Bayesian Specification Learning for Finding API Usage Errors

Vijayaraghavan Murali
Rice University, USA
vijay@rice.edu

Swarat Chaudhuri
Rice University, USA
swarat@rice.edu

Chris Jermaine
Rice University, USA
cmj4@rice.edu

ABSTRACT

We present a Bayesian framework that can learn probabilistic specifications from large, unstructured code corpora, and then use these specifications to statically detect *anomalous*, hence likely buggy, program behavior. Our key insight is to build a statistical model that correlates specifications hidden inside a corpus with the syntax and observed behavior of their implementations. While analyzing a program, we condition this model into a posterior distribution that prioritizes specifications that are relevant to the program. The problem of finding anomalies is framed quantitatively, as a problem of computing a distance between a "reference distribution" over program behaviors that our model expects from the program, and the distribution over behaviors that the program actually produces.

We implement our ideas in a system, called Salento, for finding API usage errors in Android programs. Salento learns specifications using a combination of a topic model and a neural network model. Our experiments show that the system can discover subtle errors in Android applications in the wild, and outperforms a comparable non-Bayesian approach.

CCS CONCEPTS

• **Computing methodologies** → **Anomaly detection**; • **Software and its engineering** → **Automated static analysis**; **Software defect analysis**;

KEYWORDS

Anomaly detection, Specification Learning, Bug Finding, APIs

ACM Reference format:
Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17)*, 12 pages.
https://doi.org/10.1145/3106237.3106284

1 INTRODUCTION

Over the years, research on automated bug finding in programs has had many real-world successes [11, 14]. However, one perennial source of difficulty is the need for formal specifications. Traditional approaches require the user to specify correctness properties; any property that is not specified is outside the scope of reasoning. However, formally specifying real-world software is a difficult task that users often refuse to undertake.

A natural response to this difficulty is to *automatically learn* specifications of popular software components like APIs and frameworks. The availability of large corpora of open-source code makes this idea especially appealing [15]. By analyzing these corpora, one can generate numerous examples of how real-world programs use a set of components. Statistical methods can then be used to learn common patterns in these examples. According to the well-known thesis that bugs are *anomalous* behaviors [20, 28], a program whose use of the components significantly deviates from these "typical" usage patterns can be flagged as erroneous.

The problem of specification learning has been studied for a long time [6–8, 25]. However, existing approaches to the problem face two basic issues when applied to large code corpora. First, examples derived from such a corpus can be *noisy*. While programs in a mature corpus are likely to be mostly correct, not all examples extracted from such a corpus represent correct behavior. Second, such a corpus is fundamentally *heterogeneous*, and may contain many different specifications, some of them mutually contradictory. For example, it may be legitimate to use a set of APIs in many different ways, and a large enough corpus would contain instances of all these usage patterns. A specification learning tool should distinguish between these patterns, and a bug-finding tool should only compare a program with the patterns that are relevant to it.

Among existing methods for specification learning, the majority follow a traditional, qualitative view of program correctness. In this view, a specification is a set of program behaviors (e.g., sequences of calls to API methods), and a behavior is either correct (in the specification) or incorrect (outside the specification). Such an approach is not robust to noise because its belief in the correctness of a behavior does not change smoothly with the behavior's observed frequency. A small number of incorrect training examples can wrongly persuade the method that a behavior is correct.

An obvious fix to this problem is to view a specification as a *probabilistic* rather than a boolean model. Such a specification assigns quantitative *likelihood* values to observed program behaviors, with higher likelihood representing greater confidence in a behavior's correctness. Some recent work adopts this view by modeling program behaviors using models like n-grams [40, 52] and recurrent neural networks [47]. To find bugs using such a model, one generates behaviors of the target program using static or dynamic analysis, then evaluates the likelihood of these behaviors [53].

While robust to noise, approaches of this sort have a basic difficulty with heterogeneity. The root of this difficulty is that these methods learn a *single* probability distribution over program behaviors. For example, if Z_1 and Z_2 are two common but distinct patterns in which programs in a corpus use a set of APIs, these approaches would learn a specification that is a *mixture* of Z_1 and

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 ACM 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.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106284

Z_2. During program analysis, such a mixture would assign low likelihoods to behaviors that match one of, but not both, Z_1 and Z_2. As behaviors from a program are likely to follow only one of the two patterns, this phenomenon would lead to inaccurate analysis.

In this paper, we present a Bayesian approach to specification learning and bug finding that is robust to heterogeneity and noise. Our key insight is to build a statistical model that captures *the entire gamut of specifications* in an unstructured code corpus. More precisely, our model learns a joint probability distribution over a random variable Z representing hidden specifications and a random variable X representing *syntactic features* of implementations of these specifications. When using this model to analyze a particular program F with feature set X_F, we specialize it into a *posterior distribution* $P(Z|X = X_F)$ over specifications. Intuitively, this distribution assigns higher weight to specifications for programs that similar to F, and can be seen as the part of the model that is relevant to F.

This model architecture can tolerate high amounts of heterogeneity in the corpus. Suppose that the programs in the corpus use a set of APIs following distinct patterns $Z_1, \ldots, Z_n$, but that programs that look like F (i.e., have feature set X_F) tend to follow Z_1. During training, our framework learns this correlation between X_F and Z_1. This means that the posterior distribution $P(Z|X = X_F)$ puts a high weight on Z_1 and low weights on $Z_2, \ldots, Z_n$, and that correctness analysis of F happens only with respect to Z_1.

Our second key idea is to frame the detection of likely errors as an operation over distributions. We assume, for each program F, a distribution $P_F(Y)$ over the behaviors Y of the program. This distribution — a *probabilistic behavior model* — may be learned from data, or, as in this paper, be a *definition* that is a parameter of the framework. This allows us to develop a model $P(Y|Z = Z)$ of the behaviors Y of a program that follows a specification Z. When combined with the posterior $P(Z|X = X_F)$ for Z, this model gives us a "reference distribution" $P(Y|X = X_F)$ over behaviors that the model *expects* from a program that looks like F. The *anomaly score* of F, which quantifies the extent to which F behaves abnormally, is now defined as a statistical distance (in particular, the Kullback-Leibler divergence [34]) between $P(Y|X = X_F)$ and $P_F(Y)$.

Our Bayesian approach is a framework, meaning that it can be implemented using various concrete statistical models. In this paper, we instantiate it with a combination of the popular topic model Latent Dirichlet Allocation (LDA) [17], and a class of neural networks that are *conditioned* on a topic model [37]. To compute the anomaly score for a program F, we repeatedly query this model for the likelihood of different behaviors of F, and then aggregate these likelihood values into an estimate of the anomaly score.

We implement our ideas in a system called Salento, and use the system to detect API misuse in Android applications. Using three APIs as benchmarks, we show that Salento can automatically discover subtle API bugs in Android applications in the wild. These violations range from GUI bugs to inadequate encryption strength. Some of these errors are difficult to characterize in logic-based specification notations, indicating the promise of our approach in settings where traditional formal methods are hard to apply. We also show that the method has good precision and recall and is more robust to heterogeneity than a comparable non-Bayesian approach.

Now we summarize the contributions of this paper:

- We present a novel Bayesian framework for learning specifications from large code corpora (Section 3).
- We offer a novel formulation of the problem of finding anomalous program behavior as the problem of computing a distance between a program and a reference distribution (Section 3).
- We present an instantiation our framework with a topic model and a topic-conditioned recurrent neural network (Section 4).
- We evaluate the approach on the problem of detecting anomalous API usage in a suite of Android applications (Section 5).

2 OVERVIEW

In this section, we present an overview of our approach, with the help of an illustrative example.

2.1 Modeling Framework and Workflow

Our approach has the following key aspects. First, we assume the existence of a *specification* Z for each program F. However, unlike traditional approaches that *start* with a formal specification, Z in our context is not observable. Instead, what is observable is X_F, a set of syntactic *features* for F. The features are evidence, or data, that inform our opinion as to the unseen specification Z. In Bayesian fashion, our uncertainty about Z is formalized as a *posterior distribution* $P(Z|X = X_F)$, where Z is a random variable over specifications and X is a random variable over features. This distribution assigns higher likelihood to a specification if we believe it is more likely to be the correct specification for programs that "look like" F, given the evidence.

Second, we allow for uncertainty regarding the *behaviors* Y — defined as sequences of observable actions — that a given program F produces. This uncertainty comes from the fact that we do not fully know the inputs on which the program will run, and is captured by a distribution $P_F(Y)$, where Y is a random variable ranging over behaviors. The framework also allows for a distribution $P(Y|Z = Z)$ over the behaviors of programs that implement a given specification Z. This uncertainty can come from the fact that we do not know the inputs to implementations of Z, or the fact that we may have never seen a specification exactly like Z before, so that we have to guess the behavior of a program implementing Z.

Our *a priori* belief about the relationships between specifications and the features and behaviors of their implementations is given by a joint distribution $P(X, Y, Z)$. Our third key idea is that this distribution is informed by data extracted from a corpus of code. This information is taken into account formally during a *learning phase* that fits the joint distribution prior model to the data.

Finally, in the *inference phase*, we frame bug detection as a problem of computing a quantitative *anomaly score*. In traditional correctness analysis, the semantics of programs and specifications are given by sets, and one checks if the *set difference* between a program and a specification is empty. Our formulation is a quantitative generalization of this, and defines the anomaly score for a program F as the *Kullback-Leibler (KL) divergence* [34] between the behavior distribution $P_F(Y)$ for F, and the posterior distribution $P(Y|X = X_F)$ that the model expects from F. Correctness analysis amounts to checking whether this score is below a threshold.

The workflow of our method is as in Figure 1. The training and inference phases are denoted by green (solid) edges and red

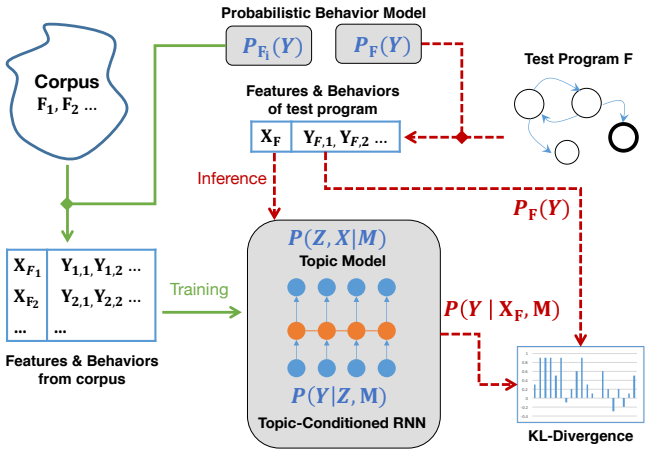


Figure 1: Workflow, with instantiations in grey boxes

(dashed) edges respectively. During training, from each program F_i in a corpus of programs $F_1, F_2, \ldots$, we extract a set of features X_{F_i}, and sample a set of behaviors from the distribution $P_{F_i}(Y)$, forming the training data. From this data, we learn the joint distribution $P(X, Y, Z \mid \mathbf{M})$, where $\mathbf{M}$ represents the model parameters.

During inference, we extract the features X_F of a given program F, and query the trained model for the distribution $P(Y|X = X_F, \mathbf{M})$ that tells us how F should behave. Separately, we obtain the distribution $P_F(Y)$ over observed behaviors of F. The anomaly score of F is then computed as the KL-divergence between these distributions.

2.2 Instantiating the Framework

An instantiation of our framework must concretely define program features and behaviors, and the way in which the distributions $P(X, Y, Z \mid \mathbf{M})$ and $P_F(Y)$ are obtained. In this paper, we consider a particular instantiation, embodied in the Salento tool, where the goal is to learn patterns in the way programs call methods in a set of APIs. We abstract each such call as a *symbol* from a finite set, and define a behavior Y as a sequence of symbols. The feature X_F for a program F is the set of symbols that F can generate.

A key idea in this instantiation is to capture hidden specifications using a *topic model*. Here, "topic" is an abstraction of the hidden semantic structure of a program. A specification for a program F is a vector of probabilities whose the i-th component is the probability that F follows the i-th topic. For example, the topics in a given corpus may correspond to GUI programs and bit-manipulating programs. A program that makes many calls to GUI APIs will likely have a higher probability for the former topic.

Specifically, we use *Latent Dirichlet Allocation (LDA)* [17] to learn a joint distribution $P(Z, X|\mathbf{M})$ over the topics and features of programs. A *topic-conditioned recurrent neural network* model [37], is used to learn conditional distributions of the form $P(Y|Z = Z, \mathbf{M})$. The joint distribution $P(X, Y, Z|\mathbf{M})$ that our framework maintains can be factored into these two distributions.

Our probabilistic model $P_F(Y)$ for behaviors of programs F is *not* data-driven. This is because to learn this distribution statistically, we would need data on the inputs that F receives in the real world. Since such data is hard to get, we simply *assume* a definition of $P_F(Y)$. While many such definitions are possible, the one we pick models F as a class of automata, called *generative probabilistic automata* [7, 39]. The distribution $P_F(Y)$ is simply the semantics of this automaton.

2.3 Example

Consider the problem of finding bugs in GUIs, where the right and wrong ways of invoking GUI API methods are seldom formally defined. Specifically, consider a dialog box in a GUI that does not give the user an option to close the box, and a dialog box that does not display any textual content. Clearly, such boxes violate user expectations, and are buggy in that sense. Two such boxes, produced by real-world Android apps, are shown in Figure 2(a).

The code snippets responsible for these boxes are shown in Figure 2. For example, in Figure 2(b)(i), b is a dialog box; the method `b.setItems(...)` adds content to the dialog box; the method `b.show()` displays the box. If the branches in lines 4 and 7 are not taken, then `b.show()` opens the box without a "close" button. Note that the sequences of API calls that lead to these bugs are not forbidden by the API, and would not be caught by a traditional program analysis. In contrast, a statistical method like ours can observe thousands of programs and learn that these sequences are abnormal.

Operationally, to debug this program using Salento, we generate features and behaviors from a corpus of Android apps. Using these features, LDA learns to classify programs by the APIs they use, and to also distinguish between different usage patterns in the same API. Consider the examples of dialog box creation in Figure 2(b), where program F_1 in (b)(i) explicitly specifies the items that go into the box, and the program F_2 in (b)(ii) provides a `View` that encompasses the items that go into the box. LDA can assign different topics to these usage patterns. For example, the pattern used in F_1 could be assigned the first topic, resulting in a topic vector (Z) $\langle 0.98, 0.01, 0.01 \rangle$, and the pattern used in F_2 could be assigned the second topic, resulting in the topic vector $\langle 0.01, 0.98, 0.01 \rangle$.

Conditioned on such a topic vector Z, a topic-conditioned RNN provides the probability of an API call sequence Y, that is, $P(Y = Y|Z = Z)$. For instance, given the former topic vector, a topic-conditioned RNN trained on thousands of examples of topics and behaviors would provide a high probability to a sequence such as:

```
new A() setTitle(...) setItems(...) show()
```

and a low probability to an abnormal sequence such as

```
new A() setTitle(...) show()
```

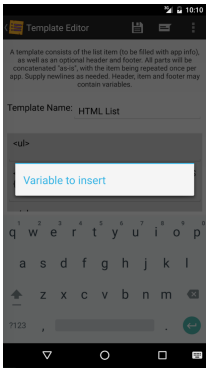
as it shows a dialog without any content. However, our probabilistic automaton model $P_{F_1}(Y)$ of F_1 assigns about 0.66 and 0.33 probability, respectively, to these sequences. In general, the KL-divergence between the two distributions is high, causing F_1 to be flagged as anomalous.

3 BAYESIAN SPECIFICATION FRAMEWORK

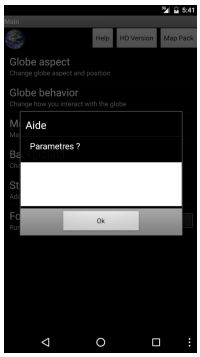
In this section, we formalize our framework, along with the problems of specification learning and anomaly detection.

3.1 Program Behaviors and Features

Our framework is parameterized by a programming language. Each program in the language has a syntax and an operational semantics. Because the details of the language do not matter to the framework,



(a)(i)



(a)(ii)

```
1  AlertDialog.Builder b = new AlertDialog.Builder(this);
2  b.setTitle(R.string.title_variable_to_insert);
3  if (focus.getId() == R.id.tmpl_item)
4      b.setItems(R.array.templatebodyvars, this);
5  else if (focus.getId() == R.id.tmpl_footer)
6      b.setItems(R.array.templateheaderfootervars, this);
7  b.show();
```

(b)(i)

```
1  AlertDialog.Builder b = new AlertDialog.Builder(this);
2  b.setMessage("Parametres?");
3  b.setCancelable(false);
4  b.setView(dlgLayout);
5  b.setPositiveButton("Ok", new OnClickListener(){...});
6  b.setTitle("Aide")
7  b.show();
```

(b)(ii)

Figure 2: (a) Abnormal dialog boxes discovered by our anomaly detection (b) Code snippets corresponding to the dialog boxes

we do not concretely define this syntax and semantics. Instead, we assume that the syntax of each program F can be abstracted into a *feature set* $\mathsf{X_F}$. For instance, such features can include syntactic constructs, assertions, and natural language comments. We also assume that program actions during execution can be abstracted into a finite alphabet Σ of *observable symbols* (including an *empty symbol* ϵ). We model program executions as *behaviors* Y, defined to be words in Σ^*. A behavior is the result of a probabilistic generative process that takes place when a program is executed. Accordingly, we assume a *probabilistic behavior model* of F, defined as a distribution $P_{\mathsf{F}}(Y)$ over the behaviors of F.

3.2 Specification Learning

Our framework builds a probabilistic model $P(X, Y, Z)$ that factorizes as $P(X, Y, Z) = P(Y|Z)P(X|Z)P(Z)$. The model captures the intuition that every program is implementing some unknown specification in the space of all specifications ($P(Z)$), which determines the program's behavior ($P(Y|Z)$) and features ($P(X|Z)$).

Building this model requires data, in the form of a large corpus of example programs. As in all statistical learning methods, we first develop an appropriate statistical model, which is typically a distribution family, and then *learn* that model—choose the parameters for the model family so they match reality—by training it on data. To this end, $P(X, Y, Z)$ also takes as input a set of model parameters $\mathbf{M}$. Fully parameterized, this distribution becomes:

$$P(X, Y, Z|\mathbf{M}) = P(Y|Z, \mathbf{M})P(X|Z, \mathbf{M})P(Z|\mathbf{M}) \tag{1}$$

The available data are then used to choose an appropriate set of parameters $\mathbf{M}$. For this, we follow the standard recipe of *maximum likelihood* [16]. Suppose that we are given a large corpus of programs $\{\mathsf{F}_1, \ldots, \mathsf{F}_N\}$, and for each program F_i we have extracted the pair $(\mathsf{X}_{\mathsf{F}_i}, \langle \mathsf{Y}_{i,1}\ \mathsf{Y}_{i,2}, \ldots \rangle)$ consisting of its feature set and a number of examples of its behavior sampled from its behavior model. Given this data, our goal is to choose $\mathbf{M}$ that maximizes the function:

$$\prod_{i=1}^{N} \left(\int_{\mathsf{Z}} \left(\prod_{\mathsf{Y}_{i,j}} P(\mathsf{Y}_{i,j}|Z = \mathsf{Z}, \mathbf{M}) \right) P(\mathsf{X}_{\mathsf{F}_i}|Z = \mathsf{Z}, \mathbf{M}) P(\mathsf{Z}; \mathbf{M})\ d\mathsf{Z} \right).$$

Note that we integrate out Z, since this is an unseen random variable, as we typically do not know the value of the precise specification associated with each code in the corpus. Once $\mathbf{M}$ is learned, the distribution would represent our prior belief as to what the "typical" specification, behavior and features look like, informed by the programs in the corpus.

3.3 Anomaly Detection

Suppose that we are given a new program F and would like to obtain a quantitative measure of the "bugginess" of F. On the one hand, since we already have learned a joint distribution over behaviors, features and specifications, $P(X, Y, Z|\mathbf{M})$, we can *condition* this distribution with the newly observed $\mathsf{X_F}$, to obtain the posterior:

$$P(Y, Z|X = \mathsf{X_F}, \mathbf{M}) = \frac{P(Y, Z, X = \mathsf{X_F}|\mathbf{M})}{P(X = \mathsf{X_F}|\mathbf{M})}$$

From Equation 1, we have

$$P(Y, Z|X = \mathsf{X_F}, \mathbf{M}) = \frac{P(Y|Z, \mathbf{M})P(X = \mathsf{X_F}|Z, \mathbf{M})P(Z|\mathbf{M})}{P(X = \mathsf{X_F}|\mathbf{M})}$$

Applying Bayes' rule to the term $P(X = \mathsf{X_F}|Z, \mathbf{M})$ we rewrite $P(Y, Z|X = \mathsf{X_F}, \mathbf{M})$ as

$$\begin{aligned} &= \frac{P(Y|Z, \mathbf{M})\dfrac{P(Z|X = \mathsf{X_F}, \mathbf{M})P(X = \mathsf{X_F}|\mathbf{M})}{P(Z|\mathbf{M})}P(Z|\mathbf{M})}{P(X = \mathsf{X_F}|\mathbf{M})} \\ &= P(Y|Z, \mathbf{M})P(Z|X = \mathsf{X_F}, \mathbf{M}) \end{aligned}$$

From this, since we do not know the precise specification that F is implementing, we can integrate out Z to obtain the (marginalized) posterior distribution over behaviors:

$$P(Y|X = \mathsf{X_F}, \mathbf{M}) = \int_{\mathsf{Z}} P(Y|Z = \mathsf{Z}, \mathbf{M})P(\mathsf{Z}|\mathsf{X_F}, \mathbf{M})\ d\mathsf{Z} \tag{2}$$

This form is amenable to Monte Carlo integration, which estimates an integral through random sampling. Intuitively, it gives us a distribution over the program behaviors Y, that would be *anticipated*, given learned parameters $\mathbf{M}$, for a program with feature set $\mathsf{X_F}$.

On the other hand, we have a distribution $P_{\mathsf{F}}(Y)$ over the *actual* behaviors of F when it is executed. The final step is to then compare this actual distribution with the anticipated distribution over

behaviors, that is, $P(Y|X = \mathsf{X_F}, \mathbf{M})$. A measure such as the Kullback-Leibler (KL) divergence [34] between distributions is appropriate here. The KL-divergence between two distributions P_1 and P_2 over the domain i is a quantitative measure defined as:

$$D_{KL}(P_1 \| P_2) = \sum_i P_1(i) \log \frac{P_1(i)}{P_2(i)} \tag{3}$$

Using this measure, we can compute the *anomaly score* of F by setting P_1 and P_2 to the distributions $P_\mathsf{F}(Y)$ and $P(Y|X = \mathsf{X_F}, \mathbf{M})$ respectively, and ranging i over the domain of all possible program behaviors in the language Σ^*:

$$\sum_{\mathsf{Y} \in \Sigma^*} P_\mathsf{F}(Y = \mathsf{Y}) \log \frac{P_\mathsf{F}(Y = \mathsf{Y})}{P(Y = \mathsf{Y}|X = \mathsf{X_F}, \mathbf{M})} \tag{4}$$

Choosing an Abstraction. When instantiating the framework, the exact form of the feature set $\mathsf{X_F}$ must be chosen with some care. If the feature set $\mathsf{X_F}$ does not provide any abstraction for the program (i.e., $\mathsf{X_F}$ is the program itself) and the model and learner are arbitrarily powerful, then $P(Y|X = \mathsf{X_F}, \mathbf{M})$ (Equation 2) could, in theory, describe the compiler and symbolic executor used to produce the training data. This would mean that the KL divergence (Equation 3) is zero for any program.

When applying the framework to a problem, we protect against this possibility by choosing a feature set $\mathsf{X_F}$ that abstracts the program adequately. For example, when debugging API usage, it makes sense to choose $\mathsf{X_F}$ as the bag of API calls made by the code. This ensures that $P(Y|X = \mathsf{X_F}, \mathbf{M})$ is limited to attaching probabilities to sequences that can be made out of those calls, and it is impossible for the learner to "learn" to compile and execute a program.

4 INSTANTIATION OF THE FRAMEWORK

Now we present a concrete instantiation of our framework.

4.1 Probabilistic Behavior Model $P_\mathsf{F}(Y)$

First, our instantiation includes a definition of the probabilistic behavior model $P_\mathsf{F}(Y)$. This definition relies on the abstraction of programs as *generative probabilistic automata* [39, 49].

Program Model. A generative probabilistic automaton is a tuple $\mathsf{F} = \langle Q, \Sigma, q_0, Q_A, \delta \rangle$ where Q is a finite set of states, Σ is the alphabet of observable symbols that was introduced earlier, $q_0 \in Q$ is the *initial state*, $Q_A \subseteq Q$ is a set of *final or accepting states*, and $\delta : Q \times \Sigma \times \mathbb{R}_{(0,1]} \times Q$ is a *transition relation*. We have $\delta(q_i, s, p, q_j)$ if the automaton can transition between states q_i and q_j with a probability $p \in (0, 1]$, generating the symbol s. (We write $q_i \xrightarrow{s,p} q_j$ if such a transition exists.) Transitions with probability 0, or infeasible transitions, are excluded from the automaton.

A program in a high-level language is transformed into the above representation through *symbolic execution* [31] in a preprocessing phase. Symbolic execution runs a program with symbolic inputs and keeps track of *symbolic states*, analogous to a program's memory. The symbolic states encountered become the states Q, and the accepting states Q_A are typically the states at a final location (or some location of interest) in the program. Unbounded loops can be handled by imposing a bound on symbolic loop unrolls, or through a predicate abstraction of the program to make variable domains finite. The detection of infeasible states—in general an undecidable problem—depends on the underlying theorem prover that is used.

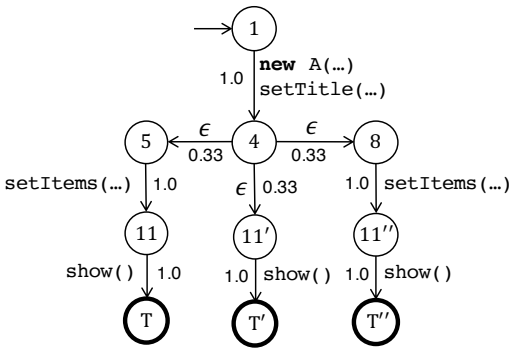


Figure 3: Automaton for the example in Figure 2(b)(i)

As symbolic execution is a standard method in formal methods [9, 18, 29], this section only gives an example of the method's use. As it is applied at a preprocessing level, we often use the term "program" to refer to an automaton generated via symbolic execution, rather than a higher-level program to which preprocessing is applied.

Semantics. A *run* π of F is defined as a finite sequence of transitions $q_0 \xrightarrow{s_1,p_1} q_1 \xrightarrow{s_2,p_2} \cdots \xrightarrow{s_n,p_n} q_n$ beginning at the initial state q_0. π is *accepting* if $q_n \in Q_A$. The probability of π is $P(\pi) = \prod_{i=1}^{n} p_i$. Every run π generates a behavior $\mathsf{Y} \in \Sigma^*$, denoted as $[\![\pi]\!] = s_1 s_2 \cdots s_n$. Let Π_F be the set of all accepting runs of F, and $\Pi_\mathsf{F}(\mathsf{Y}) \subseteq \Pi_\mathsf{F}$ be the set of all accepting runs π such that $[\![\pi]\!] = \mathsf{Y}$.

The *probabilistic behavior model* $P_\mathsf{F}(Y) : \Sigma^* \to [0, 1]$ is:

$$P_\mathsf{F}(Y = \mathsf{Y}) = \frac{1}{\nu} \sum_{\pi \in \Pi_\mathsf{F}(\mathsf{Y})} P(\pi) \tag{5}$$

where $\nu = \sum_{\pi \in \Pi_\mathsf{F}} P(\pi)$ is a *normalization factor*.

It is easy to see that $P_\mathsf{F}(Y)$ defines a probability distribution over behaviors. To generate a "random" behavior of F, we simply sample from the distribution $P_\mathsf{F}(Y)$.

Features. Given a program F, the feature set $\mathsf{X_F}$ is defined as $\{s \mid q_i \xrightarrow{s,p} q_j \in \delta\} \setminus \{\epsilon\}$, i.e., the set of all non-empty symbols in the transition system of F.

Example. The automaton for the code in Figure 2(b)(i) is shown in Figure 3. Each "state" in the automaton is labeled with a program location, with multiple instances of the same location being primed. The initial state is the first location, and the accepting states, in bold, are all instances of a (special) terminal location T in the program. The transitions follow the structure of the code (for brevity, we collapse sequential statements into a single transition), emitting as symbols API methods called at each location.

Note that we gave a uniform probability at each state to transition to the next possible states, but this can be controlled through other means. For instance, one can apply *model counting* on a branch condition and compute the probability of the program executing one branch over another. Such a definition is not necessarily a better choice than ours, as it would assign low probabilities to corner cases that get triggered on a small number of inputs but

are often of interest to users of static analysis. The two definitions simply make different tradeoffs. We use a uniform distribution at branches because it is simpler and worked well in our experiments.

{new A(), setTitle(...), setItems(...), show()} is the feature set for this program. There are three accepting runs of F, and two behaviors generated by these accepting runs:

Y_1 = new A() setTitle(...) setItems(...) show()
Y_2 = new A() setTitle(...) show()

We have $\nu = 1.0$, the sum of the probabilities of all accepting runs. Hence, $P_F(Y_1) = (0.33 + 0.33)/1.0 = 0.66$ and $P_F(Y_2) = 0.33$.

Assume now that after training on a large number of behaviors, the model had learned that conditioned on specifications such as $\langle 0.98, 0.01, 0.01\rangle$ (that gave a high probability to the first topic), program behaviors tend to always add a title and items to dialog boxes. This might result in the behavior Y_1 having a very high probability, say 0.99, and all other behaviors having a very low probability. Particularly, a behavior that only calls setTitle without setItems would be assigned a very low probability, say, 10^{-5}. In our program F, we saw that $P_F(Y_1) = 0.66$ and $P_F(Y_2) = 0.33$, and the probability of any other Y is 0. Thus, the anomaly score of F is: $0.66 \log \frac{0.66}{0.99} + 0.33 \log \frac{0.33}{10^{-5}} = 3.16$ Suppose now, that the state $11'$ in the program model was infeasible. Then, both accepting runs in the model would only generate Y_1, and so $P_F(Y_1) = 1$. The anomaly score of this "correct" program would then be $\log \frac{1}{0.99} = 0.01$.

4.2 Topic Models for $P(Z, X \mid \mathbf{M})$

Topic models are used in natural language processing to automatically extract topics from a large number of "documents" containing textual data as words. In our case, a document is the feature set of a program, words are symbols from the observable alphabet that a program uses, and the topic distribution of a document is its unknown specification.

LDA [17] is a popular topic model that models the generative process of documents in a corpus where each document X_{F_i} contains a bag of words. The inputs to LDA are the number K of topics to be extracted, and two hyper-parameters α and η. LDA models a document as a distribution over topics, and a topic as a distribution over words in the vocabulary. An LDA model is characterized by the variables: (i) α and η, hyper-parameters of a Dirichlet prior that chooses the topic distribution of each document and the word distribution of each topic, respectively (ii) Z_{F_i}, the topic distribution of document X_{F_i}, (iii) β_k, the word distribution of topic k.

The result of training an LDA model is a learned value for all the latent variables α, η, Z_{F_i} and β_k, which forms our model parameter $\mathbf{M}$. During inference, we are given a document X_F, and we would like to compute the posterior distribution $P(Z|X = X_F, \mathbf{M})$. Since LDA has already learned a joint distribution $P(Z, X|\mathbf{M})$, this is simply a matter of conditioning this distribution with the newly observed X_F to get a posterior distribution over Z, which is often approximated through a technique called *Gibbs sampling* [24].

4.3 Recurrent Neural Networks for $P(Y \mid Z, \mathbf{M})$

Neural networks have been used to solve classification problems such as image recognition and part-of-speech tagging. These problems involve classifying an input $\mathbf{x}$ into a set of (output) classes $\mathbf{y}$, using the conditional distribution $P(\mathbf{y}|\mathbf{x}, \mathbf{M})$.

Suppose we are given a value of $\mathbf{x}$: a given sequence of symbols (characters) $s_1 s_2 \ldots s_{t-1}$ where each symbol is from the alphabet Σ, and we would like the model to generate the next symbol s_t. We can cast this generative problem as a classification task by creating $x_1 x_2 \ldots x_{t-1}$, where each x_k is the *one-hot vector* of s_k, and querying the model to "classify" the sequence $x_1 x_2 \ldots x_{t-1}$ into $|\Sigma|$ classes. The output vector y_t is then interpreted as a distribution over Σ, from which a symbol s_t can be sampled [13]. Let us denote the probability of a symbol s given by the distribution y_t as $y_t(s)$.

A topic-conditioned neural network [37] takes, in addition to $\mathbf{x}$, an input Z representing the topic distribution of a document obtained from a topic model. To handle unbounded length input sequences, a *recurrent neural network* (RNN) is used. An RNN uses a hidden state to neurally encode the sequence it has seen so far. At time point t, the hidden state h_t and the output y_t are computed as:

$$h_t = f(\mathbf{W}h_{t-1} + \mathbf{V}\mathsf{Z} + \mathbf{U}x_t + \mathbf{b}_h),\ y_t = g(\mathbf{T}h_t + \mathbf{b}_y) \qquad (6)$$

where $\mathbf{W}$, $\mathbf{V}$, $\mathbf{U}$ and $\mathbf{T}$ are the weight matrices of the RNN, $\mathbf{b}_h$ and $\mathbf{b}_y$ are the bias vectors of the hidden states and outputs respectively, f is a non-linear *activation* function such as the sigmoid, and g is a *softmax* function that ensures that the output is a distribution.

Training the model involves defining an error function between the output of the RNN and the observed output in the training data. Specifically, if the training data is of the form $(X_{F_i}, \langle Y_{i,1}, Y_{i,2}, \ldots\rangle)$, then each training step of the RNN will consist of the input $\mathbf{x}$ being $Y_{i,j}$, target output $\mathbf{y}$ being $Y_{i,j}$ shifted by one position to the left (since at time point t the output y_t is interpreted as the distribution over the *next* symbol in the sequence), and Z being a sample from $P(Z|X = X_F, \mathbf{M})$ given by the trained topic model. A standard error function such as cross-entropy between the output of the RNN and the target output can be used.

Since the error function and all non-linear functions used in the RNN are differentiable, training is done using stochastic gradient descent. The result of training is a learned value for all matrices in the RNN, which together form a part of our model parameter $\mathbf{M}$.

During inference, we are given a value Z of Z and a particular $Y = s_1, \ldots, s_n$, and would like to compute $P(Y = Y|Z = Z, \mathbf{M})$. This is straightforward: we set x_t as the one-hot vector of s_t for $1 \le t \le n$. Then, $P(Y = Y|Z = Z, \mathbf{M}) = \prod_{t=1}^{n-1} y_t(s_{t+1})$ where y_t is computed using Equation 6.

4.4 Estimation of the Anomaly Score

There are two difficulties associated with computing the anomaly score in our instantiation of the framework. First, in general, the computation in Equation 4 requires summing over a possibly infinite number of program behaviors Y, which is not feasible. Second, it also requires computing $P(Y|X = X_F, \mathbf{M})$, which in turn requires integrating out the unknown specification Z (Equation 2).

Both of these difficulties can be addressed via sampling. We note that in general, to estimate a summation of the form $\sum_{i \in I} P_1(i)P_2(i)$ where $P_1(i)$ is a probability mass function over the (possibly) infinite domain I and P_2 is a function on I, it suffices to take a number of samples $i_1, i_2, \ldots, i_m \sim P_1(i)$. One can then use:

$$\sum_{i \in I} P_1(i)P_2(i) \approx \sum_{k=1}^{m} \frac{1}{m} P_2(i_k)$$

as an unbiased estimate for the desired sum. It is well known from standard sampling theory that the variance of this estimator, denoted as σ^2, reduces linearly as m increases.

We can apply this process to estimate the anomaly score for F by letting the domain I be the set of all possible behaviors in Σ^*, and sampling a large number of behaviors Y with probability proportional to $P_{\mathsf{F}}(\mathsf{Y})$, then letting $P_2(\mathsf{Y}) = \log(P_{\mathsf{F}}(Y = \mathsf{Y})) - \log(P(Y = \mathsf{Y}|X = \mathsf{X_F}, \mathbf{M}))$ and using the estimator described above. We can keep sampling until the variance of the estimate is sufficiently small.

Fortunately, sampling a behavior from the distribution $P_{\mathsf{F}}(Y)$ is easy: we can use rejection sampling [51] to sample an accepting run π of F and then simply obtain its behavior $\mathsf{Y} = [\![\pi]\!]$. However we do not yet have a complete solution to our problem. The difficulty is that for a sampled behavior Y, it is not possible to compute $P_2(Y = \mathsf{Y})$ easily because of two reasons. First, the term $P_{\mathsf{F}}(Y = \mathsf{Y})$ (Equation 5) requires summing over possibly infinite number of accepting runs Π_{F}, and second, as mentioned before, computing $P(Y = \mathsf{Y}|X = \mathsf{X_F}, \mathbf{M})$ requires integrating over the unseen Z value.

To handle this, we extend our sampling-based algorithm. Rather than just sampling behaviors, we sample the set I of $(\mathsf{Y}, \widetilde{\Pi_{\mathsf{F}}}, \widetilde{Z_{\mathsf{F}}})$ triples, where $\widetilde{\Pi_{\mathsf{F}}}$ is itself a set of accepting runs of F sampled using the same method, and $\widetilde{Z_{\mathsf{F}}}$ is a set of values sampled from $P(Z|X = \mathsf{X}_F, \mathbf{M})$. The latter set of samples can easily be obtained via Gibbs sampling. One could then estimate the divergence as:

$$\frac{1}{|I|} \sum_{\substack{(\mathsf{Y}, \widetilde{\Pi_{\mathsf{F}}}, \widetilde{Z_{\mathsf{F}}}) \\ \in I}} \log\left(\sum_{\pi \in \widetilde{\Pi_{\mathsf{F}}}(\mathsf{Y})} \frac{1}{|\widetilde{\Pi_{\mathsf{F}}}|} \right) - \log\left(\sum_{\mathsf{Z} \in \widetilde{Z_{\mathsf{F}}}} P(Y = \mathsf{Y}|Z = \mathsf{Z}, \mathbf{M}) \right)$$

where $\widetilde{\Pi_{\mathsf{F}}}(\mathsf{Y})$ is the set of paths $\pi \in \widetilde{\Pi_{\mathsf{F}}}$ such that $[\![\pi]\!] = \mathsf{Y}$. The sum in the first log term estimates the fraction of sampled accepting runs whose behavior is Y, thus estimating $P_{\mathsf{F}}(Y = \mathsf{Y})$, and the sum in the second log term estimates $P(Y = \mathsf{Y}|X = \mathsf{X_F}, \mathbf{M})$.

The problem is that this estimate will be biased, since one cannot commute the expectation operator $\mathbf{E}$ with a logarithm. That is:

$$\mathbf{E}\left[\log(\sum_{\pi \in \widetilde{\Pi_{\mathsf{F}}}(\mathsf{Y})} \frac{1}{|\widetilde{\Pi_{\mathsf{F}}}|}) \right] \neq \log(\mathbf{E}\left[\sum_{\pi \in \widetilde{\Pi_{\mathsf{F}}}(\mathsf{Y})} \frac{1}{|\widetilde{\Pi_{\mathsf{F}}}|} \right])$$

A similar problem exists for the second summation used to estimate the logarithm of $P(Y = \mathsf{Y}|X = \mathsf{X_F}, \mathbf{M})$. Intuitively, this bias is not surprising, since an over-estimate for the probability $P_{\mathsf{F}}(Y = \mathsf{Y})$ by some constant amount is likely to have little effect on an estimate of the logarithm of the probability. However, an under-estimate by the same amount can cause a radical reduction in the estimate of the logarithm, and we expect a negative bias.

A sampling-based estimate for this bias can be computed using a Taylor series expansion about the expected value of the biased estimator, which obtains an expression for the bias in terms of the central moments of a Normal distribution; estimating those moments leads to an estimate for the bias. Assume that this estimator is encapsulated in a procedure $bias(\mathsf{Y}, \widetilde{\Pi_{\mathsf{F}}}, \widetilde{Z_{\mathsf{F}}})$ that computes the bias of an estimate. Our final estimate for the anomaly score is:

$$\frac{1}{|I|} \sum_{(\mathsf{Y}, \widetilde{\Pi_{\mathsf{F}}}, \widetilde{Z_{\mathsf{F}}}) \in I} \log\left(\sum_{\pi \in \widetilde{\Pi_{\mathsf{F}}}(\mathsf{Y})} \frac{1}{|\widetilde{\Pi_{\mathsf{F}}}|} \right) - \log\left(\sum_{\widetilde{Z} \in \widetilde{Z_{\mathsf{F}}}} P(Y = \mathsf{Y}|Z = \mathsf{Z}, \mathbf{M}) \right) - bias(\mathsf{Y}, \widetilde{\Pi_{\mathsf{F}}}, \widetilde{Z_{\mathsf{F}}}).$$

Topic 1
```
A.setMessage(int)
A.setTitle(int)
new A(Context)
```

Topic 2
```
A.setPositiveButton(String,...)
A.setNegativeButton(String,...)
A.setMessage(String)
```

Topic 3
```
A.setView(View)
new A(Context)
A.setTitle(String)
```

Topic 4
```
A.setItems(String[],...)
A.setNeutralButton(int,...)
A.show()
```

Topic 5
```
C.getInstance(String)
C.init(int,Key,...)
C.doFinal(byte[])
```

Topic 6
```
B.connect()
B.getInputStream()
B.getOutputStream()
```

Figure 4: Top-3 methods from topics extracted by LDA (`A` = `AlertDialog.Builder`, `B` = `BluetoothSocket`, `C` = `Cipher`)

5 EVALUATION

In this section, we present an evaluation of our method on the task of finding API misuses in Android apps. Specifically, we seek to answer the following questions:

(1) Can we find useful *de facto* specifications followed by Android developers (Section 5.2)?
(2) Using the specifications, can we find possible bugs in the usage of the Android API in a corpus (Section 5.3)?
(3) How does specification learning help in anomaly detection (Section 5.4)?
(4) How does the Bayesian framework help in handling heterogeneity in the specifications (Section 5.5)?

5.1 Implementation and Experimental Setup

Now we briefly describe the system, Salento, that implements our method. Salento uses Soot [50] to implement symbolic execution and transform code in an Android app into our automaton model, TensorFlow [3] to implement the topic-conditioned RNN, and Scipy [30] to implement LDA. Salento builds a coarse model of the Android app life-cycle by collecting all entry points in the application which are callback methods from the Android kernel. It also uses Soot's Class Hierarchy Analysis and Throw Analysis to overapproximate the set of possible call or exception targets, and Soot's built-in constant propagator to detect infeasible paths.

In addition to API methods in Σ, Salento also collects some semantic information about the state of the program when an API call is made. This is done through the use of simple Boolean *predicates* that capture, for example, constraints on the arguments of a call, or record whether an exception was thrown by the call. This allows us to learn specification on more complex programming constructs.

The training corpus consisted of 500 Android apps from [1], and the testing corpus consisted of 250 apps from [2]. The two repositories did not overlap, perhaps since the latter is open-source and the former is not. We conducted experiments on three Android APIs: alert dialogs (`android.app.AlertDialog.Builder`), bluetooth sockets (`android.bluetooth.BluetoothSocket`) and cryptographic ciphers (`javax.crypto.Cipher`). The APIs were chosen to represent common yet varied facets of a typical Android app (UI, functionality, security). From the training and testing repositories, we created

about 6000 and 1800 automata models (henceforth just called programs) respectively. While doing so, we set the accepting location of the program as various locations of interest, that is, locations where a method in one of these APIs was invoked. This helps in *localizing* an anomaly to a particular location. All experiments were run on a 24-core 2.2 GHz machine with 64 GB of memory and an Nvidia Quadro M2000 GPU.

5.2 Specification Learning

With a goal to discover specifications of Android API usage, we applied LDA on the training corpus of programs, where the alphabet Σ consisted of 25 methods from the three APIs. We used $\alpha = 0.1$ for each topic, and $\eta = 1/|\Sigma|$ for all words in a topic. Running LDA with 15 topics (K) took a few seconds to complete. Figure 4 shows the top-3 words (methods) from six topics extracted from the corpus that we picked to exemplify. At a first glance, it may seem that LDA is simply categorizing methods from different APIs into separate topics, which can raise the question of why we need topic models if we already knew the APIs beforehand.

LDA, however, does more than that. Topic 1 and Topic 2 contain methods from the same API but, interestingly, different polymorphic versions with `int` and `String` arguments. The model has discovered that the polymorphic versions fall under separate topics, meaning that they are *not often used together in practice*. Indeed, some Android apps declare all resources they need in a separate XML file, and provide the resource ID as the `int` argument. Other apps do not make use of this feature and instead directly provide the string to use in the dialog box. Therefore, it makes sense that an app would seldom use both versions together. Similarly, Topic 3 also contains methods from the same API, however it describes yet another way to create dialog boxes. Note the lack of the `setMessage` method in this topic, as the message would already have been enveloped in the `View` passed to `setView` (using both methods together can lead to the display of corrupted dialog boxes as shown in Figure 2(a)(ii)).

As these examples show, the topic model can expose specifications of how methods in an API (or different APIs) are used together.

5.3 Anomaly and Bug Detection

To evaluate Salento on anomaly detection, we first trained the topic-conditioned RNN on 60,000 behaviors sampled from the training programs. Training took 20 minutes to complete. We then computed anomaly scores for the 1800 programs in our testing corpus. The time to compute each score was around 2-3 seconds.

The histogram of scores, in Figure 5(a), shows a high concentration of small values, such as 5 or less, and a very low concentration of high values. We chose to further investigate programs appearing in the top 10% of anomaly scores (above the red line) for possible bugs. Specifically, since each program provides a localization to a location in the app (through its accepting states), we investigated the behaviors that were sampled from the program's probabilistic behavior model, that would have determined its anomaly score.

Our definition of a "possible bug" is based on the following: is a behavior an instance of Android API usage that is questionable enough that we would expect it to be raised as an issue in a formal code review? Note that an issue raised in a code review may relate to a design choice and not necessarily cause the program to crash (an unusual button text, for example). Nonetheless, such an issue would be raised and likely fixed by engineers examining a code.

One problem with counting an anomaly as a possible bug is that multiple anomalies in an app can have the same "cause"—an incorrect statement or set of statements in the code—and we would like to avoid "double-counting" different anomalies with the same cause as different bugs. It is a hard software engineering problem to establish the cause of an anomaly/bug, which is out of scope of this paper. To avoid this problem, however, we conservatively consider only the top-most anomaly in each app in the top-10%, as clearly, anomalies in two different apps cannot have the same cause.

Through manual inspection and triage by the first author, we found 10 different types of possible bugs in our testing corpus (Figure 6), ranging from the benign to the insidious. We have already seen anomalies #6 and #10 (Figure 2) that could display corrupted or unclosable dialog boxes. #2 could lead to an exception being thrown due to a failed connection, #5 would create a crypto object that defaults to the semantically insecure ECB-encryption mode, and #8 could cause future attempts to open a socket to be blocked.

Figure 5(b)(i) shows the precision-recall plot for these possible bugs in the top-10% of anomaly scores. It can be seen that at around the top 8%, we reach full recall with 75% precision or 25% false positive rate. This is reasonable compared to industrial static analysis tools such as Coverity that advocates a 20% false positive rate for "stable" checkers [14]. Our method does not rely on specified properties to check, and many of these bugs cannot be easily expressed as a formal property for traditional static analyzers to check.

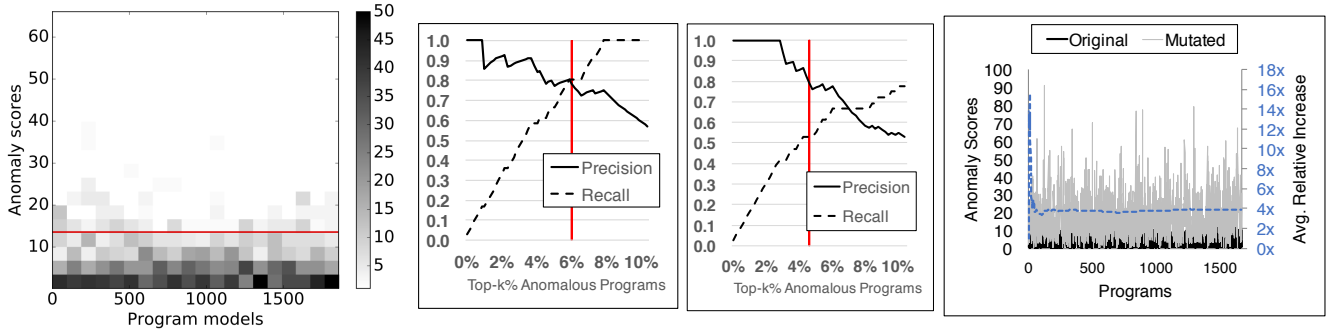
After this threshold, the precision continues to drop, and we conjecture that it will not increase any further, because almost all the possible bugs have already been found. To substantiate this conjecture, we would have to manually inspect thousands of programs to qualitatively declare that all anomalies have been triaged. Due to the practical infeasibility of this task, we instead quantitatively *injected* anomalies into the remaining 90% of programs through mutations, and measured whether our model is able to detect those mutations. For each program, we mutated the API call before its accepting states into one chosen randomly from Σ.

Figure 5(c) shows the anomaly scores before (dark) and after (light) the mutation, and the cumulative mean of the relative increase in the score (dashed line, secondary axis). As a result of the mutation, the scores are greatly increased, sometimes by 20 times or more, and the mean of the increase is about 4x. That is, a mutation, on average, caused the anomaly score to increase by 4 times, indicating that our model detected the mutation.

Note that a random mutation has the possibility of *reducing* the anomaly score of a program if it had a possible bug and the mutation happened to fix it. However, it is not very likely for a random mutation to fix a bug, and so these instances rarely occurred.

5.4 Role of Learning in Anomaly Detection

To evaluate the role of learning, we compared with a traditional outlier detection method that does not require learning. *k-nearest neighbor* (k-NN) outlier detection [5] uses a distance measure to compute the k nearest neighbors of a given point within a dataset. The larger the average distance to the k-NN, the more likely it is that the point is an outlier, or anomaly. We already have a distance



(a) (b)(i) (b)(ii) (c)

Figure 5: (a) Histogram of anomaly score values, (b) Precision-recall for the possible bugs in Figure 6 for (i) Bayesian model (ii) non-Bayesian model, and (c) Anomaly scores of remaining 90% programs before and after mutation

#	Count	Avg Score	Anomaly
1	2	43.7	C Single crypto object used to encrypt/decrypt multiple data
2	1	37.5	B Connecting to the same socket more than once
3	1	24.7	B Attempt to close unopened socket
4	16	22.1	A Using `String` and `int` polymorphic methods together
5	6	21.8	C Crypto object created without specifying mode
6	6	21.6	A Using `setMessage` with `setView`
7	1	19.8	A Dialog displayed without message
8	1	19.3	B Failed socket connection left unclosed
9	1	16.5	A Unusual button text
10	1	15.7	A Dialog displayed without buttons

Figure 6: Anomalies that are possible bugs, found in the top 10% of anomalous programs

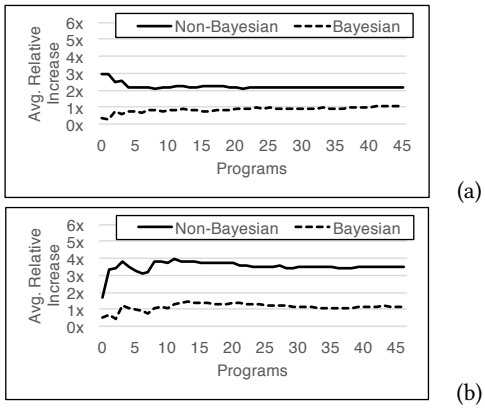
measure between distributions: the KL-divergence between the behavior model for the given program and a program in the corpus.

We implemented such a k-NN and compared our method with it by conservatively setting $k = 1$. That is, the anomaly score of a given program is the *smallest* KL-divergence with any program in the corpus. However, even with this 1-NN anomaly score, a substantial top 25% of programs had a distance of infinity to the corpus, thus providing no useful information about their anomalies.

The reason is that these programs happened to generate a behavior that was not generated by *any* program in the corpus. This sets $P_1(Y)$ to a non-zero value and $P_2(Y)$ to zero in the KL-divergence formula (Equation 3) immediately making the sum infinity. This is unreasonable because we clearly do not want to call every behavior we have not observed in the training data an anomaly, but instead would like to assign probabilities even to behaviors that were never seen before. That is, we would like to *generalize* from the corpus. This is why probabilistic specification learning is needed.

5.5 Comparison with Non-Bayesian Methods

To see how the Bayesian framework helps in handling heterogeneity in the corpus, we compared our method with a non-Bayesian specification learning method. Existing state-of-the-art methods use n-grams [52] or RNNs [47] to learn a (non-Bayesian) single



(a)

(b)

Figure 7: Average relative increase in anomaly scores of `BluetoothSocket` programs when the training corpus only uses the APIs (a) `BluetoothSocket`, `Cipher` (b) `AlertDialog.Builder`, `BluetoothSocket`, `Cipher`

probabilistic specification of program behaviors. We implemented a non-Bayesian specification learner as an RNN (*not* topic conditioned) and trained it directly on the behaviors in our training corpus. We then performed the same anomaly and bug detection experiment in Section 5.3, querying the trained model with behaviors in the testing program for inference.

Figure 5(b)(ii) shows the precision-recall rate for the top-10% of anomaly scores. Compared to our Bayesian method, the non-Bayesian method fared poorly. Consider again a "stable" checker's false-positive rate of 20%, or 80% precision. At this threshold (marked by the red line), our Bayesian method has about 80% recall compared to only 53% for the non-Bayesian method. This shows that given a reasonable precision threshold, our method is able to discover significantly more bugs compared to the non-Bayesian method. It is also worth noting that the non-Bayesian method was unable to discover any possible bug that was not found by our method.

Effect of Heterogeneity. We finally performed a series of experiments by incrementally increasing the heterogeneity of the training programs. First, as a baseline, we considered only programs that use the `BluetoothSocket` API, and learned from them both Bayesian and non-Bayesian specifications of their behaviors. We then computed anomaly scores of the 45 testing programs that use this API.

In the next step, we added to the training corpus programs that also use the `Cipher` API, making the corpus more heterogeneous, and learned new specifications. We then computed anomaly scores again, but using the new learned specifications. Figure 7(a) shows the average relative increase in anomaly scores from using the old versus the new specifications. Ideally, one would expect the scores to not change, because the addition of programs that use the `Cipher` API—behaviors on which are unrelated to the `BluetoothSocket` API— should not have any effect on the scores. This is observed in the Bayesian specification (dashed line), that lingers close to 1.0 on average. However, the non-Bayesian specification (solid line) suffers from about a 2x increase.

This was further evident when programs that also use the API `AlertDialog.Builder` were considered for training, making the corpus even more heterogeneous (this is the same training corpus in Section 5.3). In Figure 7(b), the relative increase in scores using the Bayesian specification is, on average, close to 1.0, showing that it is robust to the increased heterogeneity. However, the non-Bayesian specification induces a further increase of about 3.5x in the scores.

We expect the gap to keep widening as more heterogeneous programs are added to the corpus, at some point making the scores from the non-Bayesian model meaningless. In contrast, the scores from our Bayesian model would remain almost the same showing that the model is able to "focus" on relevant parts of the learned specification, in principle tolerating arbitrary heterogeneity.

5.6 Limitations

We conclude by summarizing the limitations of our experiments:

1. Our evaluation used Android, a platform where programs are APIs-heavy and APIs are fairly well-structured. While our experiments show good results in this domain, whether they generalize to other domains that do not share these characteristics (such as C programs) is an open question.

2. Our evaluation used a small subset of the Android API space, due to the manual effort needed to report precision-recall numbers. It is possible for the results to be different for a different set of APIs.

3. Finally, as the domains that we study often lack crisp definitions of correctness, the first author of this paper manually triaged the anomalies reported in our experiments into true and false positives. While this step was performed carefully, it is possible that a different person could have triaged some of these reports differently.

6 RELATED WORK

Learning Qualitative Specifications. The thesis that common patterns of execution can serve as a proxy for specifications has been around since the early 2000s. Most efforts in this area [6–8, 21, 25, 35, 42, 48, 55–57] focus on qualitative specifications, typically finite automata. As mentioned earlier, such qualitative specifications are problematic in the presence of noise in the training data.

Learning Probabilistic Specifications. There is also a literature [7, 12, 27, 33, 36, 43, 47] on learning probabilistic specifications from programs. Kremenek et al. [32, 33] use factor graphs constructed using static analysis to learn specifications on resource allocation and release. Anek [12] uses annotations in APIs to infer specifications. Merlin [36] starts with a given initial specification and refines it through factor graph construction and inference. Octeau et al. [43] use domain knowledge to train probabilistic models of Android inter-component communication. JSNice [46] uses a probabilistic graphical model to learn lexical and syntactical properties of programs such as variable names and types for the purpose of de-obfuscating Javascript programs. Some recent efforts [40, 41, 52] have also used *n-gram* models to learn specifications on source code structure. DeepAPI [26] uses a neural encoder-decoder to learn correlations between natural language annotations and API sequences. Haggis [4] uses statistical techniques to learn the structure of small code snippets (or "idioms") from a corpus. The work in this space that is the closest to ours are two methods by Raychev et al. [45, 47], which learn probabilistic models of program behavior from large code corpora, using RNNs among other models.

The key difference between the above approaches and ours is that these methods learn a *single* probabilistic specification. In contrast, our approach learns a *family* of probabilistic specifications simultaneously, and then specializes this model to particular analysis tasks using Bayes' rule. As demonstrated in our experiments, this hierarchical architecture is key to tolerating heterogeneity.

In very recent work, Raychev et al. [44], also argue that having a single, universal probabilistic model for code can be inadequate, and propose a decision tree algorithm that is used to choose among a bag of statistical models for tasks such as next-statement prediction. While philosophically aligned with our work, their efforts are quite different in that while we argue for conditioning of models at the *program level*, they argue for conditioning of models at the *statement level* and focus their efforts on localized prediction tasks. One could imagine using a model similar to what they have proposed within our framework as a replacement for our RNN-based $P(Y|Z)$.

Anomaly Detection. There is prior work on using learned models of executions in anomaly detection [10, 19, 22, 23, 28, 38, 52, 54]. Aside from differing in the nature of specifications used, methods in these categories tend to assign anomaly scores to individual behaviors. While our method is able to assign such scores, it is also able to produce aggregate anomaly scores for programs.

7 CONCLUSION

We have presented a Bayesian framework that can learn probabilistic specifications from large, heterogeneous code corpora and then use these specifications to find likely software errors. We have used an implementation of this framework, based on a topic-model and a recurrent neural network, to detect API misuse in Android, and shown that it can find multiple subtle bugs.

A key appeal of our framework is that it does not impose an a priori limit on the size or heterogeneity of the corpus. In principle, our training corpus could contain all the world's code, and it is our vision to scale our method to settings close to this ideal. Engineering instantiations of the framework that work at such scale is a challenge for future work.

ACKNOWLEDGEMENT

This research was supported by DARPA MUSE award #FA8750-14-2-0270.

REFERENCES

[1] Androiddrawer. http://www.androiddrawer.com. [Online; accessed 06-Jul-2016].

[2] F-Droid. https://f-droid.org. [Online; accessed 06-Jul-2016].

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. ManÃĺ, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. ViÃĺgas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[4] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM.

[5] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[6] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[7] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[8] G. Ammons, D. Mandelin, and J. R. Larus. Debugging temporal specifications with concept analysis. In *In ACM SIGPLAN Conf on Prog Lang Design and Implem*, pages 182–195. ACM Press, 2003.

[9] S. Anand, C. S. Păsăreanu, and W. Visser. Jpf–se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[10] G. K. Baah, A. Gray, and M. J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 70–77. ACM, 2006.

[11] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[12] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. *SIGPLAN Not.*, 46(6):211–221, 2011.

[13] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *journal of machine learning research*, 3(Feb):1137–1155, 2003.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[15] P. Bielik, V. Raychev, and M. Vechev. Programming with "big code": Lessons, techniques and applications. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[16] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[17] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[18] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[19] T. M. Chilimbi and V. Ganapathy. Heapmd: Identifying heap-based bugs using anomaly detection. In *ASPLOS*, volume 34, pages 219–228. ACM, 2006.

[20] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.

[22] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, volume 9, pages 149–158, 2009.

[23] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 15. ACM, 2007.

[24] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, Nov. 1984.

[25] C. Goues and W. Weimer. Specification mining with few false positives. In *TACAS*, pages 292–306, 2009.

[26] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. *CoRR*, abs/1605.08535, 2016.

[27] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. *SIGPLAN Not.*, 48(6):27–38, June 2013.

[28] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[29] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: A symbolic execution tool for verification. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 758–766, Berlin, Heidelberg, 2012. Springer-Verlag.

[30] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-11-09].

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[32] T. Kremenek, A. Y. Ng, and D. Engler. A factor graph model for software bug finding. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2510–2516, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association.

[34] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[35] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.

[36] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 75–86, 2009.

[37] T. Mikolov and G. Zweig. Context dependent recurrent neural network language model. In *SLT*, pages 234–239, 2012.

[38] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 2–25, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] A. S. Murawski and J. Ouaknine. Concur 2005 - concurrency theory. chapter On Probabilistic Program Equivalence and Refinement, pages 156–170. Springer-Verlag, London, UK, UK, 2005.

[40] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[41] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[42] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[43] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 469–484, 2016.

[44] V. Raychev, P. Bielik, and M. T. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 731–747, 2016.

[45] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016.

[46] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *POPL*, volume 50, pages 111–124, 2015.

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

[48] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[49] A. Sokolova and E. P. de Vink. *Probabilistic Automata: System Types, Parallel Composition and Comparison*, pages 1–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[50] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[51] J. Von Neumann. 13. various techniques used in connection with random digits. 1951.

[52] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International*

Conference on Automated Software Engineering, ASE 2016, pages 708–719, New York, NY, USA, 2016. ACM.

[53] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 708–719, 2016.

[54] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[55] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, 2005.

[56] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.

[57] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, 2009.