treating API method calls as if they always return new objects.

We randomly sampled 1000 Java and 1000 Python files from public GitHub repositories with 10 or more stars. We then analyzed each of these files with both analyzers and sampled 100 call sites per language yielding different aliasing information in the analyzers (note that at these sites, our approach always *increased* points-to coverage, meaning that more relations were identified). We manually inspected these call sites to classify each site into one of four categories: compared to the baseline, aliasing information of *USpec* was (i) increasing points-to coverage while maintaining precision, (ii) less precise because of an incorrect specification, (iii) less precise because of the coverage increasing approach of §6.4, or (iv) less precise for other reasons.

Tab. 4 shows the number of call sites in each category and the corresponding frequency per lines of source code (loc). We observe that for over 80% of call sites where points-to coverage increased, our analysis maintained precision. For Java (resp. Python) our analysis precisely increased points-to coverage of the baseline once every 320 (resp. 80) lines of code. While for Python we did not observe any usages of wrong specifications in the sample, such wrong specifications introduced incorrect aliasing information only once every 6'892 lines of code for Java. Even though the approach of §6.4 introduced imprecision in a few samples, we note that it increased points-to coverage in about half of the samples (not shown).

In conclusion, using our inferred specifications allows to increase points-to coverage at the cost of only slight loss of precision. In the next section, we will show how this helps to improve downstream client analysis.

7.4 Qualitative Effects on Client Analyses

Raw points-to analysis results are often used in some downstream client analysis. In this section, we exemplify how API-aware points-to sets help (i) a type-state analysis client checking correctness properties and (ii) a taint analysis client identifying security vulnerabilities. For both of these client analyses, the precision of the underlying may-alias analysis plays a vital role in providing quality results.

Type-state analysis Fig. 8a shows a real world code snippet [4] where a missing specification of the `java.util.List` API would lead to a false positive in a type-state client analysis verifying that `Iterator::hasNext` is true upon calling `Iterator::next`. If the alias analysis does not infer that the two calls of `get` return the same object, the property can not be verified. The property is important, because `next` throws an exception if `hasNext()` is false.

Taint analysis Fig. 8b shows a real code snippet vulnerable to a cross site scripting attack [9] because the user provided value `kwargs['value']` could flow into html text without sanitization. In order for a taint analysis client to accurately model this piece of code and find the vulnerability, a points-to analysis identifying `kwargs['data-value']` to alias with `kwargs.pop('value')` is required.

***Improvements with* USpec** Our tool automatically inferred specifications for the APIs used in the examples of Fig. 8 and our API-aware may-alias analysis increases points-to coverage by enough to prevent the discussed false positive and false negative in the client analyses.

We observed that developers often do not introduce new variables for return values of API methods with collection like behavior, but call the API method repeatedly instead (as shown in Fig. 8a). In many of these cases, *USpec* is able to infer sound aliasing information.

7.5 Comparison to State-of-the-Art

The recent state-of-the-art system Atlas [6] performs active learning using dynamic runs in order to infer precise points-to specifications. We ran the Atlas tool[4] for the most frequent classes used in the 1000 Java samples from §7.3. Atlas inferred sound points-to specifications for the `java.util` standard collections `Hashtable`, `ArrayList`, and `HashMap`, which however did not use precise ghost fields depending on the provided collection indexing keys. That is, the specification inferred by Atlas returns that reading from a collection may alias with *all* values inserted in that collection. These results are consistent with the paper and the source

[4] https://github.com/obastani/atlas

```
List<Iterator<Integer>> iters = ...;
for (int i=0; i<iters.size(); ++i) {
  if (iters.get(i).hasNext())
    SomeMethod(iters.get(i).next());
}
```

(a) Java code [4] with aliasing between get methods.

```
def __call__(self, ..., **kwargs):
  kwargs.setdefault('data-value',
      kwargs.pop('value', ''))
  return HTMLString(
    '<a %s>%s</a>' % (html_params(**kwargs,
        kwargs['data-value'])))
```

(b) Python code [9] with aliasing between `setdefault` and `kwargs`.

Figure 8. Code snippets using APIs. The snippets were simplified to highlight the APIs with learned specification.

repository of Atlas. Atlas produced unsound results for aliasing between the `getProperty` and `setProperty` methods of `java.util.Properties`, essentially learning that any call of these functions returns a new object. We were surprised that this class was not included in the Atlas evaluation despite being part of the standard library and being frequently used in practice and in our dataset. Further, for several other classes like `org.json.JSONObject`, Atlas inferred correct specification only for some of the methods and incorrectly learned that methods like `get` always return new objects. We believe the reason for this is that Atlas did not generate enough tests to fully cover the classes, which might be improved by tuning the search parameters. Finally, for classes like `org.w3c.dom.NodeList`, `java.sql.ResultSet` or `java.security.KeyStore`, Atlas failed to generate any non-empty specifications, because it could not figure how to call a constructor for these classes. This problem might be mitigated if additional specification is provided to the tool. None of the specifications produced by Atlas are instantiations of RetSame or RetArg patterns as Atlas' specifications do not take arguments into account.

In contrast to Atlas, *USpec* automatically inferred correct specifications for classes including `org.w3c.dom.NodeList`, `java.sql.ResultSet`, and `java.util.Properties`. Additionally, our inferred specifications allow a more precise alias analysis since we differentiate API calls based on the arguments at each call site.

We note that unfortunately, providing a quantitative comparison to Atlas (*e.g.*, in the style of Tab. 4) is difficult due to Atlas' implementation. While the approach of [6] is generally applicable, its implementation leverages various heuristics for invoking only default constructors with specific parameters targeting the Java standard library. Adapting Atlas to additional libraries requires changing its code in multiple places—a manual effort we only did for some libraries. Running Atlas on all classes in the evaluation dataset is virtually impossible without substantial changes to the system.

8 Related Work

In this section we discuss works closely related to ours.

API specification inference Inference of API specifications is widely studied. Solutions are based on dynamic executions [10, 14, 33], combinations with static analysis [19], or static analysis of library implementations [28]. Similarly to *USpec*, [15, 17, 23] observe API usages to predict specifications, however not directly targeting aliasing specifications. The work [23] infers preconditions by observing what actions users typically perform before calling an API, [15] infers type-state properties, and [17] scores possible information flows to predict security roles of APIs.

Points-to analysis and inference There is a long line of work addressing static points-to analysis summarized in [30], most often based on ideas by Andersen [3]. Pointer analysis in the presence of external libraries is challenging and has been addressed by many works. Rountev and Ryder [26] propose a technique where library modules and source code using them are analyzed separately and point-to summaries are statically computed for libraries. Recent work [32] measures the precision of points-to and numeric analysis when such summaries are used. The work of Bastani et al. [6] infers points-to specifications by synthesizing unit tests of library code in a tool called Atlas, to which we compare in §7.5. Further techniques [5, 34] require interaction with human experts to infer specifications. In contrast, our work infers specifications in a fully unsupervised way, without requiring the API's source code or black-box access to the library.

Big Code Our work relies on a large amount of programs to learn a probabilistic model of code. At a high level, similar techniques have been used for various other tasks, including predicting variable names and type annotations [16, 24], automatically correcting programs in MOOCs [22], and for statistical deobfuscation of code [8].

9 Conclusion

We presented a new approach for learning API aliasing specifications from a large dataset of programs. Our method is fully unsupervised and proceeds by learning a probabilistic model of aliasing by statically observing usages of APIs in the dataset. It then leverages the learned model to infer likely API aliasing specifications. The resulting specifications are used to augment existing may-alias analyzers so to improve their results when handling library APIs. We implemented our approach for Java and Python, and used it to infer thousands of aliasing specifications, many of which are interesting and challenging to obtain using existing methods.

Acknowledgments

The research leading to these results was partially supported by an ERC Starting Grant 680358.

$$\text{ReadGh}'_{\mathcal{S}}(m) := \begin{cases} \{\bot^{\text{id}(m)}\} & \text{if } \star \\ \text{ReadGh}_{\mathcal{S}}(m) \cup \{\top^{\text{id}(m)}\} & \text{otherwise} \end{cases}$$

$$\star\ \text{RetSame}(\text{id}(m)) \in \mathcal{S}\ \wedge\ \text{ReadGh}_{\mathcal{S}}(m) = \emptyset$$

$$\text{WriteGh}'_{\mathcal{S}}(m) := \Big\{(v, f) \mid \exists t, x.\ \text{RetArg}(t, \text{id}(m), x) \in \mathcal{S}\ \wedge$$
$$v \in \text{val}_G(\langle m, x\rangle)\ \wedge \Big(f \in \mathcal{F}(m, x, t) \cup \{\bot^t\} \vee \big(\mathcal{F}(m, x, t) = \emptyset\ \wedge\ f = \top^t\big)\Big)\Big\}$$

Figure 9. Definitions of ReadGh′ and WriteGh′. The differences to WriteGh are highlighted.

A Unknown Ghost Field Reads or Writes

First, we extend the set Ghosts of ghost field names with two kinds of special fields $\top^M$ and $\bot^M$ for method identifiers M. The field $\top^M$ is used to store any objects that would be written by a method call to a ghost field of the form $(M, \ldots)$ whose full name is unknown due to unknown argument values. The field $\bot^M$ is used to store all objects ever written to a ghost field of the form $(M, \ldots)$. The core idea is that (i) for all ghost field reads at any call site m, also $\top^{\text{id}(m)}$ must be read, and (ii) call sites m that would read a ghost field whose name is unknown must read $\bot^{\text{id}(m)}$.

We formally define functions ReadGh′ and WriteGh′ as shown in Fig. 9. For ReadGh′, the condition ⋆ expresses the fact that m is supposed to read a ghost field but at least one function argument has an unknown value. In this case, $\bot^{\text{id}(m)}$ is read. Otherwise, we read the same fields as ReadGh and additionally $\top^{\text{id}(m)}$. The definition of WriteGh′ extends WriteGh as highlighted. If the field name is unknown due to an unknown value, we write to $\top^t$. All writes are extended by a write to $\bot^t$.

For example, in Fig. 6a, `put` writes `obj` to fields $\top^{\texttt{get}}$ and $\bot^{\texttt{get}}$, and both calls of `get` return `obj` by reading from fields including $\top^{\texttt{get}}$. In Fig. 6b, `put` writes `obj` to fields (`get`, `"k"`) and $\bot^{\texttt{get}}$. The first call of `get` returns `obj` by reading from $\bot^{\texttt{get}}$, while the second call returns `obj` by reading from (`get`, `"k"`).

We adapt the deduction rules of §6.3 to use the new functions ReadGh′ and WriteGh′. In rule `GhostR`, the new object z is only allocated for $f \neq \top^{\text{id}(m)}$. This ensures that, *e.g.*, in Fig. 6a, the return values of the two calls to `get` would not alias if the call of `put` was missing.

B Selected Specifications by Library

Tab. 5 and Tab. 6 list the 12 Java and Python libraries for which the highest number of specifications were selected.

Table 5. Number of selected Java API specifications and spanned classes, grouped by Java package prefix for the 12 packages with most selected specifications.

Java Package Prefix	Specifications	API classes
java.util	94	37
org.eclipse	25	20
com.google	24	21
java.lang	17	8
org.json	17	4
org.apache	14	13
javax.swing	14	13
android.content	14	7
net.minecraft	14	6
org.w3c	11	6
android.util	9	3
org.codehaus	8	6

Table 6. Number of selected Python API specifications and spanned classes, grouped by Python library for the 12 libraries with most selected specifications.

Python Library	Specifications	API classes
numpy	37	30
pandas	14	7
os	10	9
re	10	6
django	9	8
collections	8	4
yaml	8	2
json	8	2
copy	8	2
flask	6	4
ConfigParser	6	3
xml	5	4

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL '02*. 13. https://doi.org/10.1145/503272.503275

[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. DIKU, University of Copenhagen.

[4] Adnan Aziz. 2016. epicode. https://github.com/adnanaziz/epicode/blob/884568f491146065472fafc32923e8aa73dd8076/java/src/main/java/com/epi/MergeSortedArrays.java#L38. Accessed: 27.03.2019.

[5] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM, New York, NY, USA, 553–566. https://doi.org/10.1145/2676726.2676977

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* ACM, New York, NY, USA, 678–692.

[7] Nels E. Beckman and Aditya V. Nori. 2011. Probabilistic, Modular and Scalable Inference of Typestate Specifications. In *Proceedings of*

the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11). ACM, New York, NY, USA, 211–221. https://doi.org/10.1145/1993498.1993524

[8] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[9] Paul Brown. 2015. Flask-Admin. https://github.com/flask-admin/flask-admin/commit/f447db0c78c03235d0dd6bdce0c5365fc6a7c321. Accessed: 27.03.2019.

[10] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: Mining Explicit Information Flow Specifications from Concrete Executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2771783.2771810

[11] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based Function Embedding and Its Application to Error-handling Specification Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 423–433. https://doi.org/10.1145/3236024.3236059

[12] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 567–577. https://doi.org/10.1145/1993498.1993565

[13] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018* (2018), 163–174. https://doi.org/10.1145/3236024.3236085 arXiv: 1803.06686.

[14] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing Framework Models for Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 156–167. https://doi.org/10.1145/2884781.2884856

[15] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 161–176. http://dl.acm.org/citation.cfm?id=1298455.1298471

[16] Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. 2017. Code Completion with Neural Attention and Pointer Networks. *CoRR* abs/1711.09573 (2017). arXiv:1711.09573 http://arxiv.org/abs/1711.09573

[17] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/1542476.1542485

[18] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251398.1251416

[19] Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic Generation of Program Specifications. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/566172.566213

[20] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. 165–180. https://doi.org/10.1007/978-3-540-27864-1_14

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

[22] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2016)*. ACM, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

[23] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static Specification Inference Using Predicate Mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 123–134. https://doi.org/10.1145/1250734.1250749

[24] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[25] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[26] Atanas Rountev and Barbara G. Ryder. 2001. Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, London, UK, UK, 20–36. http://dl.acm.org/citation.cfm?id=647477.727784

[27] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/292540.292552

[28] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *ISSTA '07*. 11. https://doi.org/10.1145/1273463.1273487

[29] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 46–59. https://doi.org/10.1145/3009837.3009885

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

[31] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-to Analysis of Javascript. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 435–458. https://doi.org/10.1007/978-3-642-31057-7_20

[32] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *ESOP 2018*. 653–682. https://doi.org/10.1007/978-3-319-89884-1_23

[33] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06*. 282–291. https://doi.org/10.1145/1134285.1134325

[34] Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated Inference of Library Specifications for Source-Sink Property Verification. In *APLAS'13*. Springer-Verlag, Berlin, Heidelberg, 290–306. https://doi.org/10.1007/978-3-319-03542-0_21