Reflective Entry Point Inference with Symbolic Execution

David Van Horn
Computer Science, University of Maryland, College Park MD 20742
dvanhorn@cs.umd.edu, +1 202-460-4104
Google sponsor: Jinseong Jeon, Mads Ager

Abstract

Android and other event-driven programs have a complex control-flow structure with multiple entry points, which must be listed explicitly for programming tools that process such code, like the R8 program minimizer. Enumerating entry points is tedious and easy to get wrong, particularly when applications dynamically compute them using reflection, a ubiquitous practice in large, real-world Android libraries. Mistakes can render tools like R8 ineffective or even dangerous, thwarting good code generation or causing unintended crashes after deployment. We propose to develop the theory and pragmatics of *automated entry point inference* in the setting of Android applications. We will develop new techniques and tools based on symbolic execution to help developers quickly and effectively get entry point specifications correct, even when using reflective idioms. We will evaluate the effectiveness of our approach with an empirical study of using our tool to generate R8 entry point specifications.

1 Research Problem

Modern event-driven software architectures necessitate that program components provide several application entry points to handle the myriad of events that may invoke a component's behaviors. Automated programming tools such as optimizers, minimizers, fuzzers, and static analyzers must be informed of these entry points, usually through programmer supplied listings. Constructing such lists can be quite difficult, particularly when code uses reflection to compute entry points dynamically rather than statically. Programmers faced with this tedious task can make two kinds of mistakes: (1) they overestimate and list more entry points than the application actually contains, or (2) they underestimate and miss some entry points. Both mistakes have negative consequences. Mistakes of the first form cause the tool to be ineffective: optimizations are needlessly thwarted, minimization is undermined by code appearing reachable when it's not, and static analyzers report false positives due to impossible program paths. Mistakes of the second form cause even more serious problems: code is unsafely optimized, minimized, or falsely verified, all of which can lead to run-time crashes, undefined behavior, or security vulnerabilities.

As a concrete example, consider Google's **Android** platform and the Google **R8** program shrinking and minimization tool.[1] Android is a prominent, widely used event-driven architecture and requires applications to provide entry points for various activities, services, and broadcasts. R8 operates on Android applications and aims to convert Java byte code to small, efficient dex code. R8 performs a form of reachability analysis in order to prune dead code and other optimizations. This can have dramatic reduction on code size since applications are often linked against large libraries but usually rely on a small subset of functionality. The R8 tool requires developers to list "keep rules" to specify application entry points. When

[1] https://r8.googlesource.com/r8

developers make mistakes and omit entry points, this can lead to run-time crashes, often with inscrutable error messages, as was the case in a recently reported (Aug 8, 2018) issue[2] wherein a user experienced an `UnsupportedOperationException` complaining that an "Abstract class can't be instantiated!" at run-time in their R8 optimized code. The user was surprised, since a review of the code revealed no abstract class definitions! After review by the R8 team, it was determined that R8 had "optimized" some classes to be abstract and eliminated their constructors since it saw no instances of the class being created. In reality, instances were constructed, but done so using reflection, and the reachability of these constructors had mistakenly been left off of the "keep rules" for the application.

2 Approach and Research Challenges

We propose to use and develop techniques for the symbolic execution (SE) of event-driven, reflective applications that will solve the entry point listing problem by automatically discovering entry point specifications and discrepancies with manually constructed specifications. SE is an effective semantics-based approach to automated testing and verification of software.

There are several research challenges to overcome to achieve effective entry point inference in this setting. First, existing SE techniques tend to (a) focus on reachability of crashing program states and (b) apply to batch-oriented, single entry-point programs where symbolic inputs range over basic types of values (bytes, machine integers, strings, etc.). We will adapt SE to the setting of entry-point inference and the inverted control structure of event-driven programs. Doing so will involve developing useful abstractions of *behavioral* symbolic values such as call-back objects [9]. Second, programs making extensive use of reflection will require precise symbolic reasoning about values used as arguments to reflective methods. In particular, we anticipate the symbolic executor will need to be integrated with a constraint solver with a high-quality theory of strings [7]. Third, modeling the Android run-time is a well-known and substantial undertaking for SE engines targeting the platform [4, 1]. We will take advantage of recent approaches to automatically deduce representations of the run-time system [2]. We anticipate this will require reformulation to represent information relevant to constructor reachability for entry point inference. Putting these pieces together, we aim to build and evaluate the effectiveness of a tool based on the synthesis of these three technical advances.

3 Specific Outcomes

We will implement a semantic-based tool for whole-program Android applications to automatically generate Proguard format rules that can be used with the R8 program minimizer. We will assemble a corpus of existing Android applications and manually constructed entry point specifications, such as the one in the aforecited R8 issue, that can be used to test and evaluate our system. The tool will be evaluated for its effectiveness in discovering both missing and unnecessary manual entry point listings.

4 Differentiation from Prior Work

To the best of our knowledge, there has been no prior research on inferring program entry points in reflective, event-driven programs. There has been significant and relevant prior work on the symbolic execution of Android applications, such as Symdroid [3] and JPF-Android [1], which will inform this project. However that work does not focus on entry-point inference and instead, like Google's R8, requires explicit listings

[2] https://issuetracker.google.com/issues/112386012

or models of entry points, which leads to unsound reasoning in the case of incomplete listings. Other work on static analysis of Android application requires similar assumptions and in general, sound static analysis techniques do not handle reflection or dynamic language features well [5].

The PI has recently led the development of "higher-order" SE [9, 8], which extends traditional SE techniques to handle behavioral values such as procedures and objects, a necessary foundation for reasoning about event-driven object-oriented programs such as Android applications. That work was done in the context of much simpler, single-threaded, functional programs. The proposed work will require significant advancements to work for Android applications, but will in turn lay the foundations for a much larger class of event-driven programs, even beyond Android.

5 Context of Funding

The PI has previously developed sophisticated semantics-based tools for the Android platform as part of a DARPA funded effort to automatically detect malware. This effort produced Anadroid, a static malware analysis framework for Android [5, 6]. One of the main technical challenges in that work was the sound modeling of asynchronous execution of multiple entry points, but these were assumed to be statically known. This work compliments and advances those tools, as well as basically any tool for event-driven programs.

References

[1] Heila Botha, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. Addressing challenges in obtaining high coverage when model checking android applications. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 31–40, New York, NY, USA, 2017. ACM.

[2] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 419–429, New York, NY, USA, 2018. ACM.

[3] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park, July 2012.

[4] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 156–167, New York, NY, USA, 2016. ACM.

[5] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, 2013.

[6] Shuying Liang, Matthew Might, and David Van Horn. Anadroid: Malware analysis of android with user-supplied predicates. In *Workshop on Tools for Automatic Program Analysis*, June 2013.

[7] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, June 2016.

[8] Phúc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 446–456, New York, NY, USA, 2015. ACM.

[9] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 537–554, New York, NY, USA, 2012. ACM.

A Data Policy

Advances made in this project will be published in peer-reviewed venues and/or technical reports. We intend to release software developed for this project as open source.

B Budget

The proposed budget of $55,899 includes only direct costs. This budget supports one PhD student for a year, including benefits, CALF fees (for using computing facilities at the university), and conference travel.

David Van Horn

Assistant Professor
Department of Computer Science & UMIACS
dvanhorn@cs.umd.edu
http://ter.ps/dvanhorn

3439 A.V. Williams Building
University of Maryland
College Park, MD 20742

A. Professional Preparation

University of Vermont	Burlington, VT	Computer Science & Information Systems	B.S.	2003
University of Vermont	Burlington, VT	Computer Science	M.S.	2006
Brandeis University	Waltham, MA	Computer Science	Ph.D.	2009

B. Appointments

University of Maryland	Assistant Professor	2013–present
Northeastern University	Research Assistant Professor	2011–2013
Northeastern University	CRA/CCC CIFellow Postdoctoral Researcher	2009–2011

C. Publications

Publications relevant to project

1. **POPL, 2018:** Soft Contract Verification for Higher-order Stateful Programs. Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn.
2. **POPL, 2016:** Pushdown Control-Flow Analysis for Free. Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn.
3. **PLDI, 2015:** Relatively Complete Counterexamples for Higher-order Programs. Phúc C. Nguyễn and David Van Horn.
4. **ICFP, 2014:** Soft Contract Verification. Phúc C. Nguyễn, Sam Tobin-Hochstadt and David Van Horn.
5. **ICFP, 2010:** Abstracting Abstract Machines. David Van Horn and Matthew Might.

Selected other publications

1. **ICFP, 2017:** Abstracting Definitional Interpreters. David Darais, Phúc C. Nguyễn, Nicholas Labich.
2. **ICFP, 2016:** Constructive Galois Connections. David Darais and David Van Horn.
3. **OOPSLA, 2015:** Incremental Computation with Names. Matthew Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, David Van Horn.
4. **OOPSLA, 2015:** Galois Transformers and Modular Abstract Interpreters. David Darais, Matthew Might, and David Van Horn.
5. **OOPSLA, 2012:** Higher-Order Symbolic Execution via Contracts. Sam Tobin-Hochstadt and David Van Horn.

D. Synergistic Activities

1. Curricular development of the ***How to Design Programs*** **(CS1) "Honors"** course and ***How to Design Classes*** **(CS2) "Honors"**' course at Northeastern University, still in use today. After designing the course, I taught the first several instances of the classes. The design of the CS2 course involved the design and implementation of an object-oriented pedagogical programming language and development environment, which was described in "From Principles to Practice with Class in the First Year" published in the International Workshop on Trends in Functional Programming in Education, 2013.

   These courses form the basis for new CMSC 131A and 132A ***Systematic Program Design, I & II***, offered by the PI in AY 2017–18 at UMD. The course was offered as an alternative introductory track for CS majors with the expectation they will become the standard required courses for majors replacing the existing Java-based courses.
2. ***Realm of Racket: Learn to Program, One Game at a Time!*** (NoStarch Press, 2013), an illustrated book aimed at first-year students that teaches how to program distributed video games in Racket. Co-authored and illustrated with eight undergraduate students from my *How to Design Programs* honors course at Northeastern. The book is used as a supplemental text in the *How to Design Programs* (CS1) course at Northeastern University. All of the video games developed in the book now ship with the standard distribution of Racket.
3. ***An Introduction to Redex with Abstracting Abstract Machines***, a tutorial covering programming language concepts required for the practicing PL researcher such as reduction semantics and type systems. The exposition uses the semantic engineering tool Redex to formulate and test models of programming languages. It also covers a detailed construction of the PI's *Abstracting Abstract Machines* method for systematically deriving abstract interpreters.

   The tutorial has been presented at the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), the SIGPLAN Programming Languages Mentoring Workshop (PLMW), the Oregon Programming Languages Summer School (OPLSS), the PLT Redex Summer School at the University of Oregon, the NII International Workshop on Automated Techniques for Higher-Order Program Verification, the Daghstuhl Seminar on "Scripting Languages and Frameworks: Analysis and Verification," and at invited seminars at Johns Hopkins University and the University of Chile.

   The written tutorial is available on the web (`http://ter.ps/redexaam`), along with videos from the OPLSS presentation (`http://ter.ps/oplssvideos`).
4. Curricular development of ***CMSC 631: Program Analysis and Understanding*** at UMD, a PhD-level course covering programming language theory and abstract interpretation. The course covers topics such as operational semnatics, type systems, type soundness, program verification, symbolic execution, and abstract interpretation. It takes a constructive approach to building, validating, and verifying models of programming languages and their artifacts. The course development has included an extensive set of written notes and software.
5. Steering committees: the ACM SIGPLAN International Conference on Functional Programming, the Workshop on Higher-Order Program Analysis, and the Symposium on Trends in Functional Programming. Program-, review-, or external-review committee member for: POPL (3), ICFP (3), OOPSLA, ESOP, ECOOP, PADL (2), SAS, TFP, OBT, Scheme, Scala, and others. Referee for *J. Func. Prog.*, *ACM TOPLAS*, *HOSC*, *ACM TOCL*, *ACM Comp. Sur.*, *Sci. of Comp. Prog.*. Reviewer for ICFP, POPL, OOPSLA, VMCAI, LICS, CSL, DLS.