AFRL-RI-RS-TR-2017-032

STATIC ANALYSIS OF MOBILE PROGRAMS

STANFORD UNIVERSITY

FEBRUARY 2017

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 nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-032 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation and Operations 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)*
FEBRUARY 2017	FINAL TECHNICAL REPORT	FEB 2012 – JUN 2016

4. TITLE AND SUBTITLE

STATIC ANALYSIS OF MOBILE PROGRAMS

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-12-2-0020

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Alex Aiken

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
97

5f. WORK UNIT NUMBER
71

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

Stanford University
353 Serra Mall, Gates 411
Stanford, CA 94305-9045

8. PERFORMING ORGANIZATION REPORT NUMBER

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

Air Force Research Laboratory/RIGB
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-2017-032

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of the STAMP (STatic Analysis of Mobile Programs) project has been to build tools for proving the absence of malware in Android applications, also known as apps. The performers focus is on guarantees, their techniques have a large static component, as static proofs are the only known method of reasoning about all possible program executions. Like most systems written in modern languages, Android applications heavily use pointer data structures, complex path conditions, and multiple layers of object-oriented abstractions. Stanford's hypothesis has been that static analysis techniques have reached the point that sound, precise and scalable static analysis for interesting security properties is entirely feasible.

15. SUBJECT TERMS
Static Analysis, Mobile Programs, Precise and Scalable Static Analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK WILLIAMS**
U	U	U	UU	14	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

1 OVERVIEW

The goal of the STAMP (STatic Analysis of Mobile Programs) project has been to build tools for proving the absence of malware in Android applications, also known as *apps*. Because our focus is on guarantees, our techniques have a large static component, as static proofs are the only known method of reasoning about all possible program executions. Like most systems written in modern languages, Android applications heavily use pointer data structures, complex path conditions, and multiple layers of object-oriented abstractions. Our hypothesis has been that static analysis techniques have reached the point that sound, precise and scalable static analysis for interesting security properties is entirely feasible.

There were two problems with this plan, both of which were highlighted in the original proposal and recognized as issues that would need to be addressed in the research. The first is that while static analysis has advanced tremendously in its ability to extract all information available from the program text, there is still the problem of the information that is simply not in the program. In particular, whether or not a particular program behavior constitutes malware often depends on context that is unavailable to the static analysis tool—sending personal information from an application to a website is legitimate if it is part of the advertised functionality (e.g., a service that backs-up a phone's configuration information) but is likely malware if it is done without the user's knowledge and at least implicit consent. The vast majority of research in static analysis is on methods for checking well-known, and fixed, specifications such as memory safety. Thus, a challenging aspect of the problem of detecting malware in Android applications is the approach used to specify what a particular app is allowed, and not allowed, to do.

The second issue was that a fully static analysis was never a realistic possibility, because Java, the programming langauge for Android apps, has several commonly used dynamic features that are simply not analyzable except at runtime: code reflection, dynamic loading and native methods are three good examples. Obtaining accurate information about software that contains any of these features requires at least some analysis at runtime.

As the project evolved, we converged on a standard approach of adding specifications at interfaces that were too difficult to analyze statically. These specifications took the form of aliasing declarations, saying that two program names may refer to the same memory location, and information flow specifications, saying in a polymorphic way which results of a method may be tainted if some input is tainted. However, writing these specifications required human effort, and more importantly missing a specification resulted in analysis results that were incomplete, usually incorrect, and difficult to understand and debug.

Eventually, we realized that it should be possible to infer the specifications automatically using dynamic analysis. We would run the program multiple times, gather all of the program state at the point where we needed a specification of behavior, and then simply use the test data to find the most likely specification (i.e., a specification consistent with all the test data, but with no extra features not justified by the test data). This idea came to define the project: use dynamic analyiss to guess the correct properties a program points of interest, and then use static analysis to verify them. In the end we pursued this idea successfully for several differnet properties and developed novel approaches for mining the intial guesses for specifications from traces. In every place where we tried the idea, the results have been good and much better than alternatives that we tested.

2 STAMP

The main software product of the project is STAMP, a system for analysing Android applications. While the research purpose of STAMP was to analyze programs for malware, because a goal of

STAMP was to scale to large programs it had to handle essentially all of the features of Java and could also be used as a general-purpose analysis engine. The important features of STAMP are:

- An interface for reading and interpreting DEX bytecode, allowing analysis of libraries in compiled form, both statically and dynamically. The reason analysis of bytecode is necessary is that many programs, or portions of programs, were only available to us in bytecode form (e.g., apps downloaded from app stores).
- A scalable pointer analysis. Some form of alias analysis, such as pointer analysis, is required for any static analysis of imperative languages.
- A framework for adding specifications about the behavior of methods, including methods that were unavailable as well as methods that were too difficult to analyze accurately with a pure static analysis. The specification language is fairly general, encompassing both information flow and aliasing properties. We wrote about a thousand specifications by hand during the project, and eventually automatically inferred many more.
- A static information-flow analysis framework for constructing global information flows. There are actually several different information flow analyses that we experimented with.
- An interactive user interface that overlays analysis results on the source program, enabling users to see the analysis information in context.

In addition to being used for research, the STAMP technology was licensed to a software security firm.

The specific results from the project are discussed in the following sections, with each section covering one logically related group of work.

3 STATIC ANALYSIS

Much of the work in the STAMP project involved static analysis, often of very large programs. As such we ended up exploring a number of questions relevant to the structure and expressiveness of different approaches, in addition to specific static analyses themselves.

3.1 Hybrid Static Analysis

Interprocedural static analyses are broadly classified into top-down and bottom-up, depending upon how they compute, instantiate, and reuse procedure summaries. Both kinds of analyses are challenging to scale: top-down analyses are hindered by ineffective reuse of summaries whereas bottom-up analyses are hindered by inefficient computation and instantiation of summaries. In [1],we present a hybrid approach that combines top-down and bottom-up analyses in a manner that gains their benefits without suffering their drawbacks. The approach is general in that it is parametrized by the top-down and bottom-up analyses it combines. We show an instantiation on a type-state analysis and evaluate it on a suite of 12 Java programs of size 60-250 KLOC each. The hybrid approach outperforms both conventional approaches, finishing on all the programs while both of the other approaches fail on the larger programs.

3.2 Detecting Performance Bugs

In the course of this project we read the code of hundreds of object-oriented applications and noticed that there were some repeated poor coding practices for performance that are also potential security vulnerabilities. In [2], we identify and formalizes a prevalent class of asymptotic performance bugs called redundant traversal bugs and present a novel static analysis for automatically

detecting them. We evaluate our technique by implementing it in a tool called CLARITY and applying it to widely-used software packages such as the Google Core Collections Library, the Apache Common Collections, and the Apache Ant build tool. Across 1.6M lines of Java code, CLARITY finds 92 instances of redundant traversal bugs, including 72 that have never been previously reported, with just 5 false positives. To evaluate the performance impact of these bugs, we manually repair these programs and find that for an input size of large inputs, all repaired programs are at least 2.45X faster than their original code.

3.3 Understanding Bottom-Up Analysis

Interprocedural analyses are compositional when they compute overapproximations of procedures in a bottom-up fashion. These analyses are usually more scalable than top-down analyses (but see the discussion of hybrid analyses above), which compute a different procedure summary for every calling context. However, compositional analyses are difficult to develop with enough precision. In [3], we establish a connection between compositional analyses and modular lattices, which require certain associativity between the lattice join and meet operations, and use it to develop a compositional version of the connection analysis by Ghiya and Hendren. Our version is slightly more conservative than the original top-down analysis in order to meet our modularity requirement. When applied to real world Java programs our analysis scaled much better than the original top-down version: The top-down analysis times out in the largest two of five programs, while ours incurred only 2-5% of precision loss in the remaining programs.

3.4 Bottom-up Context-Sensitive Pointer Analysis

In [4], we describe a new bottom-up, subset-based, and context-sensitive pointer analysis for Java. The main novelty of our technique is the constraint-based handling of virtual method calls and instantiation of method summaries. Since our approach generates polymorphic method summaries, it can be context-sensitive without reanalyzing the same method multiple times. We have implemented this algorithm, and compared it with k-CFA and k-obj algorithms on Java applications from the DaCapo and Ashes benchmarks. Our results show that the new algorithm achieves better or comparable precision to k-CFA and k-obj analyses at only a fraction of the cost.

3.5 Exploring Interprocedureal Control-Flow Properties

In [5], we describe a general framework–and its implementation in a tool called EXPLORER–for statically answering a class of interprocedural control flow queries about Java programs. EXPLORER allows users to formulate queries about feasible callstack configurations using regular expressions, and it employs a precise, demand-driven algorithm for answering such queries. Specifically, EXPLORER constructs an automaton A that is iteratively refined until either the language accepted by A is empty (meaning that the query has been refuted) or until no further refinement is possible based on a precise, context-sensitive abstraction of the program. We evaluate EXPLORER by applying it to three different program analysis tasks, namely, (1) analysis of the observer design pattern in Java, (2) identification of a class of performance bugs, and (3) analysis of inter-component communication in Android applications. Our evaluation shows that EXPLORER is both efficient and precise.

3.6 Understanding Relationships Among Static Analyses

Many interprocedural static analyses perform a lossy join for reasons of termination or efficiency. In [6],we study the relationship between two predominant approaches to interprocedural analysis, the summary-based (or functional) approach and the call-strings (or k-CFA) approach, in the pres-

ence of a lossy join. Despite the use of radically different ways to distinguish procedure contexts by these two approaches, we prove that post-processing their results using a form of garbage collection renders them equivalent. Our result extends the classic result by Sharir and Pnueli that showed the equivalence between these two approaches in the setting of distributive analysis, wherein the join is lossless.

We also empirically compare these two approaches by applying them to a pointer analysis that performs a lossy join. Our experiments on ten Java programs of size 400K-900K bytecodes show that the summary-based approach outperforms an optimized implementation of the k-CFA approach: the k-CFA implementation doe not scale beyond k=2, while the summary-based approach proves up to 46% more pointer analysis client queries than 2-CFA. The summary-based approach thus enables, via our equivalence result, to measure the precision of k-CFA with unbounded k, for the class of interprocedural analyses that perform a lossy join.

4 DYNAMIC ANALYSIS

As noted above, exclusively static analyses could not deal with the most dynamic features of the Android platform. Thus we also invested some research effort in dynamic analyses and in producing the tests needed to drive those dynamic analyses.

4.1 Automatic Concolic Testing of Apps

In [7], we present an algorithm and a system for generating input events to exercise smartphone apps. Our approach is based on concolic testing and generates sequences of events automatically and systematically. It alleviates the path explosion problem by checking a condition on program executions that identifies subsumption between different event sequences. We also describe our implementation of the approach for Android and the results of an evaluation that demonstrates its effectiveness on five Android apps.

4.2 Test Generation for Android

In [8], we present a system Dynodroid for generating relevant inputs to unmodified Android apps. Dynodroid views an app as an event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once and for all, Dynodroid monitors the reaction of an app upon each event in a lightweight manner, using it to guide the generation of the next event to the app. Dynodroid also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs. We evaluated Dynodroid on 50 open-source Android apps, and compared it with two prevalent approaches: users manually exercising apps, and Monkey, a popular fuzzing tool. Dynodroid, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app's Java source code on average. Monkey took 20X more events on average than Dynodroid. Dynodroid also found 9 bugs in 7 of the 50 apps, and 6 bugs in 5 of the top 1,000 free apps on Google Play.

5 COMBINED STATIC AND DYNAMIC ANALYSIS

As our ultimate goal has been to produce sound static guarantees, in the end it has been necessary for us to combine the static and dynamic components in a way that enables us to make some factual claim about the program's security properties. The most robust approach, and the one we have pursued in the most depth, has been to use the results of the dynamic analysis as a hypothesis that the static analysis then attempts to verify.

5.1 Using Data to Drive Equivalence Checking

In [9], we present a data driven algorithm for equivalence checking of two loops. The algorithm infers simulation relations using data from test runs. Once a candidate simulation relation has been obtained, off-the-shelf SMT solvers are used to check whether the simulation relation actually holds. The algorithm is sound: insufficient data will cause the proof to fail. We demonstrate a prototype implementation of our algorithm, which is the first sound equivalence checker for loops written in low-level langauges.

5.2 Interactive Verification of Information Flow

App stores are increasingly the preferred mechanism for distributing software, including mobile apps (Google Play), desktop apps (Mac App Store and Ubuntu Software Center), computer games (the Steam Store), and browser extensions (Chrome Web Store). The centralized nature of these stores has important implications for security. While app stores have unprecedented ability to audit apps, users now trust hosted apps, making them more vulnerable to malware that evades detection and finds its way onto the app store. Sound static explicit information flow analysis has the potential to significantly aid human auditors, but it is handicapped by high false positive rates. Instead, auditors currently rely on a combination of dynamic analysis (which is unsound) and lightweight static analysis (which cannot identify information flows) to help detect malicious behaviors.

In [10] and [11], we propose a process for producing apps certified to be free of malicious explicit information flows. In practice, imprecision in the reachability analysis is a major source of false positive information flows that are difficult to understand and discharge. In our approach, the developer provides tests that specify what code is reachable, allowing the static analysis to restrict its search to tested code. The app hosted on the store is instrumented to enforce the provided specification (i.e., executing untested code terminates the app). We use abductive inference to minimize the necessary instrumentation, and then interact with the developer to ensure that the instrumentation only cuts unreachable code. We demonstrate the effectiveness of our approach in verifying a corpus of 77 Android appsour interactive verification process successfully discharges 11 out of the 12 false positives.

6 SPECIFICATION AND INVARIANT INFERENCE

Another significant use of dynamic and static analysis has been to infer specifications and other invariants that are then added as assumptions to the program (i.e., not verified). For truly difficult to analyze or missing code, we cannot do better. This approach has the advantage of first being very likely to use the correct specification (using our techniques) and second making explicit what assumptions the static infrormation flow analysis is making in reaching its conclusions.

6.1 Inferring Loop Invariants

In [12],we describe a "Guess-and-Check" algorithm for computing algebraic equation invariants of the form $\wedge_i f_i(x_1, \ldots, x_n) = 0$, where each f_i is a polynomial over the variables $x_1, \ldots, x_n$ of the program. The guess phase is data driven and derives a candidate invariant from data generated from concrete executions of the program. This candidate invariant is subsequently validated in a check phase by an off-the-shelf SMT solver. Iterating between the two phases leads to a sound algorithm. Moreover, we are able to prove a bound on the number of decision procedure queries which Guess-and-Check requires to obtain a sound invariant. We show how Guess-and-Check can be extended to generate arbitrary boolean combinations of linear equalities as invariants, which

enables us to generate expressive invariants to be consumed by tools that cannot handle non-linear arithmetic. We have evaluated our technique on a number of benchmark programs from recent papers on invariant generation. Our results show we are able to efficiently compute algebraic invariants in all cases, with only a few tests.

6.2 Inferring Library Specifications

In [13], we consider the fact that many safety properties in program analysis, such as many memory safety and information flow problems, can be formulated as source-sink problems. While there are many existing techniques for checking source-sink properties, the soundness of these techniques relies on all relevant source code being available for analysis. As noted above, many programs make use of libraries whose source code is either not available or not amenable to precise static analysis. We address this limitation of source-sink verifiers through a technique for inferring exactly those library specifications that are needed for verifying the client program. We have applied the proposed technique for tracking explicit information flow in Android applications, and we have shown that our method effectively identifies the needed specifications of the Android SDK.

6.3 Inferring Information Flow Specifications via CFL Reachability

In [14], we present a framework for computing context-free language reachability properties when parts of the program are missing. Our framework infers candidate specifications for missing program pieces that are needed for verifying a property of interest, and presents these specifications to a human auditor for validation. We have implemented this framework for a taint analysis of Android apps that relies on specifications for Android library methods. In an extensive experimental study on 179 apps, our tool performs verification with only a small number of queries to a human auditor.

6.4 Inferring Library Information Flow Specifications Using Dynamic Analysis

In [15], we present a technique to mine explicit information flow specifications from concrete executions. These specifications can be consumed by a static taint analysis, enabling static analysis to work even when method definitions are missing or portions of the program are too difficult to analyze statically (e.g., due to dynamic features such as reflection). We present an implementation of our technique for the Android platform. When compared to a set of manually written specifications for 309 methods across 51 classes, our technique is able to recover 96.36% of these manual specifications and produces many more correct annotations that our manual models missed. We incorporate the generated specifications into STAMP, and show that they enable STAMP to find additional true flows. Although our implementation is Android-specific, our approach is applicable to other application frameworks.

7 OTHER

A few things we worked on in the STAMP project do not fit neatly into any of the previous categories.

7.1 Semantics-Based Malware Detection

In [16], we present Apposcopy, a new semantics-based approach for identifying a prevalent class of Android malware that steals private user information. Apposcopy incorporates (i) a highlevel language for specifying signatures that describe semantic characteristics of malware families and

(ii) a static analysis for deciding if a given application matches a malware signature. The signature matching algorithm of Apposcopy uses a combination of static taint analysis and a new form of program representation called Inter-Component Call Graph to efficiently detect Android applications that have certain control- and data-flow properties. We have evaluated Apposcopy on a corpus of real-world Android applications and show that it can effectively and reliably pinpoint malicious applications that belong to certain malware families. The Apposcopy paper has had a significant impact; at present it is the single most cited paper from the STAMP project.

7.2 Bias-Variance Trade-Offs in Program Analysis

In [17], we observe that it is often the case that increasing the precision of a program analysis leads to worse results. It is our thesis that this phenomenon is the result of fundamental limits on the ability to use precise abstract domains as the basis for inferring strong invariants of programs. We show that bias-variance tradeoffs, an idea from learning theory, can be used to explain why more precise abstractions do not necessarily lead to better results and also provides practical techniques for coping with such limitations. Learning theory captures precision using a combinatorial quantity called the VC dimension. We compute the VC dimension for different abstractions and report on its usefulness as a precision metric for program analyses. We evaluate cross validation, a technique for addressing bias-variance tradeoffs, on an industrial strength program verification tool called YOGI. The tool produced using cross validation has significantly better running time, finds new defects, and has fewer time-outs than the current production version. Finally, we make some recommendations for tackling bias-variance tradeoffs in program analysis.

7.3 Abstraction Refinement for Program Analyses

In [18], a central task for a program analysis concerns how to efficiently find a program abstraction that keeps only information relevant for proving properties of interest. We present a new approach for finding such abstractions for program analyses written in Datalog (a standard formalism for expressing program analyses). Our approach is based on counterexample-guided abstraction refinement: when a Datalog analysis run fails using an abstraction, it seeks to generalize the cause of the failure to other abstractions, and pick a new abstraction that avoids a similar failure. Our solution uses a boolean satisfiability formulation that is general, complete, and optimal: it is independent of the Datalog solver, it generalizes the failure of an abstraction to as many other abstractions as possible, and it identifies the cheapest refined abstraction to try next. We show the performance of our approach on a pointer analysis and a typestate analysis, on eight real-world Java benchmark programs.

7.4 User-Guided Program Analysis

Program analysis tools often produce undesirable output due to various approximations. In [19], we present an approach and a system Eugene that allows user feedback to guide such approximations towards producing the desired output. We formulate the problem of user-guided program analysis in terms of solving a combination of hard rules and soft rules: hard rules capture soundness while soft rules capture degrees of approximations and preferences of users. Our technique solves the rules using an off-the-shelf solver in a manner that is sound (satisfies all hard rules), optimal (maximally satisfies soft rules), and scales to real-world analyses and programs. We evaluate Eugene on two different analyses with labeled output on a suite of seven Java programs of size 131-198 KLOC. We also report upon a user study involving nine users who employ Eugene to guide an information-flow analysis on three Java micro-benchmarks. In our experiments, Eugene significantly reduces misclassified reports upon providing limited amounts of feedback.

8 ACRONYMS/GLOSSARY

CFA	Control-Flow Analysis
CFL	Context-Free Language
SDK	Software Developers Kit
SMT	Satisfiability Modulo Theories
STAMP	STatic Analysis of Mobile Programs

References

[1] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *Proceddings of the Conference on Programming Language Design and Implementation*, June 2014.

[2] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2014.

[3] G. Castelnuovo, M. Naik, N. Rinetzky, M. Sagiv, and H. Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *Proceedings of the International Static Analysis Symposium*, September 2015.

[4] Y. Feng, X. Wang, I. Dillig, and T. Dillig. Bottom-up context-sensitive pointer analysis for java. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, November 2015.

[5] Y. Feng, X. Wang, I. Dillig, and C. Lin. Explorer : Query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications*, October 2015.

[6] R. Mangal, M. Nak, and H. Yang. A correspondence between two approaches to interprocedural analysis in the presence of joins. In *Proceedings of the European Symposium on Programming*, April 2014.

[7] S. Anand, M. Naik, H. Yang, and M. J. Harrold. Automated concolic testing of smarthphone apps. In *Proceedings of the International Symposium on Foundations of Software Engineering*, August 2012.

[8] A. MacHiry, R. Tahiliani, and M. Naik.

[9] R. Sharma, E. Schkufza, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the Conference on Object Oriented Systems, Programming Languages, and Applications*, October 2013.

[10] O. Bastani, S. Anand, and A. Aiken. An interactive approach to mobile app verification, booktitle=Proceedings of the Workshop on Mobile Development Lifecycle, month = oct, year = 2015.

[11] O. Bastani, S. Anand, and A. Aiken. Interactively verifying absence of explicit information flows in android apps. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications*, October 2015.

[12] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the European Symposium on Programming*, March 2013.

[13] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification), booktitle=Proceedings of the Asia Conference on Programming Languages and Systems, month = dec, year = 2013.

[14] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free reachability. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2015.

[15] L. Clapp, S. Anand, and A. Aiken.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Semantics-based detection of android malware through static analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering*, November 2014.

[17] R. Sharma, A. Nori, and A. Aiken. Bias-variance tradeoffs in program analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2014.

[18] X. Zhang, R. Mangal, M. Naik, and Yang H. On abstraction refinement for program analyses in datalog. In *Proceddings of the Conference on Programming Language Design and Implementation*, June 2014.

[19] R. Mangal, X. Zhang, A. Nori, and M. Naik. A user-guided approach to program analysis. In *Proceedings of the Symposium on Foundations of Software Engineering*.