AFRL-RI-RS-TR-2020-189

PLINY: AN END-TO-END FRAMEWORK FOR BIG CODE ANALYTICS

WILLIAM MARSH RICE UNIVERSITY

OCTOBER 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-189 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
GREGORY HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
OCTOBER 2020	FINAL TECHNICAL REPORT	SEP 2014 – DEC 2019

4. TITLE AND SUBTITLE

PLINY: AN END-TO-END FRAMEWORK FOR BIG CODE ANALYTICS

5a. CONTRACT NUMBER
FA8750-14-2-0270

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Vivek Sarkar

5d. PROJECT NUMBER
MUSE

5e. TASK NUMBER
DR

5f. WORK UNIT NUMBER
IC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
William Marsh Rice University
6100 Main Street
Houston TX 77005-1827

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-189

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The PLINY project has addressed the grand challenge in the Defense Advanced Research Projects Agency program on Mining and Understanding Software Enclaves (MUSE) by advancing three key technical areas in the program: Artifact Generators, Mining Engine, and Analytics. The PLINY project has introduced technologies that leverage the institutional knowledge embodied in the vast corpus of existing software to simplify the creation of new software. The main advances of the PLINY project have been in the following areas: Code search (Source Forager), Anomaly detection (Salento), Test-based synthesis (Splicer, SyPet, and Hunter), Evidence-based synthesis (Bayou), and Scalable infrastructure for search and learning (PlinyCompute). The PLINY technologies extend the reasoning capabilities of the programmer through the use of automated analysis and synthesis tools. The technologies developed in the PLINY project help address many of the productivity and cost challenges associated with software development today, and will also reshape the way people think about programming in the future.

15. SUBJECT TERMS
Anomaly Detection, API calls, Bayesian Networks, Code Search, Code Synthesis, Distributed Computing, Machine Learning, Program Analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	60	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1 Summary

The Defense Advanced Research Projects Agency's (DARPA) program on "Mining and Understanding Software Enclaves" (MUSE) put forth a grand challenge to leverage the institutional knowledge embedded in millions of lines of existing code corpora ("big code") to revolutionize the field of software engineering by transforming the way software is developed. The PLINY project addressed this grand challenge by advancing three key technical areas in the MUSE program — Technical Area (TA) 2: Artifact Generators, TA 3: Mining Engine, and TA 4: Analytics. The results of our research have been published in over 40 peer-reviewed publications [1–40] and multiple open-source software releases. As described in these publications, the main advances have been in the following areas:

1. *Code search:* We developed a new code-search engine named Source Forager [41]. Given a query in the form of a C/C++ function, Source Forager searches for similar C/C++ functions using a pre-populated code database generated from available code corpora. Source Forager preprocesses the database to extract a unique set of syntactic and semantic code features that capture different aspects of code. A search returns the top-k functions in the database that are most similar to the query and allows for assigning different weights to different syntactic/semantic code features. Our experiments show that the ranked results returned by Source Forager are accurate and that query-relevant functions can be reliably retrieved even when searching through a large code database that contains very few query-relevant functions.
2. *Anomaly detection:* We developed a novel Bayesian framework [21] that can learn probabilistic specifications from large, unstructured code corpora, and then use these specifications to detect anomalous, hence likely buggy, regions of code. This approach has been implemented in an open-source system, called SALENTO [42], for finding application programming interface (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.
3. *Test-based synthesis:* Our early experiences with code search led to creating an approach to automatic code synthesis in which a) the user provides an incomplete sketch of a program, b) available code corpora are searched to return functions that are most similar to the input, c) formal methods are used to generate candidate complete executable programs from the sketch and search results, and d) user-provided test cases are used to prune the candidate set to those that pass all tests. We developed three systems to demonstrate this approach — Splicer, SyPet and Hunter.

   With Splicer [12], the programmer provides a sketch that includes incomplete code, natural language comments, and correctness requirements. A program synthesizer that interacts with a

large, searchable database of program snippets then automatically completes the sketch into a program that meets the requirements. Splicer was implemented for the Java programming language, using a code corpus of over 3.5 million Java methods.

SyPet [28] is a program synthesis tool that helps programmers to use Java libraries. The programmer provides SyPet with: (1) a signature of the method to be synthesized, (2) a set of test cases, and (3) a set of Java libraries. SyPet will automatically find a sequence of API calls from these Java libraries that will pass all test cases provided by the programmer. The key novelty in SyPet is the use of a compact Petri-net representation to model relationships between methods in an API.

Hunter [30] is a tool that facilitates code reuse by finding relevant methods in large codebases and automatically synthesizing any necessary wrapper code for code adaptation. We have implemented Hunter as an Eclipse plug-in and it is available on Eclipse marketplace [43].

4. *Evidence-based synthesis:* To further reduce the burden on developers for providing test inputs and sketch programs, we also explored an alternate approach to code synthesis [7] that significantly reduces the input that a user needs to provide to a small amount of "evidence". Examples of evidence include a small set of API calls and/or data types that are desired in the generated code. This approach is unique in that it trains a neural generator on program sketches rather than on complete source codes, and synthesizes code by sampling a posterior distribution over sketches and then concretizing samples from this distribution into type-safe programs. This approach was implemented in an open-source system named BAYOU [44] for generating API-heavy Java code, which demonstrated how the entire body of a method can be predicted given just a few API calls or data types that are desired in the method.
5. *Scalable infrastructure for search and learning:* The above techniques for code search, anomaly detection, and code synthesis all rely on the ability to perform both scalable search and scalable learning on large code corpora. Since no currently available system can perform both kinds of operations in a scalable manner, we developed the open-source PLINYCompute system [1, 2, 10, 45]. PLINYCompute supports the development of high-performance, data-intensive, distributed computing tools and libraries, making it especially well suited for scalable search and learning applications. It performs automatic, relational-database style optimizations on declarative commands to determine how best to stage distributed computations. This capability is enabled by a persistent object data model and associated memory management system designed specifically for high performance, distributed, data-intensive computing, resulting in superior performance and scalability relative to distributed systems built on managed runtimes such as Java Virtual Machines (JVMs).

6. *Extension project on identifying API misuse in JavaScript code:* In this extension project, we performed a transition-related task mentored by GitHub as a potential transition partner[1]. Specifically, we focused on scanning JavaScript programs for API misuse by leveraging our earlier experiences with using big code for anomaly detection. While our past work on anomaly detection was prototyped on Java and C/C++ code, the extension project focused on Javascript, since it has a higher transition priority as the most popular programming language used in GitHub projects. We also prototyped our approach in JS-Smart, a pipeline that performs code mining and anomaly detection for JavaScript codes as described in more detail later in this report.

[1] Note that our research is open and available to any industry partner interested in these capabilities.

2 Introduction

Inspired by the goals of the DARPA MUSE program, the PLINY project has introduced technologies that simultaneously bring the vast experience of expert programmers and the rigor of automated tools to the fingertips of ordinary software developers. In this manner, PLINY leverages the institutional knowledge embodied in the vast corpus of existing software to simplify the creation of new software. Furthermore, the PLINY technologies extend the reasoning capabilities of the programmer through the use of automated analysis and synthesis tools. We believe that the technologies developed in the PLINY project will help address many of the productivity and cost challenges associated with software development today, and also reshape the way people think about programming in the future.

From the perspective of a PLINY user, programming is no longer a solitary exercise where the only sources of help are the debugger and the occasional helpful posts on websites like Stack Overflow. PLINY instead interacts with the programmer to complete program sketches, find bugs, and suggest useful fixes. The programmer will no longer need to spend unproductive hours trying to understand the undocumented intricacies of a complex library, but can instead ask PLINY to help fill in boilerplate code and identify inconsistencies in API usage. These capabilities will not only help novice programmers but will also dramatically increase the productivity of expert software developers. Figure 1 summarizes this overall PLINY vision, which to a software developer can appear as "magic in the cloud" that uses the knowledge in millions of software repositories to enable advanced Code Search, Anomaly Detection, and Code Synthesis, all powered by Scalable Infrastructure in the cloud.

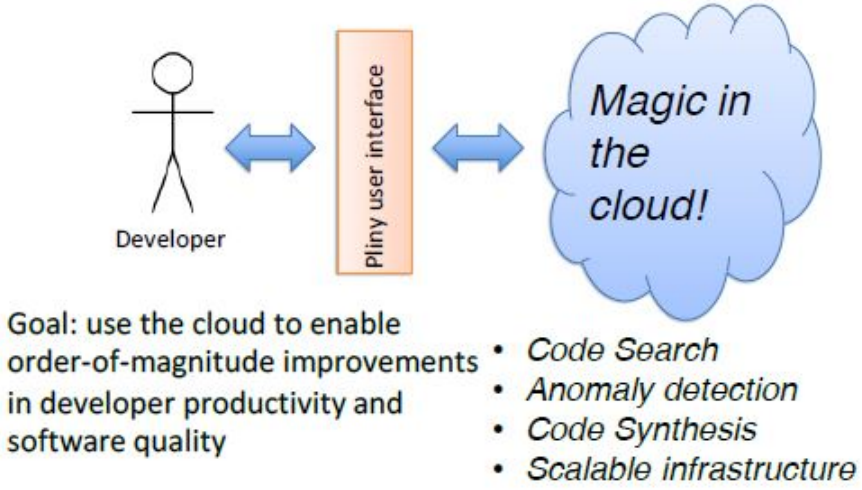

Figure 1: The PLINY vision.

The PLINY project realized the vision in Figure 1 by advancing three key technical areas in the MUSE program — TA 2: Artifact Generators, TA 3: Mining Engine, and TA 4: Analytics — as indicated in Figure 2. To enable the "magic in the cloud" the PLINY system crawls large software

repositories ("code corpora") asynchronously in batch mode, similar to web crawlers used by search engines. However, unlike search engines, the artifact generation encompassed by TA 2 includes feature extraction from source code that is driven by a sophisticated program analyses in the PLINY Language Framework. Our work related to TA 3 is embodied in the PLINYCompute engine which supports both code search and machine learning (ML) on code artifacts with high scalability. Finally, TA 4 represents the part of the PLINY system that interfaces with developers at all expertise levels by leveraging the PLINY Language Framework, the PLINY Reasoning Framework, and the PLINYCompute engine, to enable code search, anomaly detection, and code synthesis capabilities that are central to enhancing developer productivity.

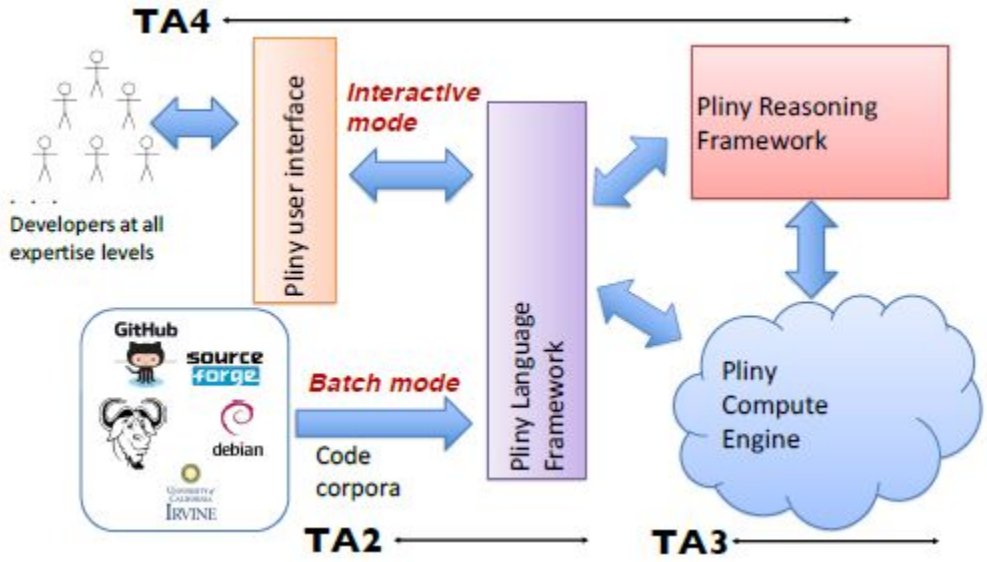

Figure 2: Realizing the PLINY vision.

In the remainder of this report, we elaborate on the methods, assumptions, and procedures (Section 3) and present and discuss results (Section 4) for key components of the PLINY project.

The results of our research have been disseminated in multiple software releases and 40 peer-reviewed publications [1–40] presented at the following top-tier venues:

International Conference on Computer-Aided Verification (CAV) 2018;

European Symposium on Programming (ESOP) 2016;

Association for Computing Machinery (ACM) Special Interest Group on Software Engineering (SIGSOFT)/Symposium on Foundations of Software Engineering (FSE) 2016

ACM SIGSOFT Joint Meeting on European Software Engineering Conference (ESEC) and Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE) 2017, 2018;

International Conference on Learning Representations (ICLR) 2018;

International Conference on Machine Learning (ICML) 2018;

Institute of Electrical and Electronics Engineers (IEEE) International Conference on Software Engineering (ICSE) 2018;

The Network and Distributed System Security Symposium (NDSS) 2017;

Object Oriented Programming, Systems, Languages and Applications (OOPSLA) 2015, 2017, 2018;

ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Programming Language Design and Implementation (PLDI) 2015, 2016, 2017, 2018.

ACM SIGPLAN Special Interest Group on Algorithms and Computation Theory (SIGACT) Symposium on Principles of Programming Languages (POPL) 2016, 2017, 2018;

Proceedings of the Very Large Data Bases (VLDB) Endowment (PVLDB) 2019;

ACM Special Interest Group on Management of Data (SIGMOD)/Principles of Database Systems (PODS) International Conference on Management of Data 2018;

ACM Transactions on Programming Languages and Systems (TOPLAS);

International Conference on VLDB 2018; and

International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2016, 2017.

3 Methods, Assumptions and Procedures

This section includes details on three key components of the PLINY project:

1. *Anomaly detection*: We developed a novel Bayesian framework [21] that can learn probabilistic specifications from large, unstructured code corpora, and then use these specifications to detect anomalous, hence likely buggy, regions of code. This approach has been implemented in an open-source system, called SALENTO [42], 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.
2. *Evidence-based Synthesis*: To further reduce the burden on developers for providing test inputs and sketch programs, we also explored an alternate approach to code synthesis [7] that significantly reduces the input that a user needs to provide to a small amount of "evidence". Examples of evidence include a small set of API calls and/or data types that are desired in the generated code. This approach is unique in that it trains a neural generator on program sketches rather than on complete source codes, and synthesizes code by sampling a posterior distribution over sketches and then concretizing samples from this distribution into type-safe programs. This approach was implemented in an open-source system named BAYOU [44] for generating API-heavy Java code, which demonstrated how the entire body of a method can be predicted given just a few API calls or data types that are desired in the method.
3. *Extension project on identifying API misuse in JavaScript code*: In this extension project, we performed a transition-related task mentored by GitHub as a potential transition partner[2]. Specifically, we focused on scanning JavaScript programs for API misuse by leveraging our earlier experiences with using big code for anomaly detection. While our past work on anomaly detection was prototyped on Java and C/C++ code, the extension project focused on Javascript, since it has a higher transition priority as the most popular programming language used in GitHub projects. We also prototyped our approach in JS-Smart, a pipeline that performs code mining and anomaly detection for JavaScript codes as described in more detail later in this report.

Additional aspects of our project are described in our publications [1–41], and include the Source Forager code-search engine [41]; test-based synthesis using the Splicer [12], SyPet [28] and Hunter [30] technologies and tools; and, the PLINYCompute engine [1, 2, 10], which supports both code search and ML on code artifacts with high scalability.

[2] Note that our research is open and available to any industry partner interested in these capabilities.

3.1 Anomaly Detection

3.1.1 Overview

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

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* $P(Z \mid X = X_F)$, 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 $P(Y \mid Z = Z)$ 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 divergence* (KL-divergence) [46] between the behavior distribution $P_F(Y)$ for F, and the posterior distribution $P(Y \mid X = X_F)$ that the model expects from F. Correctness analysis amounts to checking whether this score is below a threshold.

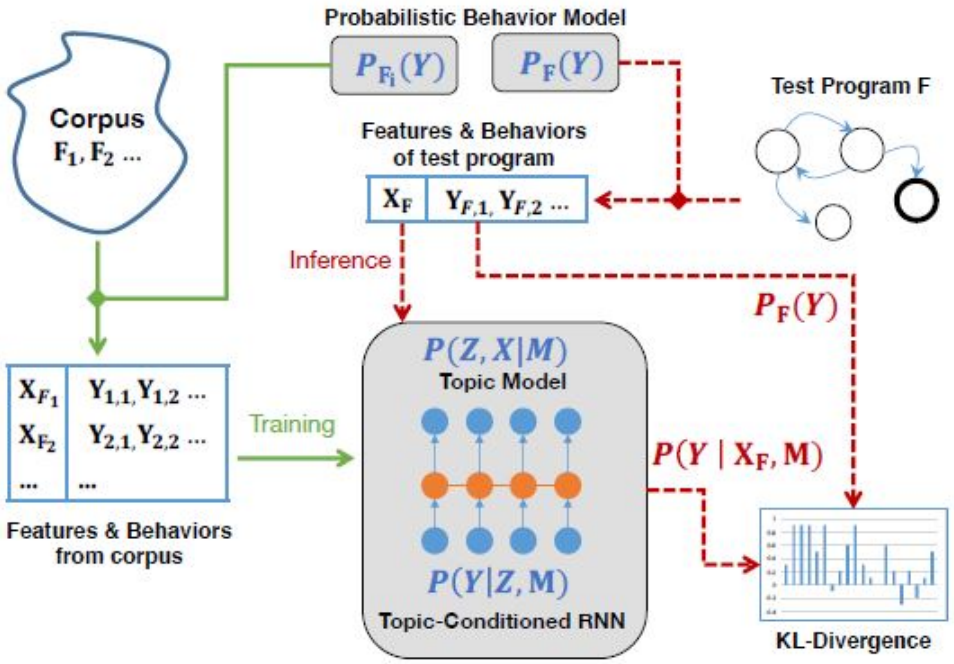

Figure 3: Workflow, with instantiations in grey boxes

The workflow of our method is shown in Figure 3. The training and inference phases are denoted by green (solid) edges and red (dashed) edges respectively. During training, from each program F_i in a corpus of programs F_1, F_2, . . ., we extract a set of features X_{Fi} , and sample a set of behaviors from the distribution $P_{Fi}(Y)$, forming the training data. From this data, we learn the joint distribution $P(X, Y, Z|M)$, where 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 $P(Y|X = X_F, M)$ 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.

Instantiating the Framework

An instantiation of our framework must concretely define program features and behaviors, and the way in which the distributions $P(X, Y, Z|M)$ and $P_F(Y)$ are obtained. 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 i-th component is the probability that F follows the i-th topic. For

example, the topics in a given corpus may correspond to graphical user interface (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) [47] to learn a joint distribution $P(Z, X|M)$ over the topics and features of programs. A *topic-conditioned recurrent neural network model* [48], is used to learn conditional distributions of the form $P(Y|Z = Z, M)$. The joint distribution $P(X, Y, Z|M)$ 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* [49, 50]. The distribution $P_F(Y)$ is simply the semantics of this automaton.

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 4(a).

a(i)

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

(b)(i)

a(ii)

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

(b)(ii)

Figure 4: (a) Abnormal dialog boxes discovered by our anomaly detection (b) Code snippets corresponding to the dialog boxes

The code snippets responsible for these boxes are shown in Figure 4. For example, in Figure 4(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 4(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) ⟨0:98; 0:01; 0:01⟩, and the pattern used in F_2 could be assigned the second topic, resulting in the topic vector ⟨0:01; 0:98; 0:01⟩.

Conditioned on such a topic vector Z, a topic-conditioned recurrent neural network (RNN) provides the probability of an API call sequence Y, that is, $P(Y = Y \mid Z = Z)$. 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 $P_{F1}(Y)$ 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.

3.1.2 Bayesian Specification Framework

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

Program Behaviors and Features

Our framework is parameterized by a programming language. Each program in the language has syntax and operational semantics. Because the details of the language do not matter to the framework, 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* 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 $P_F(Y)$ over the behaviors of F.

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 M. Fully parameterized, this distribution becomes:

$$P(X, Y, Z|M) = P(Y|Z, M)\, P(X|Z, M)P(Z|M) \quad (1)$$

The available data are then used to choose an appropriate set of parameters M. For this, we follow the standard recipe of *maximum likelihood* [51]. Suppose that we are given a large corpus of programs $\{F_1,..., F_N\}$, and for each program F_i we have extracted the pair $(X_{Fi}, \langle Y_{i,1}\ Y_{i,2}\ \ldots\rangle)$ 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 M that maximizes the function:

$$\prod_{i=1}^{N}\left(\int_{Z}\left(\prod_{Y_{i,j}} P(Y_{i,j}|Z=Z,\mathbf{M})\right) P(X_{F_i}|Z=Z,\mathbf{M})P(Z;\mathbf{M})\ \ dZ\right)$$

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 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.

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, $P(X, Y, Z|M)$, we can condition this distribution with the

newly observed X_F, to obtain the posterior:

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

From Equation 1, we have:

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

Applying Bayes' rule to the term $P(X = X_F | Z, M)$ we rewrite $P(Y, Z | X = X_F, M)$ as:

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

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:

$$P(Y|X = X_F, \mathbf{M}) = \int_Z P(Y|Z = Z, \mathbf{M})P(Z|X_F, \mathbf{M})\, dZ \tag{2}$$

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, which would be *anticipated*, given learned parameters M, for a program with a feature set X_F.

On the other hand, we have a distribution $P_F(Y)$ 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, $P(Y | X = X_F, M)$. A measure such as the Kullback-Leibler divergence [46] 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 $P_F(Y)$ and $P(Y | X = X_F, M)$ respectively, and ranging i over the domain of all possible program behaviors in the language Σ^*:

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

Choosing an Abstraction

When instantiating the framework, the exact form of the feature set X_F must be chosen with some care. If the feature set X_F does not provide any abstraction for the program (i.e., X_F is the program itself) and the model and learner are arbitrarily powerful, then $P(Y \mid X = X_F, M)$ (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 X_F that abstracts the program adequately. For example, when debugging API usage, it makes sense to choose X_F as the bag of API calls made by the code. This ensures that $P(Y \mid X = X_F, M)$ 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.

3.1.3 Instantiation of the Framework

Now we present a concrete instantiation of our framework.

Probabilistic Behavior Model $P_F(Y)$

First, our instantiation includes a definition of the probabilistic behavior model $P_F(Y)$. This definition relies on the abstraction of programs as *generative probabilistic automata* [50, 52].

Program Model

A generative probabilistic automaton is a tuple $F = \langle \mathcal{Q}, \Sigma, q_0, Q_A, \delta \rangle$ where $\mathcal{Q}$ is a finite set of states, Σ is the alphabet of observable symbols that was introduced earlier, $q_0 \in \mathcal{Q}$ is the *initial state*, $Q_A \subseteq \mathcal{Q}$ is a set of *final or accepting states*, and δ: $\mathcal{Q} \times \Sigma \times \mathbb{R}_{(0,1)} \times \mathcal{Q}$ 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* [53] 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 $\mathcal{Q}$, 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.

As symbolic execution is a standard method in formal methods [54–56], 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 $q_i \xrightarrow{s_1,p_1} q_j \xrightarrow{s_2,p_2} \ldots \xrightarrow{s_n,p_n} q_n$ 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 $Y \in \Sigma^*$, denoted as $\|\pi\| = s_1 s_2 \ldots s_n$. Let $\prod_F$ be the set of all accepting runs of F, and $\prod_F(Y) \subseteq \prod_F$ be the set of all accepting runs π such that $\|\pi\| = Y$.

The *probabilistic behavior model* $P_F(Y) : \Sigma^* \rightarrow [0, 1]$ is:

$$P_F(Y = Y) = \frac{1}{\nu} \sum_{\pi \in \Pi_F(Y)} P(\pi) \tag{5}$$

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

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

Features

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

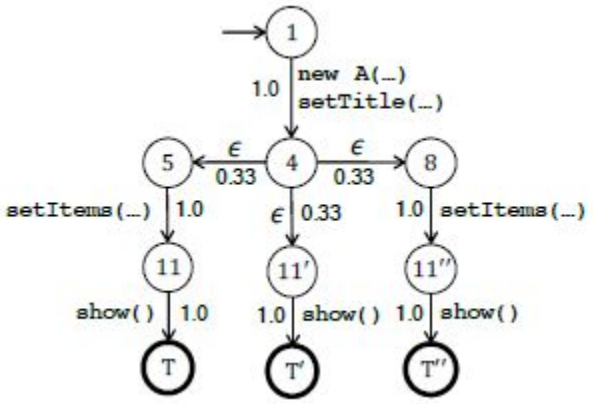

Figure 5: Automation for the example in Figure 4(b)(i)

Example.

The automaton for the code in Figure 4(b)(i) is shown in Figure 5. 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:

Y1 = new A() setTitle(…) setItems(…) show()

Y2 = 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: $0.66 \log \frac{0.66}{0.99} + 0.33 \log \frac{0.33}{10^{-5}} 0.33 = 3.16$. 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 $\log \frac{1}{0.99} = 0.01$.

Topic Models for $P(Z, X|M)$

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 [47] is a popular topic model that models the generative process of documents in a corpus where each document X_{Fi} 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_{Fi}, the topic distribution of document X_{Fi}, and (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_{Fi} and β_k, which forms our model parameter M. During inference, we are given a document X_F, and we would like to compute the posterior distribution $P(\mathbb{Z} \mid \mathbb{X} = \mathsf{X}_F, \mathsf{M})$. Since LDA has already learned a joint distribution $P(\mathbb{Z}, \mathbb{X} \mid \mathsf{M})$, this is simply a matter of conditioning this distribution with the newly observed X_F to get a posterior distribution over $\mathbb{Z}$, which is often approximated through a technique called Gibbs sampling [57].

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

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

Suppose we are given a value of 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 [58]. Let us denote the probability of a symbol s given by the distribution y_t as $y_t(s)$.

A topic-conditioned neural network [48] takes, in addition to 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 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:

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

where W, V, U and T are the weight matrices of the RNN, b_h and 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 $(\mathsf{X}_{Fi}, \langle \mathsf{Y}_{i,1}, \mathsf{Y}_{i,2}, \ldots \rangle)$, then each training step of the RNN will consist of the input x being $\mathsf{Y}_{i,j}$, target output y being $\mathsf{Y}_{i,j}$ 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 $P(\mathbb{Z} \mid \mathbb{X} = \mathsf{X}_F, \mathsf{M})$ 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 M.

During inference, we are given a value Z of $\mathbb{Z}$ and a particular Y = $s_1 s_2 \ldots s_n$, and would like to compute $P(\mathbb{Y}=\mathsf{Y}|\mathbb{Z}=\mathsf{Z}, \mathsf{M})$. This is straightforward: we set x_t as the one-hot vector of s_t for $1 \leq t \leq n$. Then, $P(\mathbb{Y} = \mathsf{Y}|\mathbb{Z} = \mathsf{Z}, \mathsf{M}) = \prod_{t=1}^{n-1} y_t(s_{t+1})$ where y_t is computed using Equation 6.

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 $P(\mathbb{Y}|\mathbb{X} = \mathsf{X_F}, \mathsf{M})$, which in turn requires integrating out the unknown specification $\mathbb{Z}$ (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 $P_F(\mathsf{Y})$, then letting $P_2(\mathsf{Y}) = \log(P_F(\mathbb{Y} = \mathsf{Y})) - \log(P(\mathbb{Y} = \mathsf{Y}|\mathbb{X} = \mathsf{X_F}, \mathsf{M}))$ 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 $P_F(\mathbb{Y})$ is easy: we can use rejection sampling [59] 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 $P_2(\mathbb{Y} = \mathsf{Y})$ easily because of two reasons. First, the term $P_F(\mathbb{Y} = \mathsf{Y})$ (Equation 5) requires summing over possibly infinite number of accepting runs $\prod_F$, and second, as mentioned before, computing $P(\mathbb{Y} = \mathsf{Y}|\mathbb{X} = \mathsf{X_F}, \mathsf{M})$ requires integrating over the unseen $\mathbb{Z}$ value.

To handle this, we extend our sampling-based algorithm. Rather than just sampling behaviors, we sample the set I of $(\mathsf{Y}, \widetilde{\prod_F}, \widetilde{Z_F})$ triples, where $\widetilde{\prod_F}$ is itself a set of accepting runs of F sampled using

the same method, and $\widetilde{Z_F}$ is a set of values sampled from $P(\mathtt{Z} \mid \mathtt{X} = \mathsf{X_F}, \mathsf{M})$. 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 $\widetilde{\prod_F}(\mathrm{Y})$ is the set of paths $\pi \in \widetilde{\prod_F}$ 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 $P_\mathsf{F}(\mathtt{Y} = \mathsf{Y})$, and the sum in the second log term estimates $P(\mathtt{Y} = \mathsf{Y} \mid \mathtt{X} = \mathsf{X_F}, \mathsf{M})$.

The problem is that this estimate will be biased since one cannot commute the expectation operator 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 $P(\mathtt{Y} = \mathsf{Y} \mid \mathtt{X} = \mathsf{X_F}, \mathsf{M})$. Intuitively, this bias is not surprising, since an over-estimate for the probability $P_\mathsf{F}(\mathtt{Y} = \mathsf{Y})$ 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 $bias(\mathsf{Y}, \widetilde{\prod_F}, \widetilde{Z_F})$ that computes the bias of an estimate. Our final estimate for the anomaly score is:

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

3.2 Evidence-based Synthesis

3.2.1 Problem Statement

Assume a universe $\mathbb{P}$ of *programs* and a universe $\mathbb{X}$ of *labels*. Also assume a set of training examples of the form $\{(\mathsf{X_1}, \mathsf{Prog_1}), (\mathsf{X_2}, \mathsf{Prog_2}), \ldots\}$, where each $\mathsf{X_i}$ is a label and each $\mathsf{Prog_i}$ is a program. These examples are sampled from an unknown distribution $\mathtt{Q}(\mathtt{X}, \mathit{Prog})$, where $\mathtt{X}$ and

$Prog$ range over labels and programs, respectively.[3]

We assume an equivalence relation $Eqv \subseteq \mathbb{P} \times \mathbb{P}$ over programs. If $(\mathsf{Prog}_1, \mathsf{Prog}_2) \in Eqv$, then Prog_1 and Prog_2 are *functionally equivalent*. The definition of functional equivalence differs across applications, but in general it asserts that two programs are "just as good as" one another.

The goal of *conditional program generation* is to use the training set to learn a function $g : \mathbb{X} \to \mathbb{P}$ such that the expected value $\mathrm{E}[I((g(\mathsf{X}), \mathsf{Prog}) \in Eqv)]$ is maximized. Here, I is the indicator function, returning 1 if its boolean argument is true, and 0 otherwise. Informally, we are attempting to learn a function g such that if we sample $(\mathsf{X}, \mathsf{Prog}) \sim Q(\mathsf{X}, Prog)$, g should be able to reconstitute a program that is functionally equivalent to Prog, using only the label X.

Instantiation

In this project, we consider a particular form of conditional program generation. We take the domain $\mathbb{P}$ to be the set of possible programs in a programming language called AML that captures the essence of API-heavy Java programs. AML includes complex control flow such as loops, *if-then* statements, and exceptions, access to Java API data types, and calls to Java API methods. AML is a strongly typed language, and by definition, $\mathbb{P}$ only includes programs that are type-safe.[4] To define labels, we assume three finite sets: a set *Calls* of possible API calls in AML, a set *Types* of possible object types, and a set *Keys* of *keywords*, defined as words, such as "read" and "file", that often appear in textual descriptions of what programs do. The space of possible labels is $\mathbb{X} = 2^{Calls} \times 2^{Types} \times 2^{Keys}$ (here 2^S is the power set of S).

Defining *Eqv* in practice is tricky. For example, a reasonable definition of *Eqv* is that $(\mathsf{Prog}_1; \mathsf{Prog}_2) \in Eqv$ iff Prog_1 and Prog_2 produce the same outputs on all inputs. But given the richness of AML, the problem of determining whether two AML programs always produce the same output is undecidable. As such, in practice, we can only measure success indirectly, by checking whether the programs use the same control structures and whether they can produce the same API call sequences.

Example

Consider the label $\mathsf{X} = (\mathsf{X}_{Calls}, \mathsf{X}_{Types}, \mathsf{X}_{Keys})$ where $\mathsf{X}_{Calls} = \{\mathsf{readLine}\}$ and X_{Types} and X_{Keys} are empty. Figure 6(a) shows a program that our best learner stochastically returns given this input. As we see, this program indeed reads lines from a file, whose name is given by a special variable $String that the code takes as input. It also handles exceptions and closes the reader, even though these actions were not directly specified.

[3] We use italic fonts for random variables and sans serif — for example X — for values of these variables.

[4] In research on programming languages, a program is typically judged as type-safe under a type environment, which sets up types for the program's input variables and return value. Here, we consider a program to be type-safe if it can be typed under some type environment.

```
String s;
BufferedReader br;
FileReader fr;
try {
 fr = new FileReader($String);
 br = new BufferedReader(fr);
 while ((s = br.readLine()) != null) {}
 br.close();
} catch (FileNotFoundException _e) {
} catch (IOException _e) {
}
```

(a)

```
String s;
BufferedReader br;
InputStreamReader isr;
try {
 isr = new InputStreamReader($InputStream);
 br = new BufferedReader(isr);
 while ((s = br.readLine()) != null) {}
} catch (IOException _e) {
}
```

(b)

Figure 6: Programs generated by BAYOU with the API method name *readLine* as a label. Names of variables of type *T* whose values are obtained from the environment are of the form *$T*.

Although the program in Figure 6-(a) matches the label well, failures do occur. Sometimes, the system generates a program as in Figure 6-(b), which uses an InputStreamReader rather than a FileReader. It is possible to rule out this program by adding to the label. Suppose we amend X_{Types} so that X_{Types} = {FileReader}. BAYOU now tends to only generate programs that use FileReader. The variations then arise from different ways of handling exceptions and constructing FileReader objects (some programs use a String argument, while others use a File object).

3.2.2 Technical Approach

Our approach is to learn g via maximum conditional likelihood estimation (CLE). That is, given a distribution family $P(Prog|X, \theta)$ for a parameter set θ, we choose $\theta^* = \text{argmax}_\theta \Sigma_i \log P(Prog_i|X_i, \theta)$. Then, $g(X) = \text{argmaxProgP}(Prog|X, \theta^*)$.

The key innovation of our approach is that here, learning happens at a higher level of abstraction than (X_i, $Prog_i$) pairs. In practice, Java-like programs contain many low-level details (for example, variable names and intermediate results) that can obscure patterns in code. Further, they contain complicated semantic rules (for example, for type safety) that are difficult to learn from data. In contrast, these are relatively easy for a combinatorial, syntax-guided program synthesizer to deal with. However, synthesizers have a notoriously difficult time figuring out the correct "shape" of a program (such as the placement of loops and conditionals), which we hypothesize should be relatively easy for a statistical learner.

Specifically, our approach learns over *sketches*: tree-structured data that capture key facets of program syntax. A sketch Y does not contain low-level variable names and operations, but carries information about broadly shared facets of programs such as the types and API calls. During generation, a program synthesizer is used to generate programs from sketches produced by the learner.

Let the universe of all sketches be denoted by $\mathbb{Y}$. The sketch for a given program is computed by applying an *abstraction function* $\alpha : \mathbb{P} \rightarrow \mathbb{Y}$. We call a sketch Y *satisfiable*, and write *sat*(Y), if

$\alpha^{-1}(\mathsf{Y}) \neq \emptyset$. The process of generating (type-safe) programs given a satisfiable sketch Y is probabilistic, and captured by a *concretization distribution* $P(\mathcal{A}|\mathsf{Y}, \text{sat}(\mathsf{Y}))$. We require that for all programs Prog and sketches Y such that *sat*(Y), we have $P(\mathsf{Prog}|\mathsf{Y}) \neq 0$ only if $\mathsf{Y} = \alpha(\mathsf{Prog})$.

Importantly, the concretization distribution is fixed and chosen heuristically. The alternative of learning this distribution from source code poses difficulties: a single sketch can correspond to many programs that only differ in superficial details, and deciding which differences between programs are superficial and which are not requires knowledge about program semantics. In contrast, our heuristic approach utilizes known semantic properties of programming languages like ours — for example, that local variable names do not matter, and that some algebraic expressions are semantically equivalent. This knowledge allows us to limit the set of programs that we generate.

Let us define a random variable $\mathtt{Y} = \alpha(\mathcal{A})$. We assume that the variables $\mathtt{X}$, $\mathtt{Y}$ and $\mathcal{A}$ are related as in the Bayes net in Figure 7. Specifically, given $\mathtt{Y}$, $\mathcal{A}$ is conditionally independent of $\mathtt{X}$. Further, let us assume a distribution family $P(\mathtt{Y}|\mathtt{X}, \theta)$ parameterized on θ.

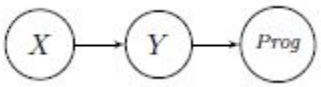

Figure 7: Bayes net for A, B, C

Let $\mathsf{Y}_i = \alpha(\mathsf{Prog}_i)$, and note that $P(\mathsf{Prog}_i|\mathsf{Y}) \neq 0$ only if $\mathsf{Y} = \mathsf{Y}_i$. Our problem now simplifies to *learning over sketches*, i.e., finding:

$$\begin{aligned}\theta^* &= \operatorname*{argmax}_\theta \sum_i \log \sum_{\mathsf{Y}:sat(\mathsf{Y})} P(\mathsf{Prog}_i|\mathsf{Y}) P(\mathsf{Y}|\mathsf{X}_i, \theta) \\ &= \operatorname*{argmax}_\theta \sum_i \log P(\mathsf{Prog}_i|\mathsf{Y}_i) P(\mathsf{Y}_i|\mathsf{X}_i, \theta) = \operatorname*{argmax}_\theta \sum_i \log P(\mathsf{Y}_i|\mathsf{X}_i, \theta). \qquad (7)\end{aligned}$$

Instantiation

Figure 8 shows the full grammar for sketches in our implementation. Here, τ_0, τ_1,… range over a finite set of *API data types* that AML programs can use. A data type, akin to a Java class, is identified with a finite set of *API method names* (including constructors), and ranges over these names. Note that sketches do not contain constants or variable names.

$$
\begin{array}{lcl}
\mathsf{Y} & ::= & \textbf{skip} \mid \textbf{call}\ \mathsf{Cexp} \mid \mathsf{Y}_1; \mathsf{Y}_2 \mid \\
 & & \textbf{if}\ \mathsf{Cseq}\ \textbf{then}\ \mathsf{Y}_1\ \textbf{else}\ \mathsf{Y}_2 \mid \\
 & & \textbf{while}\ \mathsf{Cseq}\ \textbf{do}\ \mathsf{Y}_1 \mid \textbf{try}\ \mathsf{Y}_1\ \mathsf{Catch} \\
\mathsf{Cexp} & ::= & \tau_0.a(\tau_1, \ldots, \tau_k) \\
\mathsf{Cseq} & ::= & \text{List of Cexp} \\
\mathsf{Catch} & ::= & \textbf{catch}(\tau_1)\ \mathsf{Y}_1\ \ldots\ \textbf{catch}(\tau_k)\ \mathsf{Y}_k
\end{array}
$$

Figure 8: Grammar for sketches

A full definition of the abstraction function for AML appears in [7]. As an example, API calls in AML have the syntax "call $e.a(e_1,\ldots,e_k)$", where a is an API method, the expression e evaluates to the object on which the method is called, and the expressions $e_1,\ldots,e_k$ evaluate to the arguments of the method call. We abstract this call into an *abstract method call* "call τ.a(τ_1,…, τ_k)", where τ is the type of e and τ_i is the type of e_i. The keywords skip, while, if-then-else, and try-catch preserve information about control flow and exception handling. Boolean conditions Cseq are replaced by *abstract expressions*: lists whose elements abstract the API calls in Cseq.

3.2.3 Learning

Now we describe our learning approach, shown in Figure 9. Equation 7 leaves us with the problem of computing $\text{argmax}_\theta \Sigma_i \log P(\mathsf{Y}_i|\mathsf{X}_i, \theta)$, when each X_i is a label and Y_i is a sketch. Our answer is to utilize an encoder-decoder and introduce a real vector-valued latent variable $\mathtt{Z}$ to stochastically link labels and sketches: $P(\mathsf{Y}|\mathsf{X}, \theta) = \int_{Z\in\mathbb{R}^m} P(\mathsf{Z}|\mathsf{X}, \theta)P(\mathsf{Y}|\mathsf{Z}, \theta)\mathrm{d}\mathsf{Z}$.

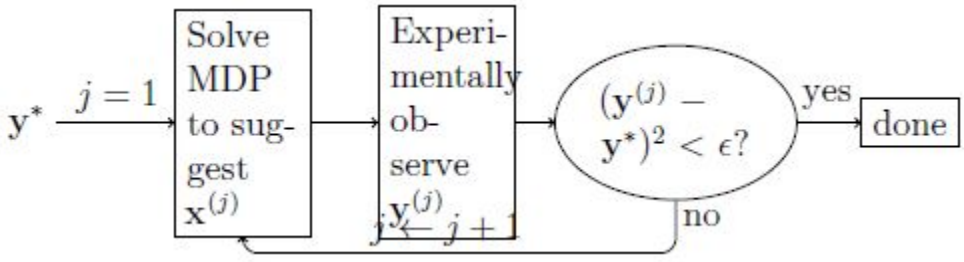

Figure 9: Active learning to "close the loop"

$P(\mathsf{Y}|\mathsf{Z}, \theta)$ is realized as a probabilistic decoder mapping a vector-valued variable to a distribution over trees. As for $P(\mathsf{Z}|\mathsf{X}, \theta)$, this distribution can, in principle, be picked in any way we like. In practice, because both $P(\mathsf{Y}|\mathsf{Z}, \theta)$ and $P(\mathsf{Z}|\mathsf{X}, \theta)$ have neural components with numerous parameters, we wish this distribution to regularize the learner. To provide this regularization, we assume a Normal $(\vec{0}, \mathbf{I})$ prior on $\mathtt{Z}$.

Recall that our labels are of the form $\mathsf{X} = (\mathsf{X}_{Calls}, \mathsf{X}_{Types}, \mathsf{X}_{Keys})$, where X_{Calls}, X_{Types}, and X_{Keys} are sets. Assuming that the j-th elements $\mathsf{X}_{Calls,j}$, $\mathsf{X}_{Types,j}$, and $\mathsf{X}_{Keys,j}$ of these sets are generated independently, and assuming a function f for encoding these elements, let:

$$P(\mathsf{X}|\mathsf{Z},\theta) = \left(\prod_j \text{Normal}(f(\mathsf{X}_{Calls,j})|\mathsf{Z},\mathbf{I}\sigma^2_{Calls})\right)\left(\prod_j \text{Normal}(f(\mathsf{X}_{Types,j})|\mathsf{Z},\mathbf{I}\sigma^2_{Types})\right)$$
$$\left(\prod_j \text{Normal}(f(\mathsf{X}_{Keys,j})|\mathsf{Z},\mathbf{I}\sigma^2_{Keys})\right).$$

That is, the encoded value of each $\mathsf{X}_{\text{Types,j}}$, $\mathsf{X}_{\text{Calls,j}}$ or $\mathsf{X}_{\text{Keys,j}}$ is sampled from a high-dimensional Normal distribution centered at Z. If f is 1-1 and onto with the set $\mathbb{R}^m$ then from Normal-Normal conjugacy, we have: $P(\mathsf{Z}|\mathsf{X})$ = Normal $(\mathsf{Z}|\ \frac{\bar{X}}{1+n}, \frac{1}{1+n}\mathbf{I})$, where:

$$\overline{\mathsf{X}} = \left(\sigma^{-2}_{Types}\sum_j f(\mathsf{X}_{Types,j})\right) + \left(\sigma^{-2}_{Calls}\sum_j f(\mathsf{X}_{Calls,j})\right) + \left(\sigma^{-2}_{Keys}\sum_j f(\mathsf{X}_{Keys,j})\right)$$

and $n = n_{Types}\sigma^{-2}{}_{Types} + n_{Calls}\sigma^{-2}{}_{Calls} + n_{Keys}\sigma^{-2}{}_{Keys}$. Here, n_{Types} is the number of types supplied, and n_{Calls} and n_{Keys} are defined similarly.

Note that this particular $P(\mathsf{Z}|\mathsf{X}, \theta)$ only follows directly from the Normal $(\vec{0}, \mathbf{I})$ prior on Z and Normal likelihood $P(\mathsf{X}|\mathsf{Z}, \theta)$ if the encoding function f is 1-1 and onto. However, even if f is not 1-1 and onto (as will be the case if f is implemented with a standard feed-forward neural network) we can still use this probabilistic encoder, and in practice, we still tend to see the benefits of the regularizing prior on Z, with $P(\mathsf{Z})$ distributed approximately according to a unit Normal. We call this type of encoder-decoder, with a single, Normally-distributed latent variable Z linking the input and output, a Gaussian encoder-decoder (GED).

Now that we have chosen $P(\mathsf{X}|\mathsf{Z}, \theta)$ and $P(\mathsf{Y}|\mathsf{Z}, \theta)$, we must choose θ to perform CLE. Note that:

$$\sum_i \log P(\mathsf{Y}_i|\mathsf{X}_i,\theta) = \sum_i \log \int_{\mathsf{Z}\in\mathbb{R}^m} P(\mathsf{Z}|\mathsf{X}_i,\theta)P(\mathsf{Y}_i|\mathsf{Z},\theta)d\mathsf{Z} = \sum_i \log \mathbf{E}_{\mathsf{Z}\sim P(\mathsf{Z}|\mathsf{X}_i,\theta)}[P(\mathsf{Y}_i|\mathsf{Z},\theta)]$$
$$\geq \sum_i \mathbf{E}_{\mathsf{Z}\sim P(\mathsf{Z}|\mathsf{X}_i,\theta)}[\log P(\mathsf{Y}_i|\mathsf{Z},\theta)] = \mathcal{L}(\theta).$$

where the $\geq$ holds due to Jensen's inequality. Hence, $\mathcal{L}(\theta)$ serves as a lower bound on the log-likelihood, and so we can compute $\theta^* = \text{argmax}_\theta \mathcal{L}(\theta)$ as a proxy for the CLE. We maximize this lower bound using stochastic gradient ascent; as $\mathsf{P}(\mathsf{Z}|\mathsf{X}_i, \theta)$ is Normal, we can use the re-parameterization trick common in variational auto-encoders while doing so. The parameter set θ contains all of the parameters of the encoding function f as well as σ_{Types}, σ_{Calls}, and σ_{Keys}, and the parameters used in the decoding distribution function $P(\mathsf{Y}|\mathsf{Z}, \theta)$.

3.2.4 Combinatorial Concretization

The final step in our algorithm is to "concretize" sketches into programs, following the distribution $P(\mathcal{A}|\mathsf{Y})$. Our method of doing so is a type-directed, stochastic search procedure that builds on combinatorial methods for program synthesis.

Given a sketch Y, our procedure performs a random walk in a space of *partially concretized sketches* (PCSs). A PCS is a term obtained by replacing some of the abstract method calls and expressions in a sketch by AML method calls and AML expressions. For example, the term "$x_1.a(x_2);\ \tau_1.b(\tau_2)$", which sequentially composes an abstract method call to b and a "concrete" method call to a, is a PCS. The state of the procedure at the i-th point of the walk is a PCS H_i. The initial state is Y.

Each state H has a set of *neighbors Next*(H). This set consists of all PCS's H' that are obtained by concretizing a single abstract method call or expression in H, using variable names in a way that is consistent with the types of all API methods and declared variables in H.

The (i+1)-th state in a walk is a sample from a predefined, heuristically chosen distribution $P(\mathsf{H}_{i+1}|\mathsf{H}_i)$. The only requirement on this distribution is that it assigns nonzero probability to a state iff it belongs to *Next*(H_i). In practice, our implementation of this distribution prioritizes programs that are simpler. The random walk ends when it reaches a state H^* that has no neighbors. If H^* is fully concrete (that is, an AML program), then the walk is successful and H^* is returned as a sample. If not, the current walk is rejected, and a fresh walk is started from the initial state.

Recall that the concretization distribution $P(\mathcal{A}|\mathsf{Y})$ is only defined for sketches Y that are satisfiable. Our concretization procedure does not assume that its input Y is satisfiable. However, if Y is not satisfiable, all random walks that it performs end with rejection, causing it to never terminate.

While the worst-case complexity of this procedure is exponential in the generated programs, it performs well in practice because of our chosen language of sketches. For instance, our search does not need to discover the high-level structure of programs. Also, sketches specify the types of method arguments and return values, and this significantly limits the search space.

3.3 Extension project on identifying API misuse in JavaScript code

In this section, we discuss the methods employed in the extension project mentored by GitHub, in which we focused on scanning JavaScript programs for API misuse. While our past work on anomaly detection was prototyped for analyzing Java and C/C++ code, this extension project focused on JavaScript, since it is the most popular programming language used in GitHub projects.

We implemented our approach in JS-Smart, a new tool chain developed in this extension project that performs code mining and anomaly detection for JavaScript codes as described in this section.

Section 3.3.1 describes the overall architecture of JS-Smart, and Sections 3.3.2–3.3.5 describe the components of the end-to-end pipeline that process the code corpus, mine rules, and checks users' input programs.

3.3.1 JS-Smart Architecture

The JS-Smart framework can be divided into two parts: an offline part that processes the code corpus and mines the rules, and an online part that analyzes the users' programs and produces analysis output.

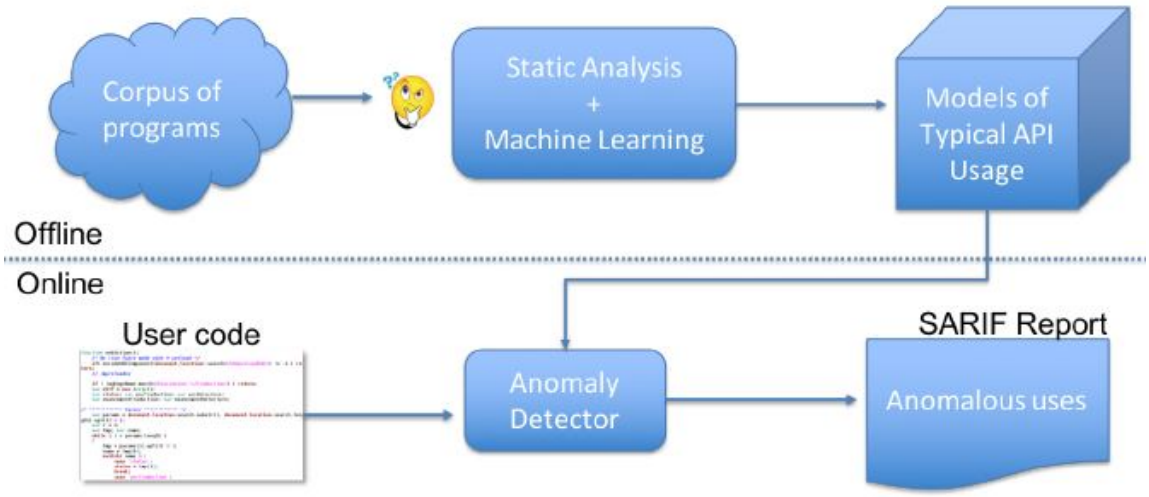

Figure 10: Overall architecture of JS-Smart

Figure 10 shows the overall architecture of JS-Smart. The offline part starts from a corpus collection that downloads node package manager (npm) packages and JavaScript projects from various open source repositories, and crawls through them for feature extraction. Each file is processed via static code analysis techniques that extract call sites and other code features of interest, which in turn are used to build a rule-based statistical model that can be used to identify anomaly patterns.

The key component of the online part is the anomaly detector which takes the user code as input, as well as the statistical model produced by the offline part, and performs a specialized program analysis to identify anomalies which are then written as output. JS-Smart's output is created in Static Analysis Results Interchange Format (SARIF) so that it can be easily displayed by different integrated development environments (IDE's) that accept SARIF.

In general, the end-to-end workflow for JS-Smart can be summarized as follows:

1. Download a given set of target and dependent packages;
2. run feature extraction, and collect the extracted features in JavaScript Object Notation (JSON) files;
3. obtain various statistics on the extracted features;
4. perform rule mining on the extracted features and generate a "rules" file, also represented in JSON format;
5. run an anomaly detector based on the "rules" file on an arbitrary JavaScript/Node.js project and generate output in SARIF format.

3.3.2 Source Code Mining

JS-Smart employs a feature extractor to perform source code mining. The feature extractor takes JavaScript/Node.js program as input, translates them to an intermediate representation (IR) to extract the call site related information. It is implemented as a Python program that performs the following steps:

1. Download the JavaScript/Node.js code corpus from the internet;
2. Derive dependent-target combinations as configurations;
3. Apply parser and feature extraction to Node.js source code with all configurations, and stores the output to files;
4. Aggregate the output files and get total counts of call sites for each function.

For intermediate representation generation, we use the TypeScript compiler [60] which is designed to support an optionally typed superset of the JavaScript programming language, and also supports the full ECMAScript language specification [61]. In our project, we used the TypeScript compiler to parse standard untyped JavaScript code, and to generate an abstract syntax tree (AST) as the intermediate representation used by JS-Smart to extract function call site related information.

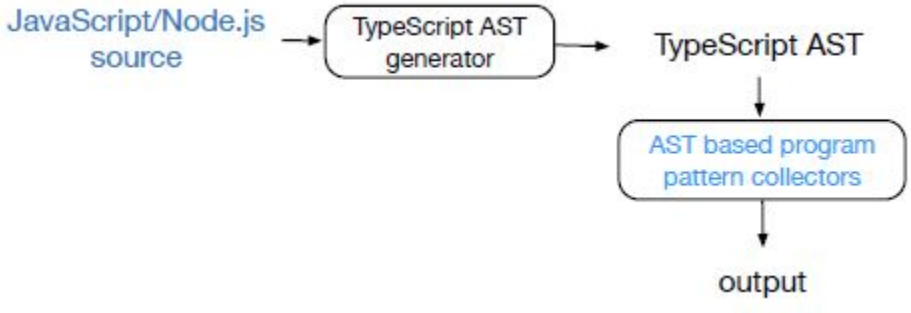

Figure 11: Feature extractions for Source Code Mining

Figure 11 shows the workflow of feature extraction for source code mining. The Node.js source is parsed by the parser to generate the TypeScript AST as IR. An AST based program pattern collector is applied on AST to extract call site information and generate output as a JSON file.

For each call site, the feature extractor extracts several features that will be used for the rule detection process to measure if the call site should be selected as an anomaly. The details of the features are presented in Table 1.

Table 1: Call site features

Name	description
CallSiteName	the target function's name and package name
CallParaminfo	parameters' names
StringArg	whether or not the call parameter is presented as a constant string
CallInTry	whether or not the call site is wrapped by a try-catch block
CallInLoop	whether of not the call site is located inside a loop
UsedReturn	whether or not the return value is used

3.3.3 Statistical Model for Rule Detection

The goal of source code mining is to extract features that feed into a statistical model that can be used to identity anomalies in specific call sites. This rule detector is invoked after the feature extractor collects all call sites with relevant features shown in Table 1 The rule detector scans the call sites, identifies anomalies, and also identifies rules to be used in anomaly detection.

Based on our observations, the specification of JavaScript/Node.js APIs is different from C/C++ and Java APIs which typically involve sequences of API calls. In contrast, JavaScript/Node.js APIs are more self-contained, thus enabling anomaly checking to focus on single API call sites. The statistical model employed here for anomaly detection is a static Bayesian network [62]; it uses a probability threshold to determine whether a pattern of API usage should be reported as an anomaly. The collected rules are converted to JSON format and saved into a JSON file that will be read by the anomaly detector summarized next.

3.3.4 Anomaly Detection

The anomaly detector takes users' programs as input, scans the source code and performs the pattern match on the call sites with the rules mined by rule detector. Figure 12 shows the implementation of the anomaly detector, which is similar in structure to the feature extractor for source code mining (Figure 11) since they both employ the TypeScript compiler as the IR generator.

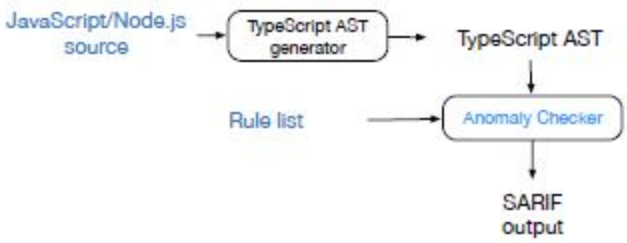

Figure 12: Anomaly detection with the rules

However, there are important differences as well, since source code mining is performed on every program scanned from source code repositories, whereas the anomaly detector is only performed on specific programs submitted by a user. The anomaly detector also imports rules from the rule file and invokes the TypeScript compiler to parse the input JavaScript/Node.js code and generate the IR (presented as TypeScript AST). For each call site in IR, the anomaly detector checks whether it is a library API call; if so, it checks whether the source program violates a previously-mined rule. Any such call sites are emitted as output in SARIF format, along with the relevant rule violation information.

3.3.5 User Interfaces

We explored different user interfaces to enable the interaction between end-users (which can include software developers and "rule curators") and the toolsets (feature extractor, rule miner and anomaly detector). For the offline part of JS-Smart, both the feature extractor and the rule miner can be run as command-line tools by staff that is responsible for curating rules. We also encapsulated these two tools as a docker container that can be easily set up and deployed.

The anomaly detector is the online part that interacts with the software developers directly by analyzing their code; it can be invoked either as an integrated development environment plugin or as a cloud service. In the IDE case, the anomaly detector can analyze the user's code directly from the IDE editor. In the cloud service case, it runs as a backend service triggered by users' commit-like operations and sends feedback to the user via the web information update. As we learned from our GitHub mentors, the second use case is well supported by the GitHub App interface.

The anomaly detector is implemented as a static analyzer that helps users identify misuse of JavaScript APIs. As indicated earlier, the analyzer has been packaged as two different tools to address two different use cases:

1. A plugin for Microsoft Visual Studio Code IDE, that takes the JavaScript being edited in the IDE as input, performs analysis and outputs the analysis result presented in SARIF [63] format which can be displayed conveniently in the IDE.
2. A GitHub App that captures user events to perform rule-based analysis on the user's repository, and sends the output to the user's GitHub issues page.

In the Github App based implementation, the anomaly detector is executed as a backend service that is invoked by the GitHub App engine on the user's repository (each Github repository owner can customize its GitHub App via a web interface). The actual workflow of the GitHub App based anomaly detector is presented in Figure 13. It captures users' "git push" events, pulls the latest version of the repository to temporary storage, and runs the anomaly detection process on the user's code. The analysis result is updated on the user's "issues" page. Another way to trigger this workflow using a GitHub App is to capture a special "issue creation" event with a pre-specified title, and the analysis result is updated in the same "issue". As shown in Figure 13, the actual implementation has two parts, one of them is a GitHub App client that captures the user's operation on a git repository, communicates with the anomaly detector and updates the analysis result in the user's repository issues page. Another part is the anomaly detector that runs on the server-side (we deployed our anomaly detector on an Amazon Web Services (AWS) lightsail cloud server), analyzes the code pulled from the user's repository, produces analysis results in SARIF format and sends them back to the GitHub App client.

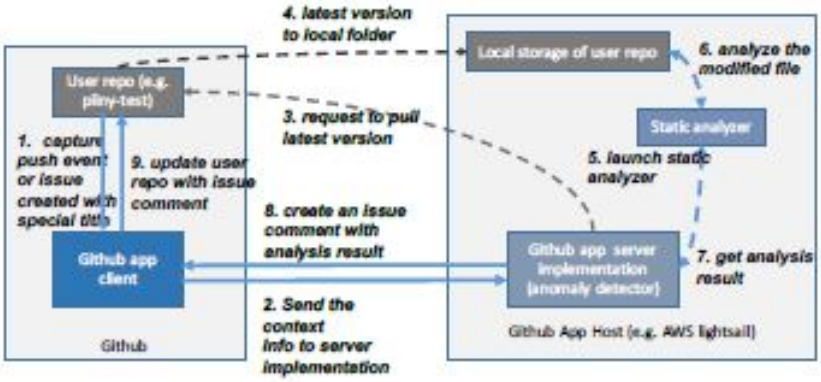

Figure 13: The workflow of JS-Smart Github App

4 Results and Discussion

4.1 Anomaly Detection Results

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 4.1)?
2. Using the specifications, can we find possible bugs in the usage of the Android API in a corpus (Section 4.1)?
3. How does specification learning help in anomaly detection (Section 4.1)?
4. How does the Bayesian framework help in handling heterogeneity in the specifications (Section 4.1)?

Implementation and Experimental Setup

Now we briefly describe the system, SALENTO, which implements our method. SALENTO uses SOOT [64] to implement symbolic execution and transform code in an Android app into our automaton model, TENSORFLOW [65] to implement the topic-conditioned RNN, and SciPy [66] 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 over approximate 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 specifications on more complex programming constructs.

The training corpus consisted of 500 Android apps from [67], and the testing corpus consisted of 250 apps from [68]. 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 graphics processing unit (GPU).

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 14 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.

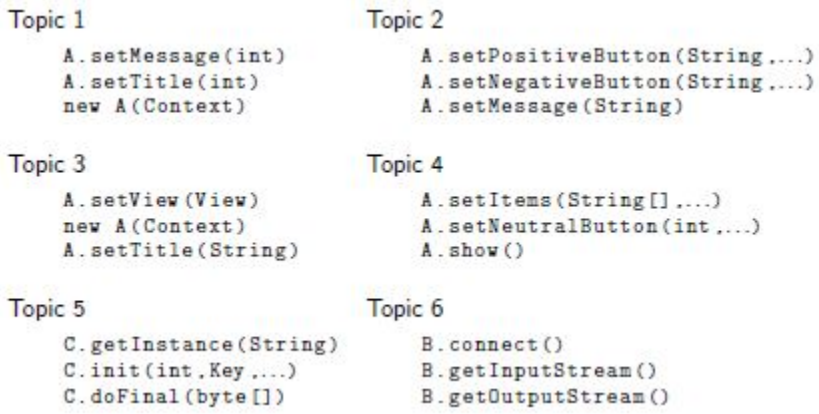

Figure 14: Top-3 methods from topics extracted by LDA (A = AlertDialog:Builder, B = BluetoothSocket, C = Cipher)

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 Extensible Markup Language (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 4(a)(ii)).

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

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 15(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, which would have determined its anomaly score.

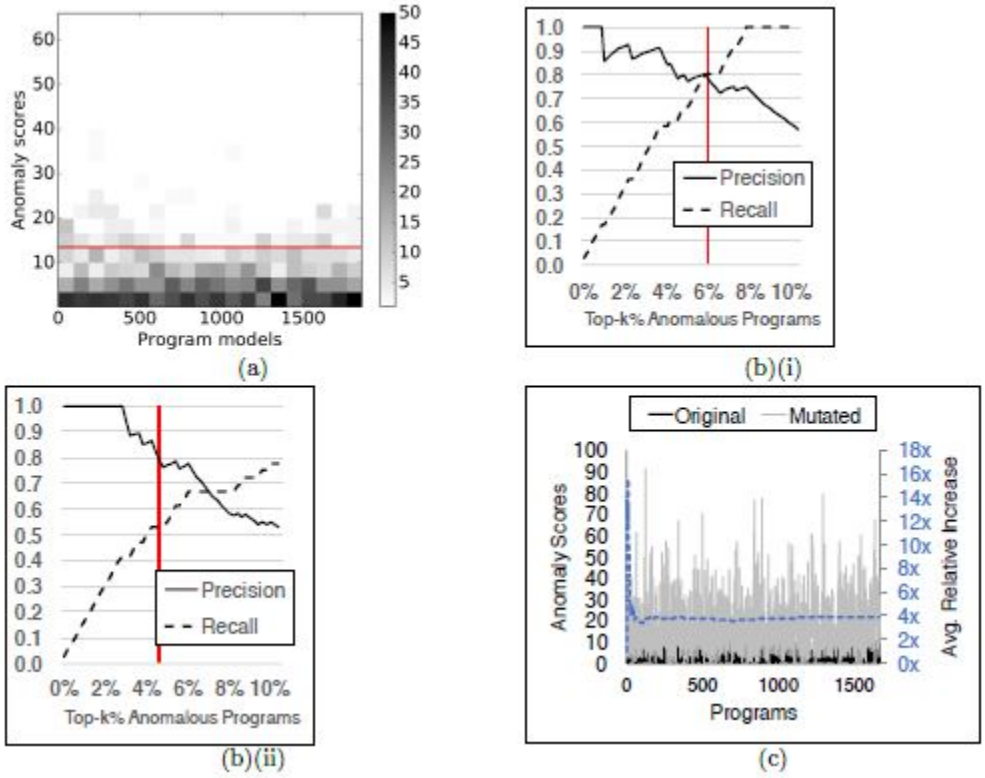

Figure 15: (a) Histogram of anomaly score values, (b) Precision-recall for the possible bugs in Figure 16 for (i) Bayesian model (ii) non-Bayesian model, and (c) Anomaly scores of remaining 90% programs before and after mutation

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 the 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 the scope of this project. 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.

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

Through manual inspection and triage, we found 10 different types of possible bugs in our testing corpus (Figure 16), ranging from the benign to the insidious. We have already seen anomalies #6 and #10 (Figure 4) 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 electronic code book (ECB) encryption mode, and #8 could cause future attempts to open a socket to be blocked.

Figure 15(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 [69]. 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 15(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.

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 [70] 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 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.

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 [71] or RNNs [72] to learn a (non-Bayesian) single 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 4.1, querying the trained model with behaviors in the testing program for inference.

Figure 15(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 newly learned specifications. Figure 17(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), which lingers close to 1.0 on average. However, the non-Bayesian specification (solid line) suffers from about a 2x increase.

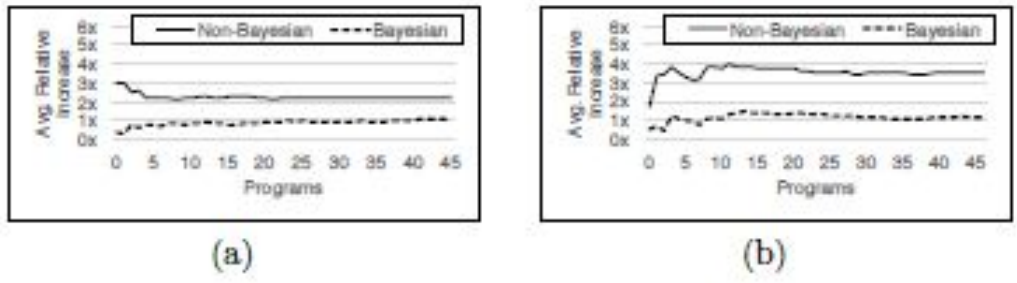

Figure 17: Average relative increase in anomaly scores of BluetoothSocket programs when the training corpus only uses the APIs (a) BluetoothSocket, Cipher (b) AlertDialog:Builder, BluetoothSocket, Cipher

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 4.1). In Figure 17(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.

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, we 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.

4.2 Evidence-based Synthesis Results

Now we present an empirical evaluation of the effectiveness of our method. The experiments we describe utilize data from an online repository of about 1500 Android apps [67]. We decompiled the Android Package Kits (APKs) using JADX [73] to generate their source code. Analyzing about 100 million lines of code that were generated, we extracted 150,000 methods that used Android APIs or the Java library. We then pre-processed all method bodies to translate the code from Java to AML, preserving names of relevant API calls and data types as well as the high-level control flow. Hereafter, when we say "program" we refer to an AML program.

From each program, we extracted the sets X_{Calls}, X_{Types}, and X_{Keys} as well as a sketch Y. Lacking separate natural language descriptions for programs, we defined keywords to be words obtained by splitting the names of the API types and calls that the program uses, based on camel case. For instance, the keywords obtained from the API call `readLine` are "read" and "line". As API methods and types in Java tend to be carefully named, these words often contain rich information about what programs do. Figure 18 gives some statistics on the sizes of the labels in the data. From the extracted data, we randomly selected 10,000 programs to be in the testing and validation data.

	Min	Max	Median	Vocab
X_{Calls}	1	9	2	2584
X_{Types}	1	15	3	1521
X_{Keys}	2	29	8	993
X	4	48	13	5098

Figure 18: Statistics on labels

Implementation and training

We implemented our approach in our tool called BAYOU, using TENSORFLOW [65] to implement the GED neural model and the Eclipse IDE for the abstraction from Java to the language of sketches and the combinatorial concretization.

In all our experiments we performed cross-validation through grid search and picked the best performing model. Our hyper-parameters for training the model are as follows. We used 64, 32 and 64 units in the encoder for API calls, types, and keywords, respectively, and 128 units in the decoder. The latent space was 32-dimensional. We used a mini-batch size of 50, a learning rate of 0.0006 for the Adam gradient-descent optimizer [74], and ran the training for 50 epochs.

The training was performed on an AWS "p2.xlarge" machine with an NVIDIA K80 GPU with 12GB GPU memory. As each sketch was broken down into a set of production paths, the total number of data points fed to the model was around 700,000 per epoch. Training took 10 hours to complete.

Clustering

To visualize clustering in the 32-dimensional latent space, we provided labels `X` from the testing data and sampled `Z` from `P(Z|X)`, and then used it to sample a sketch from `P(Y|Z)`. We then used t-distributed stochastic neighbor embedding (t-SNE) [75] to reduce the dimensionality of `Z` to 2-dimensions, and labeled each point with the API used in the sketch `Y`. Figure 19 shows this 2-dimensional space, where each label has been coded with a different color. It is immediately apparent from the plot that the model has learned to cluster the latent space neatly according to different APIs. Some APIs such as `java:io` have several modes, and we noticed separately that each mode corresponds to different usage scenarios of the API, such as reading versus writing in this case.

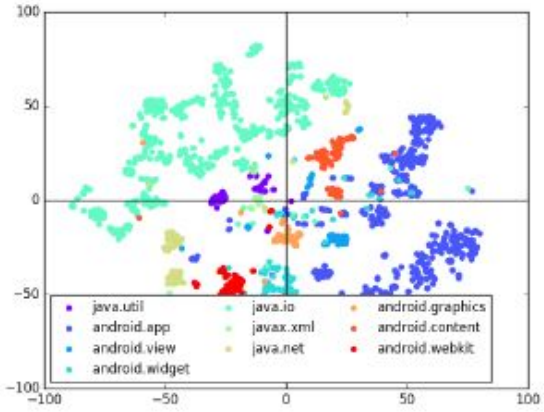

Figure 19: 2-dimensional projection of latent space

Accuracy

To evaluate prediction accuracy, we provided labels from the testing data to our model, sampled sketches from the distribution P(Y|X) and concretized each sketch into an AML program using our combinatorial search. We then measured the number of test programs for which a program that is equivalent to the expected one appeared in the top-10 results from the model.

As there is no universal metric to measure program equivalence (in fact, it is an undecidable problem in general), we used several metrics to approximate the notion of equivalence. We defined the following metrics on the top-10 programs predicted by the model:

M1 This binary metric measures whether the expected program appeared in a syntactically equivalent form in the results. Of course, an impediment to measuring this is that the names of variables used in the expected and predicted programs may not match. It is neither reasonable nor useful for any model of code to learn the exact variable names in the training data. Therefore, in performing this equivalence check, we abstract away the variable names and compare the rest of the program's Abstract Syntax Tree instead.

M2 This metric measures the minimum *Jaccard distance* between the sets of sequences of API calls made by the expected and predicted programs. It is a measure of how close to the original program were we able to get in terms of sequences of API calls.

M3 Similar to metric M2, this metric measures the minimum Jaccard distance between the sets of API calls in the expected and predicted programs.

M4 This metric computes the minimum absolute difference between the number of statements in the expected and sampled programs, as a ratio of that in the former.

M5 Similar to metric M4, this metric computes the minimum absolute difference between the number of control structures in the expected and sampled programs, as a ratio of that in the former. Examples of control structures are branches, loops, and try-catch statements.

Partial Observability

To evaluate our model's ability to predict programs given a small amount of information about its code, we varied the fraction of the set of API calls, types, and keywords provided as input from the testing data. We experimented with 75%, 50% and 25% observability in the testing data; the median number of items in a label in these cases were 9, 6, and 2, respectively.

Competing Models

In order to compare our model with state-of-the-art conditional generative models, we implemented the Gaussian Stochastic Neural Network (GSNN) presented by [76], using the same tree-structured decoder as the GED. There are two main differences: (i) the GSNN's decoder is also conditioned directly on the input label X in addition to Z, which we accomplish by concatenating its initial state with the encoding of X, (ii) the GSNN loss function has an additional KL-divergence term weighted by a hyper-parameter β. We subjected the GSNN to the same training and cross-validation process as our model. In the end, we selected a model that happened to have very similar hyper-parameters as ours, with β = 0:001.

Evaluating Sketches

In order to evaluate the effect of sketch learning for program generation, we implemented and compared it with a model that learns directly over programs. Specifically, the neural network structure is exactly the same as ours, except that instead of being trained on production paths in the sketches, the model is trained on production paths in the ASTs of the AML programs. We selected a model that had more units in the decoder (256) compared to our model (128), as the AML grammar is more complex than the grammar of sketches. We also implemented a similar GSNN model to train over AML ASTs directly.

Figure 20 shows the collated results of this evaluation, where each entry computes the average of the corresponding metric over the 10000 test programs. It takes our model about 8 seconds, on average, to generate and rank 10 programs.

Model	Input Label Observability			
	100%	75%	50%	25%
GED-AML	0.13	0.09	0.07	0.02
GSNN-AML	0.07	0.04	0.03	0.01
GED-Sk	**0.59**	**0.51**	**0.44**	**0.21**
GSNN-Sk	0.57	0.48	0.41	0.18

(a) M1. Proportion of test programs for which the expected AST appeared in the top-10 results.

Model	Input Label Observability			
	100%	75%	50%	25%
GED-AML	0.82	0.87	0.89	0.97
GSNN-AML	0.88	0.92	0.93	0.98
GED-Sk	**0.34**	**0.43**	**0.50**	**0.76**
GSNN-Sk	0.36	0.46	0.53	0.78

(b) M2. Average minimum Jaccard distance on the set of sequences of API methods called in the test program vs the top-10 results.

Model	Input Label Observability			
	100%	75%	50%	25%
GED-AML	0.52	0.58	0.61	0.77
GSNN-AML	0.59	0.64	0.68	0.83
GED-Sk	**0.11**	**0.17**	**0.22**	**0.50**
GSNN-Sk	0.13	0.19	0.25	0.52

(c) M3. Average minimum Jaccard distance on the set of API methods called in the test program vs the top-10 results.

Model	Input Label Observability			
	100%	75%	50%	25%
GED-AML	0.49	0.47	0.46	0.46
GSNN-AML	0.52	0.49	0.49	0.53
GED-Sk	**0.05**	**0.06**	**0.06**	**0.09**
GSNN-Sk	**0.05**	**0.06**	**0.06**	**0.09**

(d) M4. Average minimum difference between the number of statements in the test program vs the top-10 results.

Model	Input Label Observability			
	100%	75%	50%	25%
GED-AML	0.31	0.30	0.30	0.34
GSNN-AML	0.32	0.31	0.32	0.39
GED-Sk	**0.03**	**0.03**	**0.03**	0.04
GSNN-Sk	**0.03**	**0.03**	**0.03**	**0.03**

(e) M5. Average minimum difference between the number of control structures in the test program vs the top-10 results.

Model	Metric				
	M1	M2	M3	M4	M5
GED-AML	0.02	0.97	0.71	0.50	0.37
GSNN-AML	0.01	0.98	0.74	0.51	0.37
GED-Sk	**0.23**	**0.70**	**0.30**	**0.08**	**0.04**
GSNN-Sk	0.20	0.74	0.33	**0.08**	**0.04**

(f) Metrics for 50% observability evaluated only on unseen data

Figure 20: Accuracy of different models on testing data. GED-Aml and GSNN-Aml are baseline models trained over Aml ASTs, GED-Sk and GSNN-Sk are models trained over sketches.

When testing models that were trained on AML ASTs, namely the GED-AML and GSNN- AML models, we observed that out of a total of 87,486 AML ASTs sampled from the two models, 2525 (or 3%) ASTs were not even well-formed, i.e., they would not pass a parser, and hence had to be discarded from the metrics. This number is 0 for the GED-Sk and GSNN-Sk models, meaning that all AML ASTs that were obtained by concretizing sketches were well-formed.

In general, one can observe that the GED-Sk model performs best overall, with GSNN-Sk a reasonable alternative. We hypothesize that the reason GED-Sk performs slightly better is the regularizing prior on Z; since the GSNN has a direct link from X to Y, it can choose to ignore this regularization. We would classify both these models as suitable for conditional program generation. However, the other two models GED-AML and GSNN-AML perform quite worse, showing that sketch learning is key in addressing the problem of conditional program generation.

Generalization

To evaluate how well our model generalizes to unseen data, we gather a subset of the testing data whose data points, consisting of label-sketch pairs (X, Y), never occurred in the training data. We then evaluate the same metrics in Figure 20(a)-(e), but due to space reasons we focus on the 50% observability column. Figure 20(f) shows the results of this evaluation on the subset of 5126 (out of 10000) unseen test data points. The metrics exhibit a similar trend, showing that the models based on sketch learning are able to generalize much better than the baseline models and that the GED-Sk model performs the best.

4.3 Extension Project Results

This section discusses experimental results obtained by applying JS-Smart on a set of JavaScript/Node.js programs.

Experimental Setup

For this evaluation, we ran the feature collection and rule mining processes in JS-Smart on a Linux workstation with a 40-Core Intel Xeon processor with 128GB of random access memory (RAM). The two major programming systems used were Python3 (version 3.5, for JS-Smart tools) and Node.js (version 9.1, for user code). The TypeScript compiler used in this project was version 3.5.3.

Feature Extraction

The statistics related to feature extraction for this evaluation are listed in Table 2.

Table 2: Statistics for feature-extraction (code-mining).

Target library packages	670
Unique dependent packages	161,429
Library API functions targeted	8,419
Relevant call-sites in client packages	1,024,236
LOC analyzed	782,933,215

Rule Mining

Our goal is to achieve low false-positives, so the mined rules are further triaged and curated. We focused on "fixed string" style rules, since the Open Web Application Security Project (OWASP) vulnerabilities (A2: Broken authentication, A3: Sensitive data exposure, A6: Security mis-configuration, A10: Insufficient Logging & Monitoring) are all relevant to the string parameters for API calls. Table 3 summarizes statistics for the rules that we mined. We manually identified 34 rules that are security relevant among the 128 rules that were mined overall.

Table 3: Statistics for mined rules.

Category	#
Security Relevant	34
Type Mismatch	70
Fixed Computation	24

Beside "fixed string" style rules, we also identified the following types of rules as security relevant — "run in loop", "run in try-catch" and "used return value". "Run in loop" helps with identifying Distributed Denial of Service (DDoS) vulnerabilities, e.g., if a network "send" API is invoked within a loop, it could be a potential DDoS attack. "Run in try-catch" identifies cases where API invocation was not protected by an appropriate exception handling block. The "used return value" rule identifies APIs with return values that must be used by the caller, and not ignored.

Detected Vulnerabilities

We applied the JS-Smarts' anomaly detector to the Node.js applications provided by end-users (i.e. software developers who volunteered for evaluating our tools). As mentioned above, we mined 4 types of security relevant rules: "fixed string", "run in loop", "run in try-catch" and "used return value". Figure 21 contains examples of violations detected by our anomaly detector for all four rules.

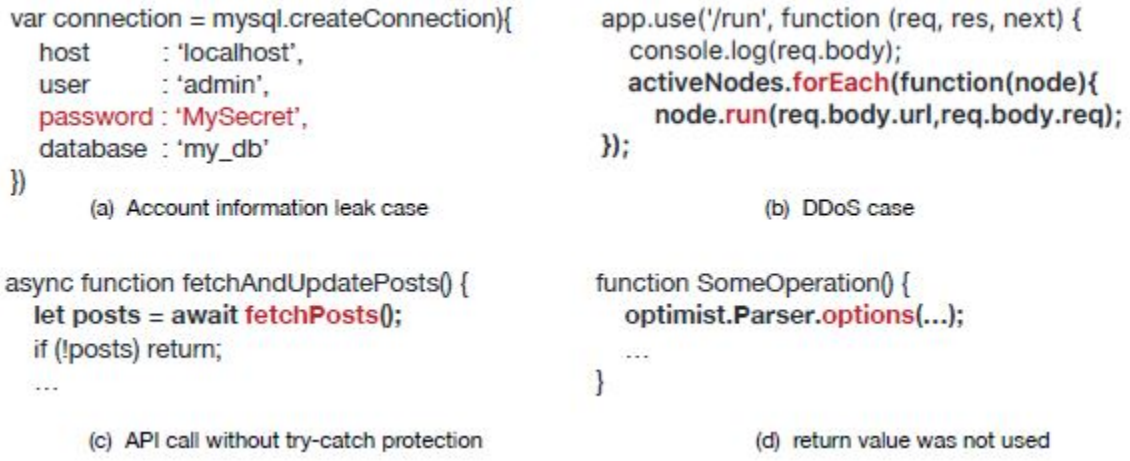

Figure 21: The cases identified by JS-Smart anomaly detector.

5 Conclusions

The PLINY project has introduced technologies that leverage the institutional knowledge embodied in the vast corpus of existing software to simplify the creation of new software. The main advances produced by the PLINY project include: Source Forager, a novel, accurate and reliable query-relevant code-search engine; SALENTO, a novel Bayesian framework for finding API usage errors (anomaly detection) in Android programs; Splicer, SyPet (a program synthesis tool) and Hunter (a tool to facilitates code reuse) for test-based synthesis; BAYOU an open-source evidence-based synthesis system for generating API-heavy Java code; the PLINYCompute system, a scalable infrastructure supporting high-performance, data-intensive, distributed computing tools and libraries, suited for search and learning applications; and extensions of these to JavaScript programs. The PLINY technologies extend the reasoning capabilities of the programmer through the use of automated analysis and synthesis tools. The results of the research have been disseminated in multiple software releases and multiple peer-reviewed publications [1–40]. We believe that the technologies developed in the PLINY project help address many of the productivity and cost challenges associated with software development today, and will also reshape the way people think about programming in the future.

References

[1] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. Declarative recursive computation on an RDBMS. *PVLDB*, 12(7):822–835, 2019.

[2] Jia Zou, Arun Iyengar, and Chris Jermaine. Pangea: Monolithic distributed storage for data analytics. *PVLDB*, 12(6):681–694, 2019.

[3] Yuepeng Wang, Xinyu Wang, and Isil Dillig. Relational program synthesis. *PACMPL*, 2(OOPSLA):155:1–155:27, 2018.

[4] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. Verified three-way program merge. *PACMPL*, 2(OOPSLA):165:1–165:29, 2018.

[5] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas W. Reps. Code vectors: understanding programs through embedded abstracted symbolic traces. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 163–174. ACM, 2018.

[6] Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L. McMillan. Learning abstractions for program synthesis. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 2018.

[7] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[8] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5052–5061. PMLR, 2018.

[9] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435. ACM, 2018.

[10] Jia Zou, R. Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. Plinycompute: A platform for high-performance, distributed, data-intensive tool development. In Gautam

Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1189–1204. ACM, 2018.

[11] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *PVLDB*, 11(5):580–593, 2018.

[12] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. Program splicing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 338–349, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.

[14] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. Non-linear reasoning for invariant synthesis. *PACMPL*, 2(POPL):54:1–54:33, 2018.

[15] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.

[16] William R. Harris, Somesh Jha, Thomas W. Reps, and Sanjit A. Seshia. Program synthesis for interactive-security systems. *Formal Methods Syst. Des.*, 51(2):362–394, 2017.

[17] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[18] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *PACMPL*, 1(OOPSLA):62:1–62:26, 2017.

[19] Venkatesh Srinivasan, Ara Vartanian, and Thomas W. Reps. Model-assisted machine-code synthesis. *PACMPL*, 1(OOPSLA):61:1–61:26, 2017.

[20] Binhang Yuan, Vijayaraghavan Murali, and Christopher M. Jermaine. Abridging source code. *PACMPL*, 1(OOPSLA):58:1–58:26, 2017.

[21] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 151–162. ACM, 2017.

[22] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. The care and feeding of wild-caught mutants. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 511–522. ACM, 2017.

[23] Sourav Sikdar, Kia Teymourian, and Chris Jermaine. An experimental comparison of complex object implementations for big data systems. In *Proceedings of the 2017*

Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017, pages 432–444. ACM, 2017.

[24] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436. ACM, 2017.

[25] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. Compositional recurrence analysis revisited. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 248–262. ACM, 2017.

[26] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. Newtonian program analysis via tensor product. *ACM Trans. Program. Lang. Syst.*, 39(2):9:1–9:72, 2017.

[27] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[28] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017.

[29] Tushar Sharma and Thomas W. Reps. Sound bit-precise numerical domains. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 500–520. Springer, 2017.

[30] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. Hunter: next-generation code reuse for java. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 1028–1032. ACM, 2016.

[31] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 508–521. ACM, 2016.

[32] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa*

Barbara, CA, USA, June 13-17, 2016, pages 57–69. ACM, 2016.

[33] Thomas W. Reps and Aditya V. Thakur. Automating abstract interpretation. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 3–40. Springer, 2016.

[34] Antoine Miné, Jason Breck, and Thomas W. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 560–588. Springer, 2016.

[35] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. Newtonian program analysis via tensor product. In Rastislav Bodík and Rupak Majumdar, editors, *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 663–677. ACM, 2016.

[36] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In Rastislav Bodík and Rupak Majumdar, editors, *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 789–801. ACM, 2016.

[37] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 520–534. ACM, 2015.

[38] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239. ACM, 2015.

[39] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for java. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, volume 9458 of *Lecture Notes in Computer Science*, pages 465–484. Springer, 2015.

[40] Anna Drummond, Yanxin Lu, Swarat Chaudhuri, Christopher M. Jermaine, Joe Warren, and Scott Rixner. Learning to grade student programs in a massive open online course. In Ravi Kumar, Hannu Toivonen, Jian Pei, Joshua Zhexue Huang, and Xindong Wu,

editors, *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 785–790. IEEE Computer Society, 2014.

[41] Vineeth Kashyap, David Bingham Brown, Ben Liblit, David Melski, and Thomas W. Reps. Source forager: A search engine for similar source code. *CoRR*, abs/1706.02769, 2017.

[42] Pliny team at Rice University. A statistical bug-detection framework based on the machine learning model. https://github.com/capergroup/salento/, 2017.

[43] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. Hunter: Next-generation code reuse for java. https://marketplace.eclipse.org/content/hunter, 2020.

[44] Pliny team at Rice University. A data-driven program synthesis system for java api idioms. https://github.com/capergroup/bayou/, 2018.

[45] Pliny team at Rice University. A platform for high-performance distributed tool and library development. https://github.com/riceplinygroup/plinycompute, 2018.

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

[47] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[48] Tomas Mikolov and Geoffrey Zweig. Context dependent recurrent neural network language model. In *SLT*, pages 234–239, 2012.

[49] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

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

[51] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

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

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

[54] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew 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.

[55] Cristian Cadar, Daniel Dunbar, and Dawson 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.

[56] Saswat Anand, Corina S Păsăreanu, and Willem 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.

[57] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, November 1984.

[58] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *journal of machine learning research*, 3(Feb):1137–1155, 2003.

[59] John Von Neumann. 13. various techniques used in connection with random digits. 1951.

[60] Microsoft. The typescript programming language, 2020.

[61] Ecma International. Ecmascript language specification, 2015.

[62] Dawn E. Holmes and Lakhmi C. Jain, editors. *Innovations in Bayesian Networks: Theory and Applications*, volume 156 of *Studies in Computational Intelligence*. Springer, 2008.

[63] OASIS Committee. The static analysis results interchange format (sarif). https://docs.oasis-open.org/sarif/sarif/v2.1.0/cs01/, 2020.

[64] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay 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.

[65] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[66] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-11-09].

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

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

[69] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson 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, February 2010.

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

[71] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin 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.

[72] Veselin Raychev, Martin Vechev, and Eran 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.

[73] skylot. JADX - Dex to Java decompiler. `https://github.com/skylot/jadx`, 2017. [Online; accessed 06-Jul-2017].

[74] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[75] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[76] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3483–3491, 2015.

List of Symbols, Abbreviations, and Acronyms

ACM	Association for Computing Machinery
AML	not an acronym, a programming language that captures the essence of API-heavy Java programs
API	Application Programming Interface
APK	Android Package Kit
AST	Abstract Syntax Tree
AWS	Amazon Web Services
BAYOU	not an acronym, system for generating API heavy Java code
CAV	International Conference on Computer-Aided Verification
CLE	Conditional Likelihood Estimation
DARPA	Defense Advanced Research Agency
DDoS	Distributed Denial of Service
ECB	Electronic Code Book
ECMAScript	not an acronym, a general purpose programming language as defined in ECMA-262 by ECMA International
ESEC	European Software Engineering Conference
ESOP	European Symposium on Programming
FSE	Symposium on Foundation of Software Engineering
GED	Gaussian Encoder-Decoder
GPU	Graphics Processing Unit
GSNN	Gaussian Stochastic Neural Network
GUI	Graphical User Interface
HUNTER	not an acronym, a tool that facilitates code reuse
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment

IEEE	Institute of Electrical and Electronics Engineers
IR	Intermediate Representation
JADX	not acronym, a command line and GUI tools to produce Java source code from Android Dex and Apk files
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KL-divergence	Kullback-Leibler divergence
k-NN	k-nearest neighbor
LDA	Latent Dirichlet Allocation
ML	Machine Learning
MUSE	Mining and Understanding of Software Enclaves
NDSS	The Network and Distributed System Security Symposium
npm	node package manager
OOPSLA	Object Oriented Programming, Systems, Languages and Applications
OWASP	Open Web Application Security Project
PCS	Partially Concretized Sketches
PLDI	ACM SIGPLAN Conference on Programming Language Design and Implementation
PLINY	not an acronym, An End-to-End Framework for Big Code Analytics
PODS	Principles of Database Systems
POPL	ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages
PVLDB	Proceedings of the VLDB Endowment
RAM	Random Access Memory
RNN	Recurrent neural network
SALENTO	not an acronym, system for finding API usage errors
SARIF	Static Analysis Results Interchange Format
SciPy	not an acronym, Python based ecosystem of open-source software for mathematics, science and engineering

SIGACT	Special Interest Group on Algorithms and Computation Theory
SIGMOD	Special Interest Group on Management of Data
SIGPLAN	Special Interest Group on Programming Languages
SIGSOFT	Special Interest Group on Software Engineering
SOOT	not an acronym, a Java compiler infrastructure
SPLICER	not an acronym, a program synthesis tool
SYPET	not an acronym, a program synthesis tool
TA	Technical Area
t-SNE	t-Distributed Stochastic Neighbor Embedding
TOPLAS	ACM Transactions on Programming Languages and Systems
VLDB	International Conference on Very Large Data Bases
VMCAI	International Conference on Verification, Model Checking, and Abstract Interpretation
XML	Extensible Markup Language