AFRL-RI-RS-TR-2020-081

PETABLOX: LARGE-SCALE SOFTWARE ANALYSIS AND ANALYTICS USING DATALOG

GEORGIA TECHNOLOGY RESEARCH INSTITUTE

MAY 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-2020-081 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
LAUREN HUIE-SEVERSKY
Technical Advisor, Computing
and 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)*
MAY 2020	FINAL TECHNICAL REPORT	OCT 2014 – OCT 2019

4. TITLE AND SUBTITLE

PETABLOX: LARGE-SCALE SOFTWARE ANALYSIS AND ANALYTICS USING DATALOG

5a. CONTRACT NUMBER
FA8750-15-2-0009

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Mayur Naik

5d. PROJECT NUMBER
MUSE

5e. TASK NUMBER
BG

5f. WORK UNIT NUMBER
TH

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Georgia Technology Research Institute
400 10th St NW
Atlanta Georgia 30318

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

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

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-081

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

Program analysis encompasses techniques and tools that analyze code to predict program behavior, with important benefits to programmer productivity and software quality. But widespread adoption of this technology is hindered by fundamental challenges in accuracy, scalability, and usability. This project developed foundational techniques and open-source artifacts, specifically: 1) a framework to effectively balance different analysis tradeoffs by combining logical and probabilistic reasoning; 2) methodologies to enable analysis designers leverage massive code corpora to automatically learn features, weights, and probability distributions directly from code; 3) solver techniques for efficient inference and learning; and 4) a system for integrating results of program analysis tools into prevalent developer workflows. The key findings were: 1) the discovery of 100+ new bugs in large widely-used C/C++ programs such as Linux and OpenSSL; 2) accurate analysis of malicious Android apps and enterprise Java programs for information leaks and concurrency safety, respectively; & 3) a demonstration of a continuous integration tool on the Github platform to find bugs in C/C++ projects.

15. SUBJECT TERMS

Program Analysis, Big Code, Constraint Solving, Logic Programming, Datalog, Probabilistic Reasoning, Bayesian Network, Program Synthesis, Machine Learning, Deep Learning, Neural Network

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM E. MCKEEVER**
U	U	U	UU	47	19b. TELEPHONE NUMBER (*Include area code*) **315-330-2897**

Contents

List of Figures

List of Tables

1 Summary

Program analysis encompasses a body of techniques and tools that analyze code to predict program behavior. It has important practical applications in improving programmer productivity and software quality, ranging from intelligent code editing, to finding harmful bugs and security vulnerabilities, to proving the absence of entire classes of semantic errors at compile-time, to proving rich functional correctness properties. However, the undecidable nature of this problem, as well as practical challenges in analyzing complex real-world programs, lead any program analysis to make various approximations. These approximations in turn hinder the accuracy, scalability, and usability of the program analysis at hand—and ultimately inhibit the effective application of this technology on modern codebases.

In the early 2010's, the increasing prevalence of open-source code repositories collectively comprising billions of lines of code posed a tantalizing question: is it possible to leverage them to address long-standing challenges that hinder program analysis technology? DARPA's timely program on "Mining and Understanding Software Enclaves" (MUSE for short) set out to answer this question, dubbing the problem "Big Code". Analogous to how Big Data revolutionized the manner in which information is analyzed, processed, and extracted from extremely complex and large data sets, could Big Code revolutionize "how software is built, debugged, verified, maintained and understood"?

The Petablox project answered this question in the affirmative by demonstrating how to fundamentally extend the prevalent deductive approaches to program analysis with statistical and data-driven modes of reasoning. Specifically, the project developed:

1. A general framework for effectively balancing different tradeoffs in program analyses based on combining logical and probabilistic reasoning [1–4].

2. Methodologies to enable designers of program analyses to automatically learn features, weights, and probability distributions directly from source code [5–11];

3. Solver techniques for efficient and scalable inference and learning with formal guarantees in aspects such as soundness and optimality [5, 12–14]; and

4. A framework for integrating results of program analysis tools into modern software engineering workflows of continuous integration/delivery (CI/CD) [15].

These techniques and artifacts were developed in the setting of Datalog—a declarative, constraint-based, logic programming language commonly used to specify program analyses—and evaluated on large real-world codebases in C, C++, and Java. The main findings of our evaluation are as follows:

1. We discovered over 100 new bugs and vulnerabilities in popular open-source C/C++ programs comprising millions of lines of code, such as the Linux kernel, the OpenSSL encryption library, and the MIT Drake system [16] for robotics applications. Most of these bugs were fixed within a week of reporting.

2. We demonstrated the versatility of our approach on different languages and correctness properties: information leaks in malicious Android apps from the Symantec suite[1],

[1]Developed as part of DARPA's program on Automated Program Analysis for Cybersecurity (APAC).

concurrency bugs in enterprise Java programs (from the Dacapo benchmark suite), and memory safety for C programs (from Linux coreutils).

3. We demonstrated the practicality of the approach by integrating it into realistic static program analysis toolchains including Clang Static Analyzer [17] based on symbolic execution and the Sparrow static analyzer [18] based on abstract interpretation.

4. We demonstrated the feasibility of integrating our approach into continuous integration/delivery workflows on the dominant platform for software development, GitHub, by deploying it as a service through an app on GitHub's marketplace.

We open-sourced all the tools and datasets resulting from the project [19]. We disseminated the research results through peer-reviewed papers as well as through papers that consolidated and summarized them at the following international conferences and workshops: Theory and Applications of Satisfiability Testing (SAT) [12], Computer-Aided Verification (CAV) [20], Verification, Model Checking, and Abstract Interpretation (VMCAI) [21], Machine Learning and Programming Languages (MAPL) [22], and Machine Learning for Programming Languages (ML4PL) [23].

2 Introduction

Program analyses are algorithms that discover a wide range of useful artifacts about programs, including bugs, proofs, and specifications. Existing program analyses are expressed in the form of logical axiom/inference rules that are handcrafted by experts. This logic-based approach provides important benefits. First, logical rules are human-comprehensible, making it convenient for analysis writers to express their domain knowledge. Secondly, the results produced by solving logical rules often come with explanations (e.g., provenance information), making analysis tools easy to use. Last but not least, logical rules enable program analyses to provide rigorous formal guarantees such as soundness.

While logic-based program analyses have achieved remarkable success, they also harbor significant limitations: they cannot handle uncertain knowledge and they lack the ability to learn and adapt. Although the semantics of most programs are deterministic, uncertainties arise in many scenarios due to the undecidable nature of the problem as well as practical issues such as imprecise specifications, missing program parts, imperfect environment models, and many others. Current program analyses rely on experts to manually choose their representations which cannot be changed once they are shipped to end-users. However, the diversity of usage scenarios prevents such fixed representations from addressing the needs of individual end-users. Moreover, the analysis does not improve as it reasons about more programs, and therefore is destined to repeat past mistakes.

We illustrate the challenges facing the conventional logic-based approach to program analysis by means of a real-world example. Figure 1 shows the results of Clang Static Analyzer [17], a state-of-the-art source code analysis tool for finding bugs in C, C++, and Objective-C programs, applied to OpenSSL, a popular open-source encryption library written in C. The vast majority of bug reports produced by the tool are false alarms, which hinders the ability to find real bugs. For instance, when the Heartbleed buffer overread vulnerability (CVE-2014-0160) was detected in OpenSSL in 2014, affecting as many as two-thirds of the world's web servers, the program analysis community conducted a post-mortem: *how could a fairly elementary coding error, traced to a single line of code, be overlooked by program analysis tools in one of the most heavily scrutinized open-source applications?* Andy Chou, cofounder of Coverity [24], a static code analysis company (now part of Synopsys Inc.), reported the following at that time [25]:

> Program analysis is hard and approximations and trade-offs are absolutely mandatory. We've found that the best results come from a combination of advanced algorithms and knowledge of idioms that occur in real-world code. What's particularly insightful is to analyze critical defects for clues that humans might pick up on but are hard to derive from first principles. These patterns form pieces of evidence that can then be generalized and tested empirically to make the analysis "smarter." Our experience is that this is the only way to build analyses that scale to large programs with low false positive rates, yet find critical defects. Many program analysis problems are undecidable in general, and in practice NP-complete problems and severe time/space/accuracy trade-offs crop up everywhere. Giving the analysis intuition and developer "street smarts" is key to providing high quality analysis results. The Heartbleed bug is a perfect example of this.

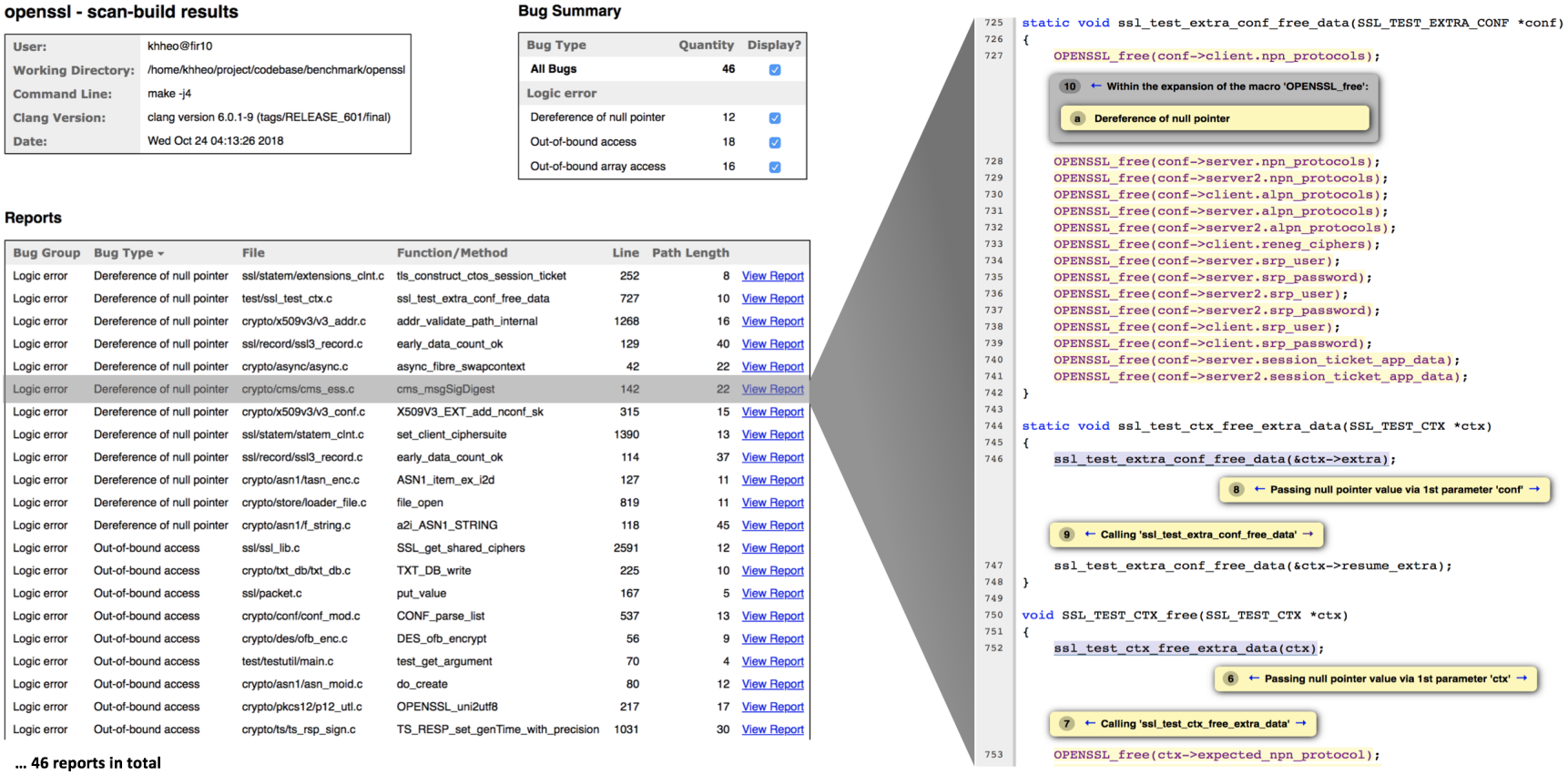


Figure 1: Results of running Clang Static Analyzer on OpenSSL.

A key insight to address the drawbacks of the purely logic-based approach to program analysis lies in drawing from and innovating in methods based on machine learning and artificial intelligence. This insight underlies Petablox—a general, flexible, and scalable program reasoning framework. Petablox does not abdicate the logic-based approach; rather, it fundamentally extends it with statistical and data-driven modes of reasoning. The important features of Petablox are as follows:

- Petablox combines logical and probabilistic reasoning in program analysis, which provides the best of both worlds, such as soundness guarantees on one hand and the ability to adapt on the other.
- Program analyses are usually specified using axiom/inference rules that admit only logical reasoning. Petablox enables incorporating probabilistic reasoning by attaching weights to such rules.
- Petablox includes a set of learning and inference algorithms that achieve scalability and accuracy by leveraging domain insights from program analysis, in particular, provenance information in the form of derivations (i.e., proof trees).
- Petablox adopts and advances a broad range of ML and AI techniques from logical rule learning (e.g. inductive logic programming), probabilistic graphical models (e.g. Markov Logic Networks, Stochastic Logic Programs, and Bayesian Networks), to modern deep learning, reinforcement learning, and transfer learning.
- Petablox enables analyses to automatically adapt to different scenarios and improve over reasoning about similar programs by either mining useful knowledge (e.g. specifications or analysis rules) from codebases or learning from user-provided feedback.

3 Methods, Assumptions and Procedures

This section is organized as follows. We set out by describing a general framework for balancing analysis tradeoffs (Section 3.1). Next, we describe solver techniques we developed for inference and learning that enable the framework to scale while providing formal guarantees on soundness and optimality (Section 3.2). The primary contribution of our framework and solver is a general approach that greatly outperforms even hand-tuned specialized solutions to specific program analysis problems in metrics of accuracy, scalability, and usability.

We then proceed to show how to integrate machine learning into program analysis with application to ranking analysis results either based on ground truth (Section 3.3) or relevance to a code change (Section 3.4). The latter enables our approach to integrate analysis results into CI/CD workflows which are prevalent in modern software development.

Lastly, we focus on two unsupervised learning approaches which are fully automatic and do not require any labels. The first is an approach to statically find Application Programming Interface (API) misuse bugs without needing any specifications (Section 3.6). The second is an end-to-end deep learning approach for verification, synthesis, and bug-finding/repair problems (Section 3.7).

3.1 A Framework for Balancing Analysis Tradeoffs

Constraint-based analysis [26] is a popular approach to program analysis. The core idea underlying this approach is to divide a program analysis task into two separate steps: constraint generation and constraint resolution. The former produces constraints from a given program that constitute a declarative specification of the desired information about the program, while the latter then computes the desired information by solving the constraints. This approach provides many benefits such as separating the analysis specification from the analysis implementation, and allowing to leverage sophisticated off-the-shelf constraint solvers. Due to these benefits, the constraint-based approach has achieved remarkable success, as exemplified by the numerous applications of solvers for SAT and Satisfiability Modulo Theories (SMT).

Existing constraint-based analyses predominantly involve formulating and solving a *decision problem*, which is ill-equipped to handle the tradeoffs involved in software analysis. A natural approach to address this limitation is to extend the decision problem to allow incorporating *optimization objectives*. These objective functions serve to effectively formulate various tradeoffs while preserving the benefits of the constraint-based approach.

Maximum Satisfiability (MaxSAT) [27] is one such optimization extension of the Boolean Satisfiability (SAT) problem. A MaxSAT instance comprises a system of mixed hard and soft clauses, wherein a soft clause is simply a hard clause with a weight. The goal of a (exact) MaxSAT solver is to find a solution that is *sound*, i.e., satisfies all the hard clauses, and *optimal*, i.e., maximizes the sum of the weights of satisfied soft clauses. Thus, hard clauses enable to enforce soundness conditions of a software analysis while soft clauses enable to encode different tradeoffs.

Petablox embodies a MaxSAT based approach to balancing tradeoffs in software analysis. We demonstrated the versatility of this approach using three diverse instantiations that advance the state-of-the-art:

1. *Automated verification* with the goal of finding a cheap yet precise program abstraction for a given analysis [1];

2. *Interactive verification* with the goal of overcoming the incompleteness of a given analysis in a manner that minimizes the user's effort [3]; and

3. *Static bug detection* with the goal of accurately classifying alarms reported by a given analysis by learning from a subset of labeled alarms [2].

We next provide an illustrative overview of the approach using the static bug detection instantiation from [2]. We implemented this instantiation in a tool named Eugene whose architecture is shown in Figure 2. Specifically, we use the example of applying a static datarace detection tool Chord [28] to a real-world multi-threaded Java program, Apache FTP server [29], comprising 150KB of Java bytecode.

Figure 3 shows a code fragment from the program. The `RequestHandler` class is used to handle client connections and an object of this class is created for every incoming connection to the server. The `close`() method is used to clean up and close an open client connection, while the `getRequest`() method is used to access the `m_request` field. Both these methods can be invoked from various components of the program, and thus can be simultaneously

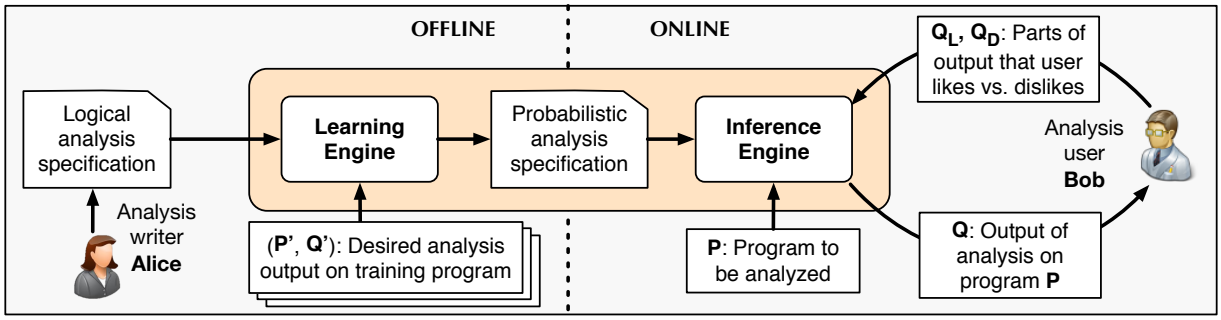


Figure 2: Architecture of the Eugene static bug detection system.

```
1  package org.apache.ftpserver;
2  public class RequestHandler {
3      Socket m_controlSocket;
4      FtpRequestImpl m_request;
5      FtpWriter m_writer;
6      BufferedReader m_reader;
7      boolean m_isConnectionClosed;
8      public FtpRequest getRequest() {
9          return m_request;
10     }
11     public void close() {
12         synchronized(this) {
13             if (m_isConnectionClosed)
14                 return;
15             m_isConnectionClosed = true;
16         }
17         m_request.clear();
18         m_request = null;
19         m_writer.close();
20         m_writer = null;
21         m_reader.close();
22         m_reader = null;
23         m_controlSocket.close();
24         m_controlSocket = null;
25     }
26 }
```

Analysis Relations:

$\mathsf{next}(p_1, p_2)$	(program point p_1 is immediate successor of program point p_2)
$\mathsf{parallel}(p_1, p_2)$	(different threads may reach program points p_1 and p_2 in parallel)
$\mathsf{mayAlias}(p_1, p_2)$	(instructions at program points p_1 and p_2 may access the same memory location, and constitute a possible datarace)
$\mathsf{guarded}(p_1, p_2)$	(at least one common lock guards program points p_1 and p_2)
$\mathsf{race}(p_1, p_2)$	(datarace may occur between different threads while executing the instructions at program points p_1 and p_2)

Analysis Rules:

$$\mathsf{parallel}(p_1, p_2) \wedge \mathsf{next}(p_3, p_1) \Rightarrow \mathsf{parallel}(p_3, p_2) \quad (1)$$

$$\mathsf{parallel}(p_1, p_2) \Rightarrow \mathsf{parallel}(p_2, p_1) \quad (2)$$

$$\begin{pmatrix} \mathsf{parallel}(p_1, p_2) & \wedge \\ \mathsf{mayAlias}(p_1, p_2) & \wedge \\ \neg\mathsf{guarded}(p_1, p_2) & \end{pmatrix} \Rightarrow \mathsf{race}(p_1, p_2) \quad (3)$$

Figure 3: Java code snippet of Apache FTP server (left) and simplified datarace detection analysis specified in Datalog (right).

executed by multiple threads in parallel on the same `RequestHandler` object. To ensure that this parallel execution does not result in any dataraces, the `close`() method uses a boolean flag `m_isConnectionClosed`. If this flag is set, all calls to `close`() return without any further updates. If the flag is not set, then it is first updated to true, followed by execution of the clean-up code (lines 17–24). To avoid dataraces on the flag itself, it is read and updated while holding a lock on the `RequestHandler` object (lines 12–16). All the subsequent code in `close`() is free from dataraces since only the first call to `close`() executes this section. However, note that an actual datarace still exists between the two accesses to field `m_request` on line 9 and line 18.

We motivate the approach in Eugene by contrasting the goals and capabilities of a *writer of an analysis*, such as the datarace detection analysis in Chord, with those of a *user of the analysis*, such as a developer of the Apache FTP server.

The role of the analysis writer. The designer or writer of a static analysis tool, say Alice, strives to develop an analysis that is precise yet scales to real-world programs, and

is widely applicable to a large variety of programs. In the case of Chord, this translates into a datarace detection analysis that is context-sensitive but path-insensitive. This is a common design choice for balancing precision and scalability of static analyses. The analysis in Chord is expressed using Datalog, a declarative logic programming language, and Figure 3 shows a simplified subset of the logical inference rules used by Chord. The actual analysis implementation uses a larger set of more elaborate rules but the rules shown here suffice for the discussion. These rules are used to produce output relations from input relations, where the input relations express known program facts and output relations express the analysis outcome. These rules express the idioms that the analysis writer Alice deems to be the most important for capturing dataraces in Java programs. For example, Rule (1) in Figure 3 conveys that if a pair of program points (p_1, p_2) can execute in parallel, and if program point p_3 is an immediate successor of p_1, then (p_3, p_2) are also likely to happen in parallel. Rule (2) conveys that the parallel relation is symmetric. Via Rule (3), Alice expresses the idiom that only program points not guarded by a common lock can be potentially racing. In particular, if program points (p_1, p_2) can happen in parallel, can access the same memory location, and are not guarded by any common lock, then there is a potential datarace between p_1 and p_2.

The role of the analysis user. The user of a static analysis tool, say Bob, ideally wants the tool to produce exact (i.e., sound and complete) results on his program. This allows him to spend his time on fixing the bugs in the program instead of classifying the reports generated by the tool as spurious or real. In our example, suppose that Bob runs Chord on the Apache FTP server program in Figure 3. Based on the rules in Figure 3, Chord produces the list of datarace reports shown in Figure 4(a). Reports R1–R5 are identified as potential dataraces in the program, whereas for report E1, Chord detects that the accesses to `m_isConnectionClosed` on lines 13 and 15 are guarded by a common lock, and therefore do not constitute a datarace. Typically, the analysis user Bob is well-acquainted with the program being analyzed, but not with the details of underlying analysis itself. In this case, given his familiarity with the program, it is relatively easy for Bob to conclude that the code from line 17–24 in the body of the `close`() method can never be executed by multiple threads in parallel, and thus reports R2–R5 are spurious.

The mismatch between analysis writers and users. The design decisions of the analysis writer Alice have a direct impact on the precision and scalability of the analysis. The datarace analysis in Chord is imprecise for various theoretical and usability reasons.

First, the analysis must scale to large programs. For this reason, it is designed to be path-insensitive and over-approximates the possible thread interleavings. To eliminate spurious reports R2–R5, the analysis would need to only consider feasible thread interleavings by accurately tracking control-flow dependencies across threads. However, such precise analyses do not scale to programs of the size of Apache FTP server, which comprises 130 KLOC.

Scalability concerns aside, relations such as mayAlias are necessarily inexact as the corresponding property is undecidable for Java programs. Chord over-approximates this property by using a context-sensitive but flow-insensitive pointer analysis, resulting in spurious pairs (p_1, p_2) in this relation, which in turn are reported as spurious dataraces.

Third, the analysis writer may lack sufficient information to design a precise analysis, because the program behaviors that the analysis intends to check may be vague or ambiguous. For example, in the case of datarace analysis, real dataraces can be *benign* in that they do not affect the program's correctness [30]. Classifying such reports typically requires knowledge

about the program being analyzed.

Fourth, the program specification can be incomplete. For instance, the datarace in report R1 above could be harmful but impossible to trigger due to timing reasons extrinsic to Apache FTP server, such as the hardware environment.

In short, while the analysis writer Alice can influence the design of the analysis, she cannot foresee every usage scenario or program-specific tweaks that might improve the analysis. Conversely, analysis user Bob understands the program under analysis, and can classify the analysis reports as spurious or real. But he lacks the expertise to suppress the spurious bugs by modifying the underlying analysis based on his intuition and program knowledge.

Closing the gap between analysis writers and users. Our approach aims to empower the analysis user Bob to adjust the underlying analysis as per his demands without involving the analysis writer Alice. Eugene achieves this by automatically incorporating user feedback into the analysis. The user provides feedback in a natural fashion, by "liking" or "disliking" a subset of the analysis reports, and re-running the analysis. For example, when presented with the datarace reports in Figure 4(a), Bob might start inspecting from the first report. This report is valid and Bob might choose to either like or ignore this report. Liking a report conveys that Bob accepts the reported bug as a real one and would like the analysis to generate more similar reports, thereby reinforcing the behavior of the underlying analysis that led to the generation of this report. However, suppose that Bob ignores the first report, but indicates that he dislikes the second report by clicking on the corresponding icon. Re-running Chord after providing this feedback produces the reports shown in Figure 4(b). While the true report R1 is generated in this run as well, all the remaining spurious reports are eliminated. This highlights a key strength of our approach: it not only incorporates user feedback but it also generalizes the feedback to other similar results of the analysis. Reports R2–R5 are correlated and are spurious for the same root cause: the code from line 17–24 in the body of the `close`() method can never be executed by multiple threads in parallel. Bob's feedback on report R2 conveys to the underlying analysis that lines 17 and 18 cannot execute in parallel. Eugene is able to generalize this feedback automatically and conclude that none of the lines from 17–24 can execute in parallel.

3.2 Solver Techniques for Inference and Learning

Enabling the aforementioned instantiations of our framework on real-world programs necessitates solving large MaxSAT instances comprising over 10^{30} clauses in a sound and optimal manner, which is beyond the reach of existing MaxSAT solvers. We first illustrate how such instances arise and then give an overview of the solver techniques developed in Petablox to handle such instances.

Figure 5 shows a subset of the analysis's input and output facts as well as a snippet of the MaxSAT formula constructed for our running example from Figure 3. The input facts are derived from the analyzed program (Apache FTP server) and comprise the next, mayAlias, and guarded relations. In all these relations, the domain of program points is represented by the corresponding line number in the code. Note that all the analysis rules expressed in Figure 3 are hard rules since existing tools like Chord do not accept soft rules. However, we assume that when this analysis is fed to our system, rule (1) is specified to be soft by analysis writer Alice, which captures the fact that the parallel relation is imprecise. Our system

Detected Races	
R1: Race on field org.apache.ftpserver.RequestHandler.m_request	
org.apache.ftpserver.RequestHandler: 9	org.apache.ftpserver.RequestHandler: 18
R2: Race on field org.apache.ftpserver.RequestHandler.m_request	
org.apache.ftpserver.RequestHandler: 17	org.apache.ftpserver.RequestHandler: 18
R3: Race on field org.apache.ftpserver.RequestHandler.m_writer	
org.apache.ftpserver.RequestHandler: 19	org.apache.ftpserver.RequestHandler: 20
R4: Race on field org.apache.ftpserver.RequestHandler.m_reader	
org.apache.ftpserver.RequestHandler: 21	org.apache.ftpserver.RequestHandler: 22
R5: Race on field org.apache.ftpserver.RequestHandler.m_controlSocket	
org.apache.ftpserver.RequestHandler: 23	org.apache.ftpserver.RequestHandler: 24

Eliminated Races	
E1: Race on field org.apache.ftpserver.RequestHandler. m_isConnectionClosed	
org.apache.ftpserver.RequestHandler: 13	org.apache.ftpserver.RequestHandler: 15

(a) Before feedback.

Detected Races	
R1: Race on field org.apache.ftpserver.RequestHandler.m_request	
org.apache.ftpserver.RequestHandler: 9	org.apache.ftpserver.RequestHandler: 18

Eliminated Races	
E1: Race on field org.apache.ftpserver.RequestHandler. m_isConnectionClosed	
org.apache.ftpserver.RequestHandler: 13	org.apache.ftpserver.RequestHandler: 15
E2: Race on field org.apache.ftpserver.RequestHandler.m_request	
org.apache.ftpserver.RequestHandler: 17	org.apache.ftpserver.RequestHandler: 18
E3: Race on field org.apache.ftpserver.RequestHandler.m_writer	
org.apache.ftpserver.RequestHandler: 19	org.apache.ftpserver.RequestHandler: 20
E4: Race on field org.apache.ftpserver.RequestHandler.m_reader	
org.apache.ftpserver.RequestHandler: 21	org.apache.ftpserver.RequestHandler: 22
E5: Race on field org.apache.ftpserver.RequestHandler.m_controlSocket	
org.apache.ftpserver.RequestHandler: 23	org.apache.ftpserver.RequestHandler: 24

(b) After feedback.

Figure 4: Race reports produced for Apache FTP server by Eugene. Each report specifies the field involved in the race, and line numbers of the program points with the racing accesses. The user feedback is to "dislike" report R2.

Input facts:

$\mathsf{next}(18,17)$ $\mathsf{mayAlias}(18,17)$ $\mathsf{guarded}(13,13)$
$\mathsf{next}(19,18)$ $\mathsf{mayAlias}(20,19)$ $\mathsf{guarded}(15,15)$
$\mathsf{next}(20,19)$ $\mathsf{mayAlias}(22,21)$ $\mathsf{guarded}(15,13)$ ...

MaxSAT formula:

$$
\begin{array}{ll}
w_1 : (\neg\mathsf{parallel}(17,17) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,17)) & \wedge \\
\quad (\neg\mathsf{parallel}(18,17) \vee \mathsf{parallel}(17,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(17,18) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,18)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,18) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,18)) & \wedge \\
\quad (\neg\mathsf{parallel}(19,18) \vee \mathsf{parallel}(18,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(18,19) \vee \neg\mathsf{next}(19,18) \vee \mathsf{parallel}(19,19)) & \wedge \\
w_1 : (\neg\mathsf{parallel}(19,19) \vee \neg\mathsf{next}(20,19) \vee \mathsf{parallel}(20,19)) & \wedge \\
\quad \left(\begin{array}{lll} \neg\mathsf{parallel}(18,17) & \vee & \neg\mathsf{mayAlias}(18,17) \quad \vee \\ \mathsf{guarded}(18,17) & \vee & \mathsf{race}(18,17) \end{array} \right) & \wedge \\
\quad \left(\begin{array}{lll} \neg\mathsf{parallel}(20,19) & \vee & \neg\mathsf{mayAlias}(20,19) \quad \vee \\ \mathsf{guarded}(20,19) & \vee & \mathsf{race}(20,19) \end{array} \right) & \wedge \\
\boxed{w_2 : \neg\mathsf{race}(18,17)} & \wedge \ldots
\end{array}
$$

Output facts (before feedback):

$\mathsf{parallel}(18,9)$ $\mathsf{parallel}(20,19)$ $\mathsf{race}(18,9)$ $\mathsf{race}(20,19)$ ...
$\mathsf{parallel}(18,17)$ $\mathsf{parallel}(22,21)$ $\mathsf{race}(18,17)$ $\mathsf{race}(22,21)$

Output facts (after feedback):

$\mathsf{parallel}(18,9)$ $\mathsf{race}(18,9)$ ...

Figure 5: MaxSAT-based formulation for static bug detection example.

automatically learns the weight of this rule to be w_1 from training data. Given these input facts and rules, the MaxSAT problem to be solved is generated by grounding the analysis rules, and a snippet of the constructed MaxSAT formula is shown in Figure 5. Ignoring the clause enclosed in the box, solving this MaxSAT formula (without the boxed clause) yields output facts, a subset of which is shown under "Output facts (before feedback)" in Figure 5. The output includes multiple spurious races like $\mathsf{race}(18,17)$, $\mathsf{race}(20,19)$, and $\mathsf{race}(22,21)$.

As described in Section 3.1, when analysis user Bob provides feedback that $\mathsf{race}(18,17)$ is spurious, our approach suppresses all spurious races while retaining the real race $\mathsf{race}(18,9)$. It achieves this by incorporating the user feedback itself as a soft rule, represented by the boxed clause $\neg\mathsf{race}(18,17)$ in Figure 5. The weight for such user feedback is also learned during the training phase. Assuming the weight w_2 of the feedback clause is higher than the weight w_1 of rule (1)—a reasonable choice that emphasizes Bob's preferences over Alice's assumptions—the MaxSAT semantics ensures that the solver prefers violating rule (1) over violating the feedback clause. When the MaxSAT formula (with the boxed clause) in Figure 5 is then fed to the solver, the output solution violates the clause $w_1 : (\neg\mathsf{parallel}(17,17) \vee \neg\mathsf{next}(18,17) \vee \mathsf{parallel}(18,17))$ and does not produce facts $\mathsf{parallel}(18,17)$ and $\mathsf{race}(18,17)$ in the output. Further, all the facts that are dependent on $\mathsf{parallel}(18,17)$ are not produced either. (This is due to implicit soft rules that negate each output relation, such as $w_0 : \neg\mathsf{parallel}(p_1,p_2)$ where $w_0 < w_1$, in order to obtain the least solution.) This implies that facts like $\mathsf{parallel}(19,18)$, $\mathsf{parallel}(20,19)$,

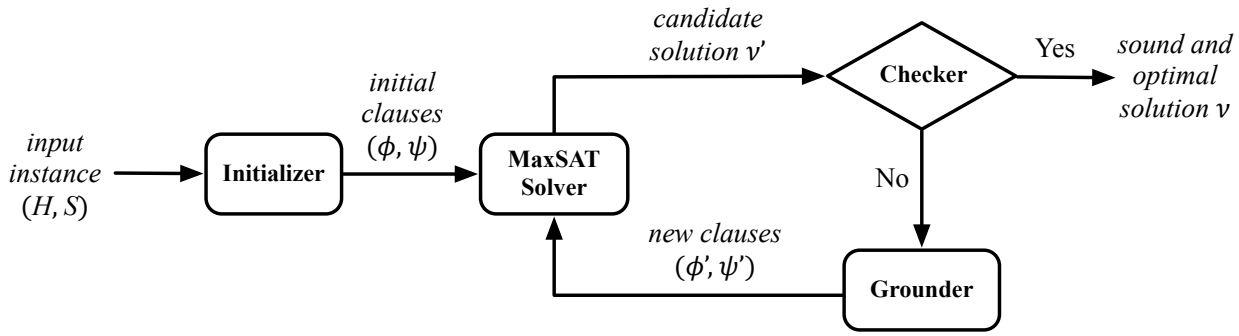


Figure 6: Architecture of our lazy grounding solver for solving large MaxSAT instances. It scales by iteratively expanding a workset comprising a subset of clauses in the input MaxSAT instance. Our bottom-up and top-down grounding techniques, and many others in the literature, are instances of this framework.

parallel$(22, 21)$ are not produced, and therefore race$(20, 19)$ and race$(22, 21)$ are also suppressed. Thus, our approach is able to generalize based on user feedback.

We developed a series of solver techniques to solve MaxSAT instances in a manner that is scalable while providing formal guarantees on soundness (respecting all hard rules) and optimal (maximizing the weights of satisfied soft rules). The key insight underlying our approach to scale is *lazy grounding*: iteratively expanding a subset of clauses while providing guarantees. We next elaborate upon the general architecture (Section 3.2.1) followed by two instantiations of it: bottom-up grounding (Section 3.2.2) and top-down grouding (Section 3.2.3). Finally, we discuss an incremental solving approach (Section 3.2.4).

3.2.1 Lazy Grounding Framework

Figure 6 shows the architecture of our lazy grounding framework. The framework, called VOLT, is described in [12]. In each iteration, VOLT only grounds a subset of clauses in the MaxSAT problem, and solves it using an off-the-shelf MaxSAT solver. VOLT provides a common ground to compare and contrast different lazy grounding approaches for solving large MaxSAT instances.

We cast four diverse approaches from the literature on information retrieval and program analysis as instances of VOLT:

- SoftCegar [31], which grounds all the soft clauses upfront but lazily grounds the hard clauses. In each iteration, this approach grounds all the hard clauses violated by the current solution.

- Cutting Plane Inference (CPI) [32, 33], which is lazier than SoftCegar and grounds no clauses upfront. In each iteration, both hard and soft constraints are checked for violations, and any violated clauses are grounded. The algorithm terminates when no new constraints are violated.

- A common approach, used in statistical relational learning tools like Alchemy [34] and Tuffy [35], that relies on the observation that most ground facts are false in the final

solution, and thereby most clauses are trivially true (since most clauses are Horn in these applications). An active ground fact is one that has a value of true. In each iteration, the clauses grounded are those containing at least one active fact as per the current solution. Initially, only the input facts are considered active. The approach terminates after a fixed number of iterations or after the weight of the satisfied clauses is greater than a target weight.

- The AbsRefine approach [1] from the automated verification instantiation in Section 3.1, which tackles a central problem in program analysis of efficiently finding a program abstraction that keeps only information relevant for proving properties of interest.

Furthermore, we demonstrated the effectiveness of VOLT under different state-of-the-art MaxSAT solvers that were available from the top performers in Random, Crafted and Industrial categories of the 9th Max-SAT Evaluation [27]. In particular, the solvers we used were: CCLS2akms [36, 37], Eva500a [38], MaxHS [39], wmifumax [40], MSCG [41, 42] and WPM-2014-co [43].

Our evaluation results indicated that the MaxSAT instances generated by VOLT are many orders of magnitude smaller than the full MaxSAT instances. It was evident that any approach attempting to tackle problems of this scale needs to employ lazy techniques for solving such instances. We made the MaxSAT instances generated in our evaluation publicly available to facilitate future research.

3.2.2 Bottom-up Grounding

In [13], we developed a bottom-up grounding approach in the VOLT framework that strikes a balance between eager grounding and lazy grounding. Our key underlying idea comprises two complementary optimizations: eagerly exploiting proofs and lazily refuting counterexamples. To eagerly exploit proofs, our algorithm uses an efficient procedure to upfront ground constraints that will necessarily be grounded during the iterative process. On the other hand, to lazily refute counterexamples, our algorithm uses an efficient procedure to check for violations of constraints by the candidate solution to the set of ground constraints in each iteration, terminating the iterative process in the absence of violations. We showed that our algorithm achieves significant speedup over existing approaches without sacrificing soundness for real-world applications from program analysis and information retrieval.

3.2.3 Top-Down Grounding

In [14], we proposed a new optimization problem "Q-MaxSAT", an extension of the MaxSAT problem. In contrast to MaxSAT, which aims to find an assignment to all variables in the formula, Q-MaxSAT computes an assignment to a desired subset of variables (or queries) in the formula. Many problems in diverse domains such as program reasoning, information retrieval, and mathematical optimization can be naturally encoded as Q-MaxSAT instances (see Figure 7).

We also developed an iterative algorithm for solving Q-MaxSAT. In each iteration, the algorithm solves a subproblem that is relevant to the queries, and applies a novel technique to check whether the partial assignment found is a solution to the Q-MaxSAT problem. If

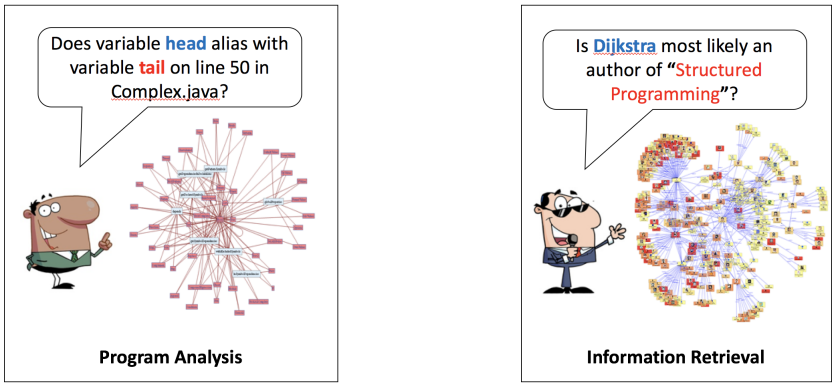


Figure 7: Example queries from two different domains motivating Q-MaxSAT.

the check fails, the algorithm grows the subproblem with a new set of clauses identified as relevant to the queries. Our empirical evaluation showed that our Q-MaxSAT approach achieves significant improvements in runtime and memory consumption over conventional MaxSAT solvers on several Q-MaxSAT instances in real-world applications from program analysis and information retrieval.

3.2.4 Incremental Solving

A special but common scenario concerns applications which pose a *sequence* of *similar* large MaxSAT instances. For example, many applications involve a sequence of small updates to a large instance (e.g., verification via abstraction refinement [1] or user interaction [2]). Alternatively, MaxSAT-based solvers pose such sequences in order to scale to ever larger instances (e.g., using bottom-up [13] or top-down [14] methods described above) or more expressive theories (e.g., MaxSMT [44] and Markov Logic Networks [12, 45]). Instead of solving each instance in the sequence from scratch, it is desirable to improve the efficiency of MaxSAT solvers by reusing results computed across invocations on such instances.

In [5], we developed an incremental approach that targets an especially common case in which the sequence of MaxSAT instances is constructed by *adding* hard or soft clauses. Moreover, the new clauses are determined by the solution to the previous instance. We adpated an unsat core-guided algorithm [46] which forms the basis of many popular MaxSAT solvers. This algorithm solves a single MaxSAT instance by solving a sequence of SAT instances until the underlying SAT solver finds a satisfying solution. Each SAT instance is constructed using the unsat cores discovered in previous SAT instances. Since adding clauses to the current MaxSAT instance does not invalidate existing unsat cores, a compelling idea to improve the performance of solving the resulting MaxSAT instance is to reuse the existing unsat cores.

Surprisingly, however, we observed that a naive implementation of this idea can fail to yield performance benefits or, even worse, sharply curtain them. To address this challenge, we propose a hybrid solving framework that alternates between the incremental algorithm and its non-incremental version. In each iteration, our framework checks whether the current instance may potentially benefit from reusing the cores learnt on previous instances. If the check succeeds, it applies the incremental algorithm by reusing such cores. Otherwise, it discards

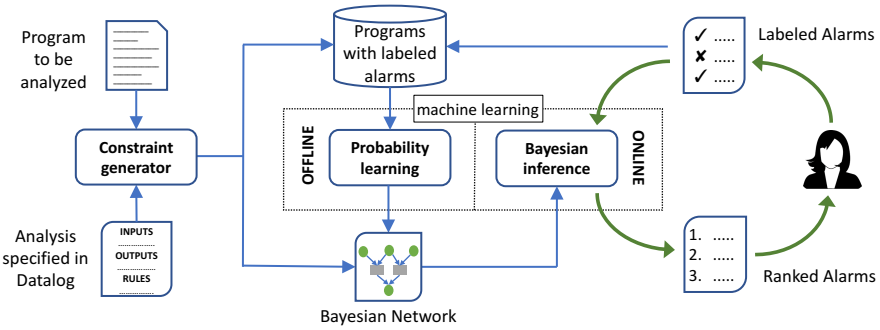


Figure 8: Architecture and workflow of the Bingo alarm ranking system.

the cores learnt thus far and applies the non-incremental algorithm. We implemented this approach in the Open-WBO MaxSAT solver [47] and evaluated it on 74 sequences generated from diverse applications in program analysis and information retrieval. Together, these sequences contain 669 MaxSAT instances, with an average of 10 million clauses per instance. Our evaluation showed that our approach outperforms the baseline approaches significantly: it yielded an average speedup of 1.8× per sequence over the non-incremental approach, and it solved 19 more sequences than the naively-incremental approach.

3.3 Ranking Analysis Alarms via Bayesian Inference

A limitation of the above constrained optimization approach is that it casts static bug finding as a *classification problem* of true alarms (real bugs) vs. false alarms (false positives), as opposed to a *ranking problem*. In follow-up work, we formulated a continuous optimization approach [4] that extends the discrete semantics of Datalog to the continuous semantics of Bayesian networks. We thereby reduce the ranking problem to Bayesian inference. This also enables to effectively incorporate user feedback into program reasoning: user-provided labels on top-ranked facts serve as evidence to condition upon in re-ranking other deduced facts. Figure 8 depicts the architecture of this approach implemented in a tool named Bingo for arbitrary analyses specified in Datalog. Bingo enabled a reduction of over 50% in alarm inspection effort for a variety of static analysis problems—finding malicious information leaks in Android apps, finding concurrency bugs in large object-oriented programs in Java, and finding memory safety vulnerabilities in low-level systems programs in C.

We next illustrate Bingo's approach using the example from Figure 3. Consider the following rule from the static datarace detection analysis in Datalog:

$$\mathsf{parallel}(p_1, p_2) \ \wedge \ \mathsf{next}(p_3, p_1) \implies \mathsf{parallel}(p_3, p_2)$$

This rule is sound but incomplete. The incompleteness is evident by grounding it on the following code fragment:

```
Thread 1:                              Thread 2:
if (num_threads == 1) // p1            x = x + 1 // p3
    x = x + 1         // p2
```

parallel(p1, p2)	next(p3, p1)	P(parallel(p3, p2) | parallel(p1, p2), next(p3, p1))
True	True	0.9
True	False	0
False	True	0
False	False	0

Figure 9: Conditional probability table for the incomplete analysis rule $\mathsf{parallel}(p_1, p_2) \wedge \mathsf{next}(p_3, p_1) \implies \mathsf{parallel}(p_3, p_2)$ with weight 0.9.

Although the antecedents of this rule, parallel(p1,p3) and next(p2,p1) hold, the consequent parallel(p2,p2) does not hold because of the fact that Thread 1 only executes statement `p2` when there is a single thread, indicated by the conditional `num_threads == 1`. While the rule could be enhanced to handle such cases, however, in the limit this problem is undecidable and therefore we will necessarily have analysis rules that are unsound or incomplete. Such noisy rules in turn result in false alarms.

The key insight underlying Bingo is to quantify the inaccuracy in such analysis rules by treating them probabilistically. For this purpose, we associate a weight w with each such analysis rule—a real number between 0 and 1 which can be learnt from labeled data—and attach a conditional probability distribution (CPD) with the rule, as shown in Figure 9. This probabilistic rule is now interpreted as follows: whenever antecedents $\mathsf{parallel}(p_1, p_2)$ and $\mathsf{next}(p_3, p_1)$ hold, the consequent $\mathsf{parallel}(p_3, p_2)$ is true with probability 0.9.

By attaching CPDs to each node in the derivation graphs of alarms produced by the analysis, our approach views the derivation graph as a Bayesian network. Specifically, Bingo performs marginal inference on the network to associate each alarm with the probability, or belief, that it is a true datarace. The alarms are then reported to the user ranked by the belief scores as shown in Figure 10 (a).

The user now inspects the top-ranked report R2 and classifies it as a false alarm. The key idea underlying Bingo is that generalizing from feedback is conditioning on evidence. By replacing the prior belief $Pr(a)$, for each alarm a, with the posterior belief, $Pr(a|\neg R2)$, Bingo effectively propagates the user feedback to the remaining conclusions of the analysis. This results in the updated list of alarms shown in Figure 10 (b). Observe that the beliefs in the closely related alarms R3, R4, and R5 drop (e.g. from 0.53 to 0.28 for R3), while the belief in the unrelated alarm R1 remains unchanged at 0.30. As a result, the entire family of false alarms drops in the ranking, so that the only true datarace R1 is now at the top.

In summary, Bingo integrates machine learning into the analysis computation itself, in contrast to existing approaches which typically apply machine learning to post-process analysis results. In doing so, Bingo is more generally applicable and generalizes user feedback more effectively.

Confidence	Detected Races	
0.81	**R2**: Race on field org.apache.ftpserver.RequestHandler.request	
	org.apache.ftpserver.RequestHandler:17	org.apache.ftpserver.RequestHandler:18
0.53	**R3**: Race on field org.apache.ftpserver.RequestHandler.writer	
	org.apache.ftpserver.RequestHandler:19	org.apache.ftpserver.RequestHandler:20
0.35	**R4**: Race on field org.apache.ftpserver.RequestHandler.reader	
	org.apache.ftpserver.RequestHandler:21	org.apache.ftpserver.RequestHandler:22
0.30	**R1**: Race on field org.apache.ftpserver.RequestHandler.request	
	org.apache.ftpserver.RequestHandler:9	org.apache.ftpserver.RequestHandler:18
0.23	**R5**: Race on field org.apache.ftpserver.RequestHandler.controlSocket	
	org.apache.ftpserver.RequestHandler:23	org.apache.ftpserver.RequestHandler:24

(a) Before feedback.

Confidence	Detected Races	
0.81 **0**	**R2:** Race on field org.apache.ftpserver.RequestHandler.request	
	org.apache.ftpserver.RequestHandler:17	org.apache.ftpserver.RequestHandler:18
0.53 **0.28**	**R3:** Race on field org.apache.ftpserver.RequestHandler.writer	
	org.apache.ftpserver.RequestHandler:19	org.apache.ftpserver.RequestHandler:20
0.35 **0.18**	**R4:** Race on field org.apache.ftpserver.RequestHandler.reader	
	org.apache.ftpserver.RequestHandler:21	org.apache.ftpserver.RequestHandler:22
0.30 **0.30**	**R1:** Race on field org.apache.ftpserver.RequestHandler.request	
	org.apache.ftpserver.RequestHandler:9	org.apache.ftpserver.RequestHandler:18
0.23 **0.12**	**R5:** Race on field org.apache.ftpserver.RequestHandler.controlSocket	
	org.apache.ftpserver.RequestHandler:23	org.apache.ftpserver.RequestHandler:24

P(R*i* | ¬ R2)

(b) After feedback.

Figure 10: Race reports produced for Apache FTP server by Bingo. Each report specifies the field involved in the race, and line numbers of the program points with the racing accesses. The user feedback is to "dislike" top-ranked false alarm R1.

```
1 - #define CMP_SIZE 529200
2 #define HEADER_SIZE 44
3 + int shift_secs;
4
5 void read_value_long(FILE *file, long *val) {
6     char buf[5];
7     fread(buf, 1, 4, file); // Input Source
8     buf[4] = 0;
9     *val = (buf[3] << 24) | (buf[2] << 16) | (buf[1] << 8) | buf[0];
10 }
11
12 wave_info *new_wave_info(char *filename) {
13     wave_info *info;
14     FILE *f;
15
16     info = malloc(sizeof(wave_info));
17     f = fopen(filename);
18     read_value_long(f, info->header_size);
19     read_value_long(f, info->data_size);
20     return info;
21 }
22
23 void trim_main(char *filename) {
24     wave_info *info;
25     info = new_wave_info(filename);
26     long header_size;
27     char *header;
28
29     header_size = min(info->header_size, HEADER_SIZE);
30     header = malloc(header_size * sizeof(char)); // Alarm(30)
31     /* trim a wave file */
32 }
```

```
33 void cmp_main(char *filename1, char *filename2) {
34     wave_info *info1, *info2;
35     long bytes;
36     char *buf;
37
38     info1 = new_wave_info(filename1);
39     info2 = new_wave_info(filename2);
40
41 -    bytes = min(min(info1->data_size, info2->data_size), CMP_SIZE);
42 +    cmp_size = shift_secs * info1->rate; // Integer Overflow
43 +    bytes = min(min(info1->data_size, info2->data_size), cmp_size);
44
45     buf = malloc(2 * bytes * sizeof(char)); // Alarm(45)
46     /* compare two wave files */
47 }
48
49 int main(int argc, char *argv) {
50     int c ;
51     while ((c = getopt(argc, argv, "c:f:ls")) != -1) {
52         switch (c) {
53         case 'c':
54 +           shift_secs = atoi(optarg); // Input Source
55             cmp_main(argv[optind], argv[optind + 1]);
56             break;
57         case 't':
58             trim_main(argv[optind]);
59             break;
60     }
61     return 0;
62 }
```

Figure 11: An example of a code change between two versions of a C program `shntool`.

3.4 Integrating Analyses into CI/CD Environments

Programs often evolve by continuously integrating changes from multiple programmers. The effective adoption of program analysis tools in this CI/CD setting is hindered by the need to only report alarms relevant to a particular program change. The idea of ranking analysis alarms via Bayesian inference can be used in this setting as well. However, the ranking criterion is different: instead of ranking alarms by ground truth, they are ranked by relevance to the code change.

We illustrate this scenario by means of a real-world example. Figure 11 shows an excerpt from the audio file processing utility `shntool`, and highlights changes made to the code between versions 3.0.4 and 3.0.5. Lines preceded by a '+' indicate code which has been added, and lines preceded by a '-'indicate code which has been removed from the new version. The integer overflow analysis in the Sparrow static analyzer [18] reports two alarms in each version of this code snippet, which we describe next.

The first alarm, reported at line 30, concerns the command line option "t". This program feature trims periods of silence from the ends of an audio file. The program reads unsanitized data into the field info−>header_size at line 25, and allocates a buffer of proportional size at line 30. Sparrow observes this data flow, concludes that the multiplication could overflow, and subsequently raises an alarm at the allocation site. However, this data has been sanitized at line 29, so that the expression header_size $*$ sizeof(char) cannot overflow. This is therefore a false alarm in both versions. We will refer to this alarm as $Alarm(30)$.

The second alarm is reported at line 45, and is triggered by the command line option "c". This program feature compares the contents of two audio files. The first version has

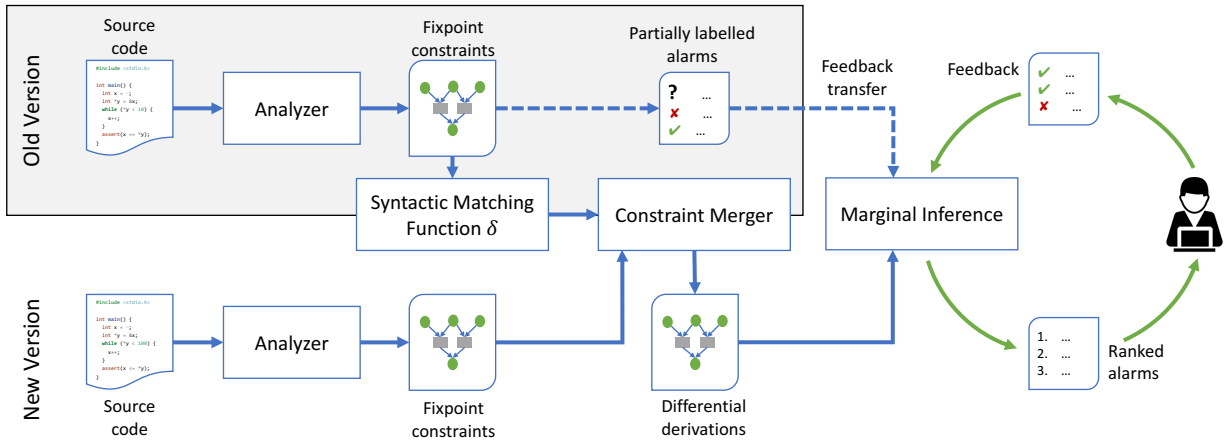


Figure 12: Architecture and workflow of the Drake alarm ranking system.

source-sink flows from the untrusted fields info1−>data_size and info2−>data_size, but this is a false alarm since the value of bytes cannot be larger than CMP_SIZE. On the other hand, the new version of the program includes an option to offset the contents of one file by shift_secs seconds. This value is used without sanitization to compute cmp_size, leading to a possible integer overflow at line 42, which would then result in a buffer of unexpected size being allocated at line 45. Thus, while Sparrow raises an alarm at the *same allocation site* for both versions of the program, which we will call $Alarm(45)$, this is a false alarm in the old version but a real bug in the new version.

The central question we sought to answer in this work can be restated as follows: *How do we alert the user to the possibility of a bug at line 45, while not forcing them to inspect all the alarms of the "batch mode" analysis, including that at line 30?*

We developed a general solution for arbitrary analyses specified in Datalog. We implemented it in a framework called Drake whose architecture is depicted in Figure 12. First, the system extracts static analysis results from both the old and new versions of the program. Since these results are described in terms of syntactic entities (such as source locations) from different versions of the program, it uses a syntactic matching function to translate the old version of the constraints into the setting of the new program. Drake then merges the two sets of constraints into a unified *differential derivation graph*. These differential derivations highlight the relevance of the changed code to the static analysis alarms. Moreover, the differential derivation graph also enables us to perform marginal inference with the feedback from the user as well as previously labeled alarms from the old version.

On a suite of 10 popular GNU utility C programs comprising 13K-112K lines of code each, Drake could discover all 26 bugs, including four known CVEs, while requiring the developer to inspect only 30 alarms on average per program, whereas the baseline batch approach required inspecting 560 alarms on average.

3.5 Synthesizing Interpretable Analyses from Data

A key limitation to our approach thus far lies in the need for accurate analysis rules which are hand-crafted by human experts. Poorly designed rules hinder metrics such as accuracy,

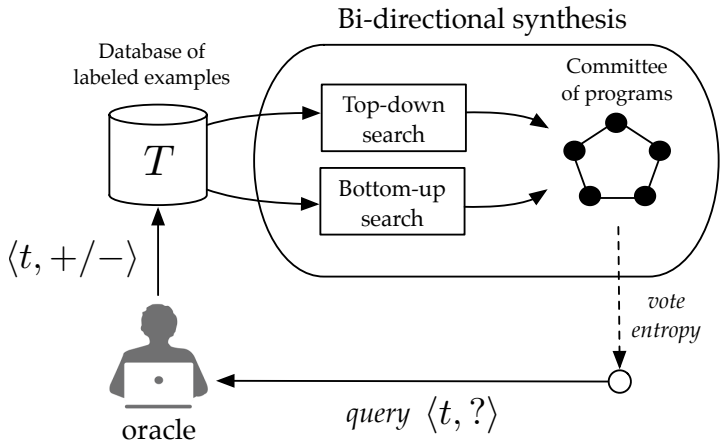


Figure 13: Architecture of the ALPS tool for synthesizing Datalog programs.

scalability, and usability despite the use of machine learning. After all, these rules are akin to the "features" in machine learning: poor features inhibit benefits such as trainability and generalization that machine learning aims to provide.

In a series of works [6, 7, 48, 49], we developed scalable algorithms for automatically synthesizing logical rules in the form of Datalog programs from input-output data. Besides being interpretable, logical rules are amenable to efficient discovery via a variety of symbolic and numeric reasoning algorithms, both in terms of time and data efficiency. This problem has been studied in the field of Inductive Logic Programming (ILP) for three decades; however, existing approaches cannot effectively handle expressive features (e.g. recursion and invented predicates), do not provide formal guarantees in terms of fitting the given data, and require syntactic bias mechanisms in the form of meta rules or templates. We next describe the synthesis techniques we developed to address these challenges.

3.5.1 Combinatorial Search Approach

In [6], we proposed a combinatorial search approach to synthesize Datalog programs from input-output data. Our key insight is that such programs in practice comprise rules that have similar latent syntactic structure. Our approach exploits this insight via the *Syntax-Guided program Synthesis* (SyGuS) paradigm [50], wherein the syntactic structure of the target class of programs is leveraged to efficiently traverse the hypothesis space of programs. For this purpose, our approach must address three key challenges: i) capture syntactic structure effectively, ii) minimize the number of examples needed, and iii) explore the search space efficiently We next elaborate upon each of these objectives. Figure 13 shows the overall architecture of this approach implemented in a tool named ALPS.

To capture the syntactic structure of rule-based programs, we use *meta-rules* [51]—templates that describe a set of possible rules that can appear in a program. The key challenge is to obtain a set of meta-rules that is *general enough to capture useful programs but specific enough to enable efficient synthesis*. We proposed a novel approach to systematically generate meta-rules, taking advantage of domain knowledge.

To minimize the number of examples needed, ALPS aims to ask an oracle a small number

of queries concerning the expected output on a given input. The oracle need only answer with *yes* or *no*, rather than crafting elaborate examples and supplying them to the synthesizer. ALPS uses an *active learning* technique, called *query by committee* (QBC) [52, 53], to pick an example that can prune the search space the most. In our setting, QBC takes as input a committee formed by a set of consistent programs, and returns an example on which the committee *disagrees the most*—the most controversial example. ALPs then prunes the programs that disagree with the given label and repeat the process. However, it is infeasible to apply QBC on the entire search space, which is prohibitively large.

To explore the search space efficiently and overcome the challenge in using QBC, APLS uses a *bidirectional synthesis strategy* to maintain the *most-general* and *most-specific* programs that are consistent with the given examples [54]. Intuitively, the most-general and most-specific programs (defined through logical entailment) form a representative set of the search space, allowing us to preserve exactness of the search. Moreover, this set is much smaller than the size of the search space, making it an ideal *committee*. To incrementally update the search space as examples are labeled, we defined efficient *top-down and bottom-up refinement operators* that are guided by the given set of meta-rules.

We evaluated ALPS on a suite of 34 benchmarks from three domains—program analysis, knowledge discovery, and database queries. The evaluation showed that ALPS could successfully synthesize 33 of these benchmarks, and outperformed the state-of-the-art tools Metagol [55] and Zaatar [5], which could synthesize only up to 10 of the benchmarks.

3.5.2 Numerical Relaxation Approach

In follow-up work [7], we took a fundamentally different approach to the problem of synthesizing Datalog programs. Inspired by the success of numerical methods in machine learning and other large scale optimization problems, and of the strategy of relaxation in solving combinatorial problems such as integer linear programming, we extended the classical discrete semantics of Datalog to a continuous setting named Difflog, where each rule is annotated with a real-valued weight, and the program computes a numerical value for each output tuple. This step can be viewed as an instantiation of the general K-relation framework for database provenance [56] with the Viterbi semiring being chosen as the underlying space K of provenance tokens. We then formalize the program synthesis problem as that of selecting a subset of target rules from a large set of candidate rules, and thereby uniformly capture various methods of inducing syntactic bias, including SyGuS [50], and template rules in meta-interpretive learning [55].

The synthesis problem thus reduces to that of finding the values of the rule weights which result in the best agreement between the computed values of the output tuples and their specified values (1 for desirable and 0 for undesirable tuples). The fundamental NP-hardness of the underlying decision problem manifests as a complex search surface, with local minima and saddle points. To overcome these challenges, we devised a hybrid optimization algorithm which combines Newton's root-finding method with periodic invocations of a simulated annealing search. Finally, when the optimum value is reached, connections between the semantics of Difflog and Datalog enable the recovery of a classical discrete-valued Datalog program from the continuous-valued optimum produced by the optimization algorithm.

A particularly appealing aspect of relaxation-based synthesis is the randomness caused by

the choice of the starting position and of subsequent Monte Carlo iterations. This manifests both as a variety of different solutions to the same problem, and as a variation in running times. Running many search instances in parallel therefore enables stochastic speedup of the synthesis process, and allows us to leverage compute clusters in a way that is fundamentally impossible with deterministic approaches. We implemented Difflog and evaluated it on a suite of 34 benchmark programs from the literature. We demonstrated significant improvements over the state-of-the-art, even while synthesizing complex programs with recursion, invented predicates, and relations of arbitrary arity.

3.5.3 Provenance-Guided Search Approach

In this work, published in [49], we developed a provenance-guided search approach to synthesize Datalog programs. Query provenance [56,57] has emerged as a powerful mechanism to enable a variety of tools that require meta-reasoning over Datalog programs, including debugging query results [58], counterexample-guided abstraction refinement (CEGAR) in static analyses [1], and confidence computation in uncertain and probabilistic databases [59]. The central insight of this work was that provenance can also play a key role in program synthesis.

In most guess-and-check approaches to program synthesis, such as counterexample-guided inductive synthesis (CEGIS) [60], the main challenge lies in identifying the reason for the failure of a particular candidate solution. Formal models of query provenance form the ideal template to structure such reasoning about failures.

Our approach, depicted in Figure 14, follows the CEGIS paradigm: in each iteration, a SAT solver generates a candidate Datalog program, and a Datalog solver evaluates the generated program to determine whether it meets the desired input-output specification. In this context, our approach can also be regarded as an instantiation of the classic Davis-Putnam-Logemann-Loveland algorithm DPLL(T) for automated theorem proving [61], with T being the theory of least fixed points.

A candidate Datalog program can fail to meet the desired specification in one of two ways: either by producing an *undesirable* output tuple or by failing to produce a *desirable* output tuple. Our approach handles both cases via additional constraints to the SAT solver in the next CEGIS iteration. Constraints encoding an erroneous derivation of an undesirable output tuple can be obtained directly via classical models of "*why*" provenance. However, reflecting on the non-derivation of a desirable output tuple leads to difficult ontological questions. We propose two new techniques to address this problem of "*why-not*" provenance: the first is a version of the delta-debugging algorithm that significantly strengthens the constraints from a non-derivation failure, and the second is a notion of co-provenance which identifies necessary constraints before the occurrence of a non-derivation failure.

In summary, our approach leverages provenance information from the Datalog solver in order to constrain the SAT solver in each CEGIS iteration. Conceptually, it constitutes a new approach to boolean function learning, where the target concept is the formula which encodes exactly the set of solutions to the synthesis problem. In practice, this provenance-guided approach is central to scaling synthesis and reducing variability in synthesis time—problems that plague existing approaches due to a large number of non-deterministic choices in the search process.

We implemented our approach in a tool called Prosynth and demonstrated that it

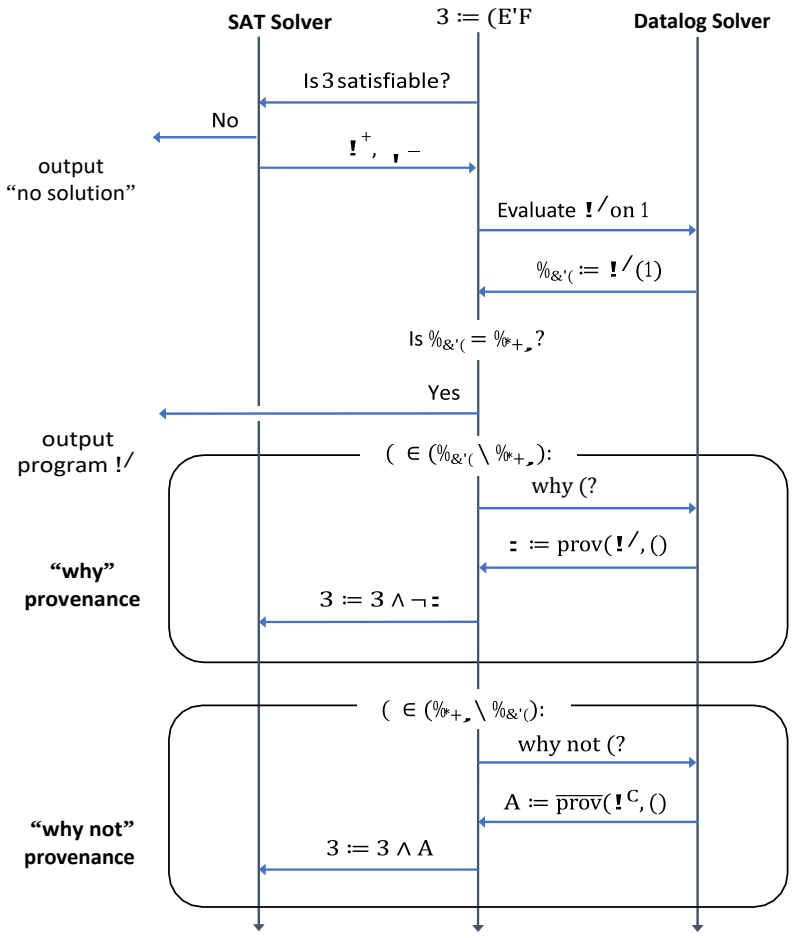


Figure 14: Message sequence chart depicting the interaction between the SAT solver and the Datalog solver in each CEGIS iteration of Prosynth.

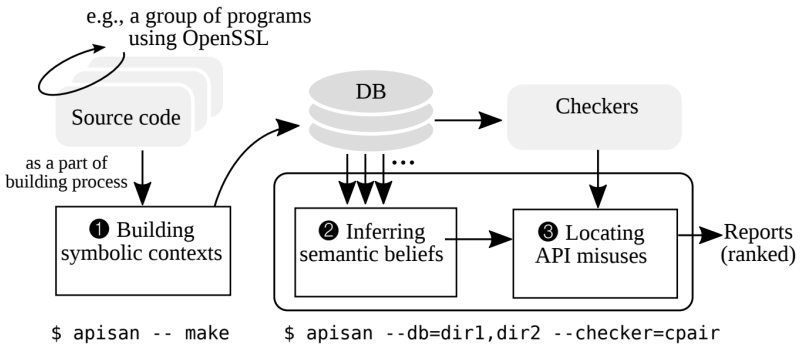


Figure 15: Architecture and workflow of the APISan system.

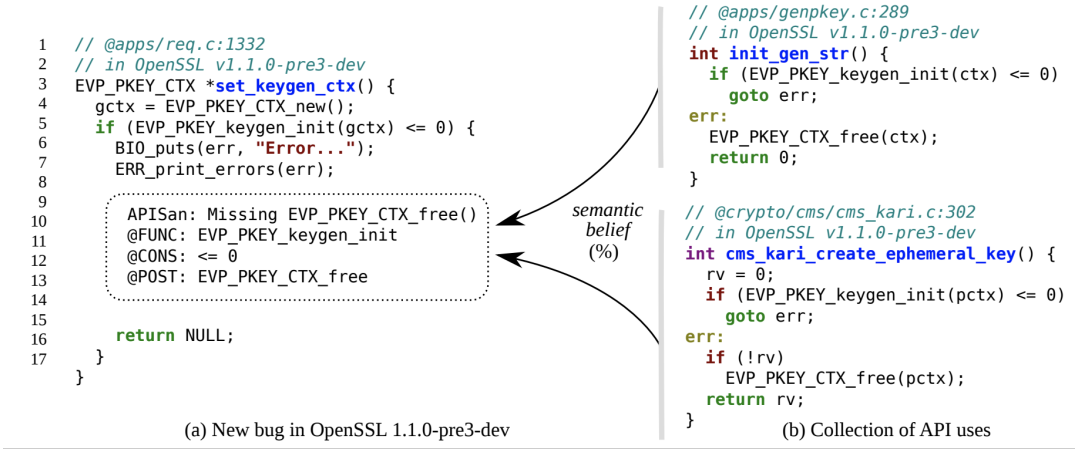
significantly improves over existing approaches, including in synthesizing invented predicates, reducing running times, and in decreasing variances in synthesis performance. In particular, we compared Prosynth to two state-of-the-art approaches described above: ALPS and Difflog. On a suite of 40 synthesis tasks from three different domains—program analysis, knowledge discovery, and relational queries—Prosynth was able to synthesize the desired program in 10 seconds on average per task, taking only under a second each for 28 of them. In contrast, ALPS timed out in one hour on six tasks and took 142 seconds on average for the rest. Likewise, Difflog timed out on three tasks and took 136 seconds on average for the rest. Finally, compared to Difflog, Prosynth exhibited much lower variability in running times across 32 runs on each task; and for all but three tasks, the maximum running time of Prosynth was lower than the median running time of Difflog.

3.6 Semantic Cross-Checking to Find API Misuse Bugs

We also explored combining logical and probabilistic reasoning in the setting of static bug detection where specifications are missing. Specifically, we targeted the problem of finding API misuse bugs in a fully automated manner [8]. We implemented this approach in a system called APISan whose architecture and workflow is depicted in Figure 15.

APISan first builds symbolic contexts from existing programs' source code and creates a database. Then APISan infers correct usages of APIs, so-called semantic beliefs, in four aspects: return values (e.g. return result of an API should be checked for NULL), arguments (e.g. the size argument should be smaller or equal to the size of the buffer argument), causality (e.g. an acquired lock should be released at the end of a critical section), and implied pre/post conditions (e.g. verifying a peer certificate is valid only if the peer certificate exists). The inferred beliefs are used to find and rank potential API misuses to be reported as bugs.

Figure 16 (a) shows a memory leak vulnerability found by APISan in OpenSSL 1.1.0. When a crypto key fails to initialize, the allocated context (i.e., `gctx`) should be freed. Otherwise, a memory leak will occur. APISan first inferred the correct usage of the API from (b) other uses of the API, and extracted a checkable rule, called a semantic belief, under the proper context (e.g., `state: EVP_PKEY_keygen_init() -> rv <= 0 && EVP_PKEY_CTX_free()`). This newly found vulnerability was reported and fixed in the mainstream with the patch we provided. In the above report, `@FUNC` indicates a target API, `@CONS` is a return value



(a) New bug in OpenSSL 1.1.0-pre3-dev (b) Collection of API uses

Figure 16: A new bug discovered using APISan in OpenSSL.

constraint, and `@POST` shows an expected post-action following the API.

3.7 Deep Learning for Program Reasoning

The dramatic benefits of leveraging machine learning for program reasoning in the above work inspired us to explore the ultimate form of continuous optimization: deep learning. While deep learning has shown promise for relatively shallow program reasoning tasks such as code completion, API prediction, and even type checking, an open question is whether it can also overcome long-standing challenges in tasks that require deep semantic reasoning. Our exploration led to the first successful formulations of such tasks using deep learning: Code2Inv [9], a program verifier; Metal [10], a program synthesizer; and Hoppity [11], an end-to-end bug-finding and program repair system.

The key insight is to leverage two kinds of state-of-the-art artificial neural networks:

- A Graph Neural Network (GNN), which embed programs into high-dimensional vectors such that the distance between vectors is correlated with semantic equivalence of the embedded programs. Graph neural networks do not require human-specified features; they directly operate on graph-based program representations to capture rich structural and semantic information in programs necessary for tasks such as verification, synthesis, and repair.

- A Long Short-Term Memory network (LSTM), which takes a series of actions required by the program reasoning task (verification/synthesis/repair) by using an attention mechanism.

We illustrate the use of these neural networks in Hoppity whose architecture is depicted in Figure 17. Hoppity's neural network model is designed in a similar vein as a Neural Turing Machine (NTM) [62]: it consists of an external memory implemented using a GNN for embedding the given program and a central controller implemented using an LSTM that makes a sequence of primitive actions (e.g., predicting type, generating patch, etc.) to perform a fix. The multi-step decision process is implemented by an autoregressive model.

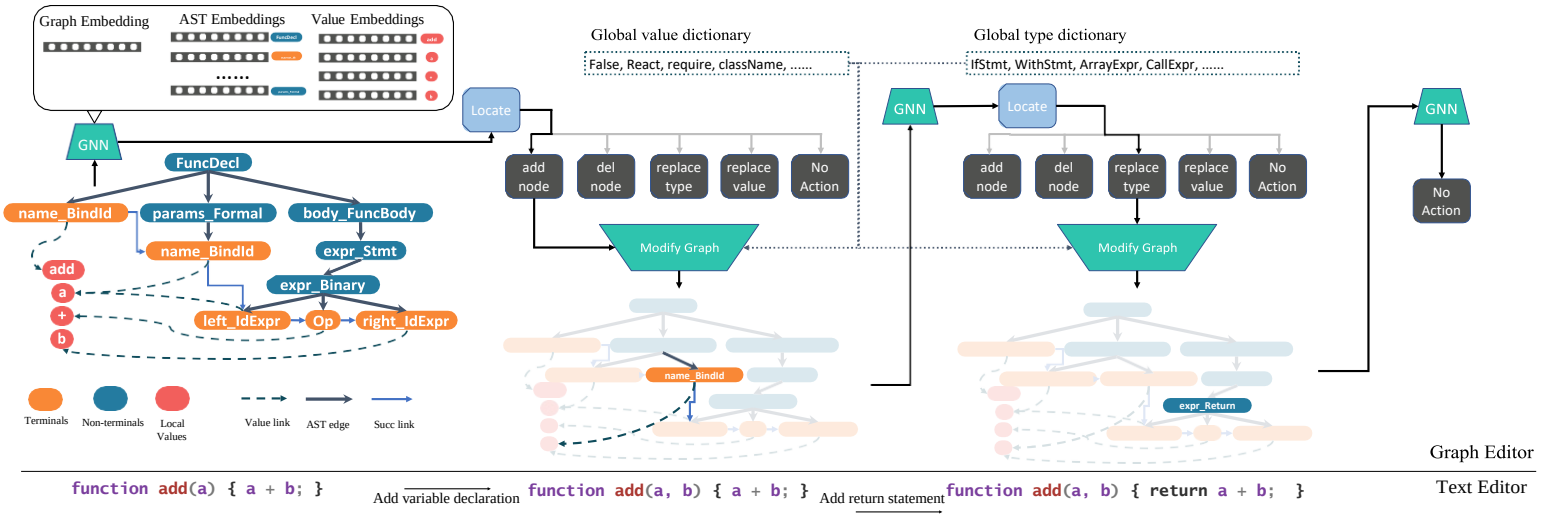


Figure 17: Code repair as graph transformation in the Hoppity system. In each step, the source code graph is edited via one of the operator modules, until STOP is triggered by the controller.

```
function clearEmployeeListOnLinkClick(){
  document.querySelector("a").addEventListener("click",
    function(event){
      document.querySelector("ul").InnerHTML = "";
    }
  );
}
```

(a) `InnerHTML` should have been `innerHTML`.

```
if (matches) {
  return {
    episode: Number(matches.groups.episode),
    hosts: matches.groups.hosts.split(/([,&]+|\sand\s)/).
              map(el => S(el).trim().s)
  };
}
```

(b) Highlighted parentheses should have been removed.

```
module.exports = function (grunt) {
  grunt.initConfig({
    execute: {...}, copy: {...}, checktextdomain: {...}
    wp_readme_to_markdown: {...}, makepot: {...}})
  ...
  grunt.registerTask('default',['wp_readme_to_markdown'
    ,'makepot','execute','checktextdomain'])
};
```

(c) `copy` function should have also been included in the highlighted list.

```
export default {
  computed: {
    level () {
      return dictMap.skillLevel[
        parseInt((this.value === 0 ? 1 : this.value)/20)];
    }
  },...
}
```

(d) `parseInt` should have been removed because `===` implies `this.value` is an integer.

Figure 18: Example programs that illustrate limitations of existing approaches inculding both rule-based static analyzers and neural-based bug predictors.

Crucially, our model differs from the standard NTM in how the memory is manipulated: apart from the common read and write operations, the controller can also expand or shrink the memory when adding or deleting nodes in the original graph.

As proof of concept, Hoppity targeted Javascript, a scripting language designed for web application development, but the main ideas are easily applicable to other languages. Javascript has been the most popular programming language on GitHub since 2014. Repairing Javascript code presents a unique challenge as bugs manifest in diverse forms due to unusual language features and the lack of tooling support. Therefore, the primary goal of our approach was generality since it must be effective against a board spectrum of programming errors, such as using wrong operators or identifiers, accessing undefined properties, mishandling variable scopes, triggering type incompatibilities, among many others.

Another important novel aspect concerns our approach's ability to deal with bugs that are more complex and semantic in nature, namely, bugs that require adding or removing statements from a program, which are not considered by prior works. Finally, compared to automated program repair techniques [63–66] which require knowledge of bug location, Hoppity is an end-to-end approach that including localizing bugs, predicting the types of fixes, and generating patches. By training on 290,715 Javascript code change commits collected from GitHub, Hoppity correctly detects and fixes bugs in 9,490 out of 36,361 programs. Figure 18 shows four different bugs that Hoppity is able to correctly identify.

These preliminary results point to the vast potential of Big Code methods based on deep learning to revolutionize program reasoning and provide significant value to regular programmers and end-users of software.

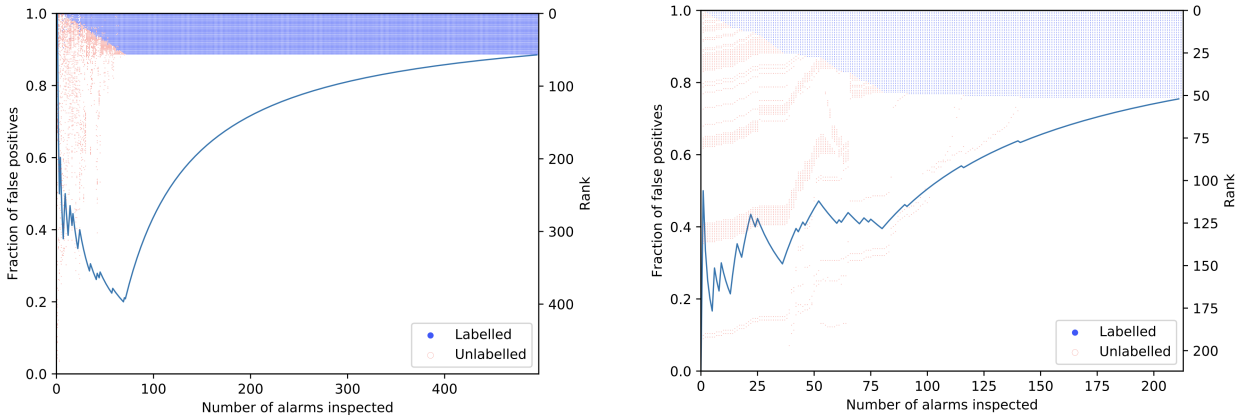


Figure 19: Result of applying Bingo to two different analyses and programs.

4 Results and Discussion

In this section, we elaborate upon the results of applying the techniques and tools developed in Petablox to real-world settings in different programming languages (e.g. Java and C), correctness properties (e.g. memory safety and API misuse), and developer environments (e.g. batch mode vs. CI/CD mode). This section is organized as follows.

Section 4.1 presents the empirical results of our approaches for ranking analysis alarms, which were described in Section 3.3.

Section 4.2 presents the empirical results of our approach for finding API misuse errors without any specifications, which was described in Section 3.6.

Section 4.3 summarizes the findings of a comprehensive bug-finding assessment we conducted in the wild to compare Petablox against state-of-the-art static bug-finding tools on a common codebase.

Section 4.4 describes our experience integrating our system into the CI/CD build environment on the GitHub platform.

4.1 Ranking Analysis Alarms Using Bayesian Inference

We applied our Bayesian inference approach to ranking analysis alarms in batch mode (Bingo [4]) and CI/CD mode (Drake [15]). We next present the results in each mode.

4.1.1 Batch Mode

In batch mode, our approach enabled a reduction of over 50% in alarm inspection effort for a variety of static analysis problems—finding malicious information leaks in Android apps, finding concurrency bugs in large object-oriented programs in Java, and finding memory safety vulnerabilities in low-level systems programs in C.

Figure 19 shows the results of applying Bingo to two different analyses and programs. In both plots, real bugs (red dots) rise in the ranking after the top-ranked alarm is labeled as real or false (blue dots) in each round of interaction, depicting how the approach generalizes user feedback.

Table 1: Statistics of ten benchmark C programs analyzed by Drake in CI/CD mode.

Program	Version		Size (KLOC)		Δ	#Bugs	Bug Type
	Old	New	Old	New	(%)		
shntool	3.0.4	3.0.5	13	13	1	6	Integer overflow
latex2rtf	2.1.0	2.1.1	27	27	3	2	Format string
urjtag	0.7	0.8	45	46	18	6	Format string
optipng	0.5.2	0.5.3	60	61	2	1	Integer overflow
wget	1.11.4	1.12	42	65	47	6	Buffer overrun
readelf	2.23.2	2.24	63	65	6	1	Buffer overrun
grep	2.18	2.19	68	68	7	1	Buffer overrun
sed	4.2.2	4.3	48	83	40	1	Buffer overrun
sort	7.1	7.2	96	98	3	1	Buffer overrun
tar	1.27	1.28	108	112	4	1	Buffer overrun

The plot on the left shows the result of applying our approach to a static datarace checker Chord [28] on Apache FTP Server [29], an open-source multi-threaded Java program comprising 150KB of Java bytecode. On this program, the baseline checker produces 522 alarms, of which only 75 are true dataraces. In the interactive mode, however, the user is able to discover all real dataraces within just 103 rounds of interaction. The false positive rate effectively drops from 86% to just 27%.

The plot on the right shows the result of applying our approach to a static taint checker [67] on a malicious Android app from the Symantec suite. The improvement trend in false positive rate is similar.

4.1.2 CI/CD Mode

The improvement in accuracy is even more dramatic in CI/CD mode compared to batch mode. We applied Drake in the Sparrow static analyzer [18] on bug-introducing code commits to 10 popular GNU utility C programs comprising 13K-112K lines of code each. Sparrow's checkers include an interval analysis for buffer overrun errors and a taint analysis for format-string and integer-overflow errors. Statistics of the benchmark programs and code changes are shown in Table 1. Drake could discover all 26 bugs, including four known CVEs, while requiring the developer to inspect only 30 alarms on average per program, whereas the baseline batch approach required inspecting 560 alarms on average.

Figure 20 shows a plot of the reduction in alarms using Drake over previous approaches, including Bingo (batch mode described above) and Syntactic Masking, the dominant approach in industry of syntactically masking alarms occurring in the older version of the program. The absolute number of alarms in the new version of each benchmark is shown at the top. Consider the case of the grep program. The program comprises 68K lines of code. When grep's program version changed from 2.18 to 2.19, only 7% of the code changed, and a buffer-overrun vulnerability (CVE-2015-1345) was introduced which allowed attackers to execute arbitrary code. Sparrow reports 913 alarms in the new version of grep. It includes an alarm corresponding to this vulnerability but it is buried amidst a large number of false alarms. Syntactic masking reports 22% of these alarms but misses the newly introduced bug

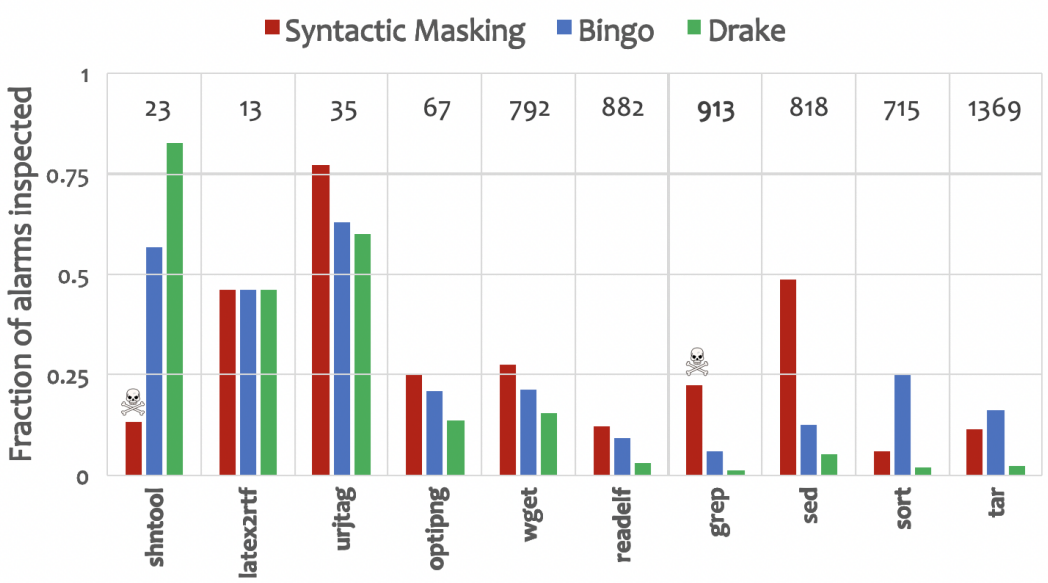


Figure 20: Result of applying Drake to a suite of 10 benchmark C programs.

(such false negatives are indicated by the Jolly Roger icon in Figure 20). Bingo, our batch mode interactive approach, is able to reveal the vulnerability in 53 rounds of interaction—a significant improvement over syntactic masking since the user has to only inspect 6% of all alarms by Sparrow. Lastly, Drake, our CI/CD mode interactive approach, is able to reveal the vulnerability in only 9 rounds of interaction–a significant improvement over Bingo since the user has to only inspect 1% of all alarms!

4.2 Checking API Misuse Errors Without Specifications

In [8], we applied APISan to popular C programs: Linux v4.5-rc4, OpenSSL 1.1.0- pre3-dev, PHP 7.0, Python 3.6, and all 1,204 Debian packages using the OpenSSL library. Collectively, the analyzed programs comprise 92 million lines of code.

APISan generated 40,006 reports in total, and we analyzed the reports according to ranks. As a result, APISan found 76 previously unknown bugs: 64 in Linux, 3 in OpenSSL, 4 in PHP, 1 in Python, and 5 in the Debian packages (see Table 2 for details). We created patches for all the bugs and sent them to the mainline developers of each project. Of these, 69 bugs were confirmed by the developers and patches for most of them were applied to the mainline repositories.

All of the bugs we found had serious security implications: e.g., code execution, system crash, Man-In-The-Middle attack, etc. For a few bugs including integer overflows in Python (CVE-2016-5636) and PHP, we could even successfully exploit them by chaining ROP gadgets. In addition, we found that the vulnerable Python module is in the whitelist of Google App Engine and reported it to Google.

Table 2: New bugs discovered by APISan in popular C programs.

Program	Module	API misuse	Impact	Checker	#bugs	S.
Linux	cifs/cifs_dfs_ref.c	heap overflow	code execution	`args`	1	✓
	xenbus/xenbus_dev_frontend.c	missing integer overflow check	code execution	`intovfl`	1	✓
	ext4/resize.c	incorrect integer overflow check	code execution	`intovfl`	1	✓
	tipc/link.c	missing `tipc_bcast_unlock()`	deadlock	`cpair`	1	✓
	clk/clk.c	missing `clk_prepare_unlock()`	deadlock	`cpair`	1	✓
	hotplug/acpiphp_glue.c	missing `pci_unlock_rescan_remove()`	deadlock	`cpair`	1	✓
	usbvision/usbvision-video.c	missing `mutex_unlock()`	deadlock	`cpair`	1	✓
	drm/drm_dp_mst_topology.c	missing `drm_dp_put_port()`	DoS	`cpair`	1	✓
	affs/file.c	missing `kunmap()`	DoS	`cpair`	1	✓
	acpi/sysfs.c	missing `kobject_create_and_add()` check	system crash	`rvchk`	1	✓
	cx231xx/cx231xx-417.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	qxl/qxl_kms.c	missing `kmalloc()` check	system crash	`rvchk`	1	P
	chips/cfi_cmdset_0001.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	ata/sata_sx4.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	hsi/hsi.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	mwifiex/sdio.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	usbtv/usbtv-video.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	cxgb4/clip_tbl.c	missing `t4_alloc_mem()` check	system crash	`rvchk`	1	✓
	devfreq/devfreq.c	missing `devm_kzalloc()` check	system crash	`rvchk`	2	✓
	i915/intel_dsi_panel_vbt.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	gpio/gpio-mcp23s08.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	drm/drm_crtc.c	missing `drm_property_create_range()` check	system crash	`rvchk`	13	✓
	gma500/framebuffer.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	emu10k1/emu10k1_main.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	m5602/m5602_s5k83a.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	hisax/isdnl2.c	missing `skb_clone()` check	system crash	`rvchk`	1	✓
	qlcnic/qlcnic_ctx.c	missing `qlcnic_alloc_mbx_args()` check	system crash	`rvchk`	1	✓
	xen-netback/xenbus.c	missing `vzalloc()` check	system crash	`rvchk`	1	✓
	i2c/ch7006_drv.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	fmc/fmc-fakedev.c	missing `kmemdup()` check	system crash	`rvchk`	1	P
	rc/igorplugusb.c	missing `rc_allocate_device()` check	system crash	`rvchk`	1	✓
	s5p-mfc/s5p_mfc.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	fusion/mptbase.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	nes/nes_cm.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	✓
	dvb-usb-v2/mxl111sf.c	missing `mxl111sf_enable_usb_output()` check	malfunction	`rvchk`	2	✓
	misc/xen-kbdfront.c	missing `xenbus_printf()` check	malfunction	`rvchk`	1	✓
	pvrusb2/pvrusb2-context.c	incorrect `kthread_run()` check	malfunction	`rvchk`	1	P
	agere/et131x.c	incorrect `drm_alloc_coherent()` check	malfunction	`rvchk`	1	✓
	drbd/drbd_receiver.c	incorrect `crypto_alloc_hash()` check	malfunction	`rvchk`	1	✓
	mlx4/mr.c	incorrect `mlx4_alloc_cmd_mailbox()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `kzalloc()` check	maintanence	`rvchk`	2	✓
	aoe/aoecmd.c	incorrect `kthread_run()` check	maintanence	`rvchk`	1	✓
	ipv4/tcp.c	incorrect `crypto_alloc_hash()` check	maintanence	`rvchk`	1	✓
	mfd/bcm590xx.c	incorrect `i2c_new_dummy()` check	maintanence	`rvchk`	1	P
	usnic/usnic_ib_main.c	incorrect `ib_alloc_device()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `usnic_fwd_dev_alloc()` check	maintanence	`rvchk`	1	✓
OpenSSL	dsa/dsa_gen.c	missing `BN_CTX_end()`	DoS	`cpair`	1	✓
	apps/req.c	missing `EVP_PKEY_CTX_free()`	DoS	`cpair`	1	✓
	dh/dh_pmeth.c	missing `OPENSSL_memdup()` check	system crash	`rvchk`	1	✓
PHP	standard/string.c	missing integer overflow check	code execution	`intovfl`	3	✓
	phpdbg/phpdbg_prompt.c	format string bug	code execution	`args`	1	✓
Python	Modules/zipimport.c	missing integer overflow check	code execution	`intovfl`	1	✓
rabbitmq	librabbitmq/amqp_openssl.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
hexchat	common/server.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
lprng	auth/ssl_auth.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	P
afflib	lib/aftest.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓
	tools/aff_bom.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓

Table 3: Four representative bugs found in the MIT Drake system.

Bug Kind	Module	Link	Tool	High-Level Description of Bug
Null Dereference	externals/ipopt	[68]	FB Infer	A pointer is allocated memory depending on a condition. But the condition may not hold when the pointer is accessed.
Null Dereference	drake/multibody	[69]	APISan	A xml attribute value is retrieved and used. If the attribute does not exist, null dereference can occur. By testing, we confirmed it causes a crash.
Buffer Underrun	externals/ipopt	[70]	APISan	An integer element is retrieved from a priority queue and used as index of a buffer. If the queue is empty, the index can be -1.
Performance Bug	drake/multibody	[71]	APISan	The C++ idiom "move" to avoid unnecessary copies is not used at a point whereas it is almost always used throughout the code.

4.3 MIT Drake Study

In June 2017, we conducted a comprehensive bug-finding assessment in the wild, comparing Petablox against state-of-the-art static bug-finding tools on a common codebase. The codebase we chose was MIT Drake [16], a large and complex system for robotic application, which comprises 2.1M lines of C/C++ source code (1.97 million lines of Drake externals plus 138 thousand lines of Drake core). The goal was to run all tools, manually triage bugs found, and evaluate both quantitative metrics such as false positive rates and qualitative aspects such as the kinds and severity of bugs found.

We ran various checkers in the following four tools on the Drake codebase:

- APISan [8]: API misuse checkers in Petablox
- Facebook Infer [72]: Memory safety checkers
- MIT Kint [73]: Integer overflow checker
- Coverity [24]: Various bug pattern checkers

We were able to obtain results for only Petablox and FB Infer. Kint reported problems compiling Drake that are easy but tedious to fix. Coverity is not free to download but it is available as a free service for projects hosted on the GitHub platform (such as Drake); however, we were unable to obtain the results from the service despite multiple attempts.

Our main findings were as follows:

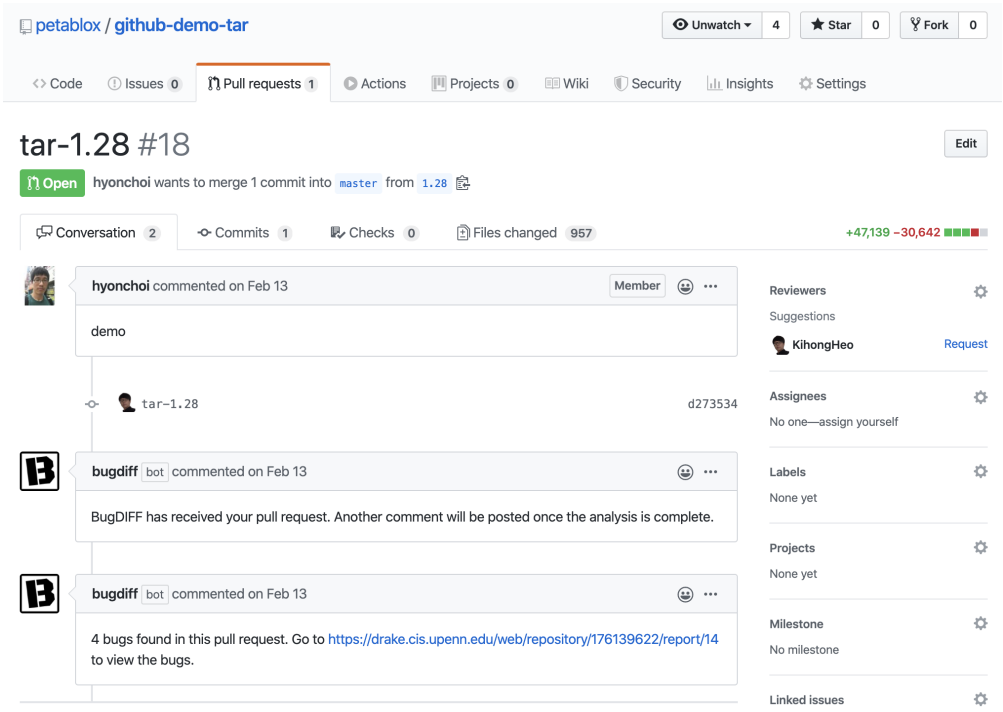


Figure 21: Screenshot of notifications on the analyzed project's GitHub webpage.

Alarms Introduced in Pull Request

tar-1.28

Head Branch: 1.28

Base Branch: master

Alarms are ordered by how dissimilar they are to alarms in the base version. Alarms at the top are most likely to have been introduced by the code changes.

Alarm ID	Alarm Type	File	Function	Line	Bug Prob*	Veracity	
1	Dereference of null pointer	src/checkpoint.c	getarg	182	0.531441	Unknow	View Alarm
2	Dereference of undefined pointer value	src/exclist.c	hg_initfn	287	0.9	True / False / ✓ Unknown	View Alarm
3	Dereference of null pointer	src/buffer.c	_write_volume_label	1655	0.81	Unknow	View Alarm
4	Dereference of null pointer	src/buffer.c	gnu_add_multi_volume_header	1732	0.185302	Unknow	View Alarm

See All Reports for this Repository

* Bug Prob is a *beta* feature measuring how likely is is that this alarm is a true bug on a [0,1] scale

Figure 22: Screenshot of reports produced upon completion of Petablox's analysis.

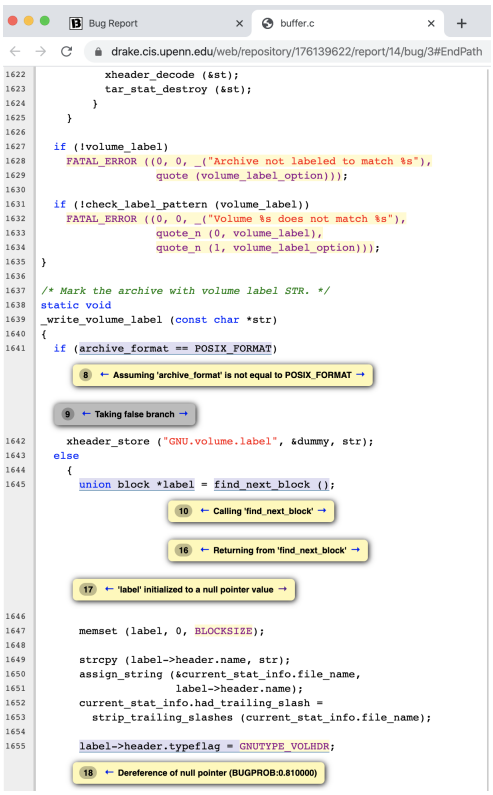


Figure 23: Screenshot of a sample Clang Static Analyzer report.

1. APISan and FB Infer were able to run successfully on the multi-million line codebase of Drake. This attests to the suitability of these tools for analyzing large and complex autonomous software systems.

2. Petablox found 1 performance bug, 2 buffer underrun bugs and 9 null dereference bugs, while FB Infer found 138 potential null dereference bugs. Four representative bugs are shown in Table 3.

Comprehensive details of our study and raw logs of APISan and FB Infer are available at `https://github.com/petablox/Drake-Analysis-June2017`.

4.4 Technology Transition to GitHub

The Petablox project successfully demonstrated the ability to statically discover software defects in large Java and C/C++ programs with high accuracy (i.e. low false-positive and false-negative rates). In the final year of the project, we integrated the Petablox system into a production platform for developers. As proof of concept, we targeted the GitHub platform, a mature and widely-used platform that caters to an extensive body of open-source projects and developers.

Specifically, we developed an app on GitHub's marketplace that allows developers of projects hosted on the GitHub platform to give permission to Petablox's code analyzers to 1) access their source code, and 2) listen to code commit events. We also integrated the

analyzers into GitHub's build system called CI (Continuous Integration). The resulting system triggers a run of the analyzers every time a developer commits code to the project (called a "pull request" in Git parlance).

We demonstrate the working of this system by simulating the introduction of an actual null dereference error that was introduced in the popular UNIX program `tar` when its version changed from 1.27 to 1.28. Figure 21 shows a screenshot of notifications by the Petablox app on the analyzed project's webpage on GitHub. The notifications are triggered upon the pull request committing the code change in `tar`.

The first message the developer receives is:

> "BugDIFF has received your pull request. Another comment will be posted once the analysis is complete".

In the interim, Petablox runs Drake on the two program versions (`tar` 1.27 and `tar` 1.28) on a remote server. Upon completion, the results of the analysis are posted via a second message to the developer:

> "4 bugs found in this pull request. Go to <url> to view the bugs".

Figure 22 shows a screenshot of the analysis results when the developer follows the link specified in <url> in the second message above. In this case, the underlying static analyzer is Clang Static Analyzer [17]. Note that the null dereference error is among the four alarms reported by Drake (it is the one with Bug Probability value of 0.81). Clicking on the "View Alarm" link beside the alarm takes the developer to Clang Static Analyzer's details of the alarm, depicted in Figure 23. Upon inspecting the details of an alarm, the developer can label the alarm as True or False (the default label is Unknown), which allows Petablox to improve the ranking in future runs.

5 Conclusions

The Petablox project set out to address long-standing challenges of accuracy, scalability, and usability in program analysis by leveraging Big Code. The project achieved this goal by fundamentally extending the prevalent logical reasoning approach to program analysis with statistical and data-driven modes of reasoning. It adopted and advanced a broad range of ML and AI techniques spanning logical rule learning, probabilistic graphical models, and modern deep learning. The approach was successfully demonstrated in a variety of real-world settings—different programming languages, correctness properties, and developer environments. The ideas conceived in this project have brought the vision of intelligent programming systems for ensuring software quality a step closer to realization.

6 References

[1] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2014.

[2] Ravi Mangal, Xin Zhang, Mayur Naik, and Aditya V. Nori. A user-guided approach to program analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2015.

[3] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective interactive resolution of static analysis alarms. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, October 2017.

[4] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using bayesian inference. In *International Conference on Programming Language Design and Implementation (PLDI'18)*, June 2018.

[5] Xujie Si, Xin Zhang, Vasco Manquinho, Mikolas Janota, Alexey Ignatiev, and Mayur Naik. On incremental core-guided MaxSAT solving. In *International Conference on Principles and Practice of Constraint Programming (CP)*, September 2016.

[6] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of Datalog programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2018.

[7] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing Datalog programs using numerical relaxation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[8] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic cross-checking. In *USENIX Security Symposium*, August 2016.

[9] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *International Conference on Neural Information Processing Systems (NeurIPS)*, December 2018.

[10] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations (ICLR)*, May 2019.

[11] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, April 2020.

[12] Ravi Mangal, Xin Zhang, Mayur Naik, and Aditya V. Nori. Volt: A lazy grounding framework for solving very large MaxSAT instances. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, September 2015.

[13] Ravi Mangal, Xin Zhang, Aditya Kamath, Aditya V. Nori, and Mayur Naik. Scaling relational inference using proofs and refutations. In *Conference on Artificial Intelligence (AAAI)*, February 2016.

[14] Xin Zhang, Ravi Mangal, Mayur Naik, and Aditya V. Nori. Query-guided maximum satisfiability. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2016.

[15] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuously reasoning about programs using differential Bayesian inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2019.

[16] The MIT Drake system for robotic applications. `https://drake.mit.edu/`.

[17] Clang static analyzer. `https://clang-analyzer.llvm.org/`.

[18] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. The Sparrow static analyzer. `https://github.com/ropas/sparrow`, 2012.

[19] Petablox source code repositories. `https://github.com/petablox`.

[20] Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in software analysis: Applications and techniques. In *International Conference on Computer-Aided Verification (CAV)*, July 2017.

[21] Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in program analysis: Applications and techniques. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2018.

[22] Xin Zhang, Xujie Si, and Mayur Naik. Combining the logical and the probabilistic in program analysis. In *First ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL)*, June 2017.

[23] Mukund Raghothaman, Sulekha Kulkarni, Richard Zhang, Xujie Si, Kihong Heo, Woosuk Lee, and Mayur Naik. Difflog: Beyond deductive methods in program analysis. In *First Workshop on Machine Learning for Programming (ML4P)*, July 2018.

[24] 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.

[25] Andy Chou. On detecting Heartbleed with static analysis. `https://www.synopsys.com/blogs/software-security/detecting-heartbleed-with-static-analysis/`.

[26] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[27] MaxSAT evaluations. http://www.maxsat.udl.cat/.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[29] Apache FTP Server. `http://mina.apache.org/ftpserver-project/`.

[30] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[31] Arun Chaganty, Akash Lal, Aditya Nori, and Sriram Rajamani. Combining relational learning with SMT solvers using CEGAR. In *International Conference on Computer-Aided Verification (CAV)*, July 2013.

[32] Sebastian Riedel. Improving the accuracy and efficiency of MAP inference for Markov Logic. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2008.

[33] Jan Noessner, Mathias Niepert, and Heiner Stuckenschmidt. RockIt: Exploiting parallelism and symmetry for MAP inference in statistical relational models. In *Conference on Artificial Intelligence (AAAI)*, February 2013.

[34] S Kok, M Sumner, M Richardson, P Singla, H Poon, D Lowd, and P Domingos. The Alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2007.

[35] Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. Tuffy: Scaling up statistical inference in Markov Logic Networks using an RDBMS. In *International Conference on Very Large Data Bases (VLDB)*, 2011.

[36] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. CCLS: an efficient local search algorithm for weighted maximum satisfiability. *IEEE Trans. Computers*, 64(7):1830–1843, 2015.

[37] Adrian Kügel. Improved exact solver for the weighted MAX-SAT problem. In *POS-10. Pragmatics of SAT*, 2010.

[38] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Conference on Artificial Intelligence (AAAI)*, July 2014.

[39] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MaxSAT solving. In *International Conference on Principles and Practice of Constraint Programming (CP)*, September 2013.

[40] Mikoláš Janota. MiFuMax — a literate MaxSAT solver, 2013.

[41] Joao Marques-Silva and Alexey Ignatiev and António Morgado. MSCG - Maximum Satisfiability: a Core-Guided approach, 2014.

[42] António Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided maxsat with soft cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming (CP)*, September 2014.

[43] WPM MaxSAT solver. `http://web.udl.es/usuaris/q4374304/`.

[44] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, July 2013.

[45] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2), 2006.

[46] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, August 2006.

[47] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, July 2014.

[48] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of Datalog programs. In *International Conference on Principles and Practice of Constraint Programming (CP)*, August 2017.

[49] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of Datalog programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2020.

[50] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, October 2013.

[51] Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1), 2015.

[52] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *5th Annual Workshop on Computational Learning Theory (COLT)*, 1992.

[53] Yoav Freund, H. Sebastian Seung, Eli Shamir, and Naftali Tishby. Selective sampling using the query by committee algorithm. *Machine Learning*, 28(2-3), 1997.

[54] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[55] Andrew Cropper and Stephen Muggleton. Logical minimisation of meta-rules within meta-interpretive learning. In *Inductive Logic Programming*, pages 62–75. Springer, 2015.

[56] Todd Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *ACM Symposium on Principles of Database Systems (PODS)*, 2007.

[57] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[58] Grigoris Karvounarakis, Zachary Ives, and Val Tannen. Querying data provenance. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2010.

[59] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *IEEE International Conference on Data Engineering (ICDE)*, April 2008.

[60] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[61] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 36–50. Springer, 2005.

[62] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[63] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, November 2019.

[64] Andrew Scott, Johannes Bader, and Satish Chandra. Getafix: Learning to fix bugs automatically. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, October 2019.

[65] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, November 2018.

[66] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808*, 2018.

[67] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, November 2014.

[68] Bug 1 found by APISan in the MIT Drake system. `https://github.com/petablox/Drake-Analysis-June2017/blob/master/analysis_results/FB_Infer.md#alarm-1-drakeexternalsipoptipoptsrclinalgipexpansionmatrixcpp371`.

[69] Bug 2 found by APISan in the MIT Drake system. `https://github.com/petablox/Drake-Analysis-June2017/blob/master/analysis_results/Petablox.md#alarm-1-missing-non-null-check-drakemultibodyparsersurdf_parsercc1320`.

[70] Bug 3 found by APISan in the MIT Drake system. https://github.com/petablox/Drake-Analysis-June2017/blob/master/analysis_results/Petablox.md#alarm-2-missing--1-check-externalsipoptthirdpartymetismetis-40libsfmc352.

[71] Bug 4 found by APISan in the MIT Drake system. https://github.com/petablox/Drake-Analysis-June2017/blob/master/analysis_results/Petablox.md#alarm-3-causality-between-stdmove-and-set_parent-drakemultibodyparserssdf_parsercc657.

[72] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Method Symposium*, pages 3–11. Springer, 2015.

[73] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with KINT. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

7 List of Symbols, Abbreviations, and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CEGAR	Counter-Example Guided Abstraction Refinement
CEGIS	Counter-Example Guided Inductive Synthesis
CI/CD	Continuous Integration/Continuous Delivery
CPD	Conditional Probability Distribution
DPLL	Davis-Putnam-Logemann–Loveland
GNN	Graph Neural Network
ILP	Inductive Logic Programming
LSTM	Long Short-Term Memory
MaxSAT	Maximum Satisfiability
ML	Machine Learning
NTM	Neural Turing Machine
QBC	Query By Committee
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories
SyGuS	Syntax-Guided Synthesis