
AFRL-RI-RS-TR-2017-008

# SCALABLE AND PRECISE ABSTRACTION OF PROGRAMS FOR TRUSTWORTHY SOFTWARE

UNIVERSITY OF UTAH

*JANUARY 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-2017-008 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)* |
|---|---|---|
| JAN 2017 | FINAL TECHNICAL REPORT | FEB 2012 – JUN 2016 |

**4. TITLE AND SUBTITLE**

SCALABLE AND PRECISE ABSTRACTION OF PROGRAMS FOR TRUSTWORTHY SOFTWARE

**5a. CONTRACT NUMBER**
FA8750-12-2-0106

**5b. GRANT NUMBER**
N/A

**5c. PROGRAM ELEMENT NUMBER**
61101E

**6. AUTHOR(S)**

Matthew Might, William Byrd

**5d. PROJECT NUMBER**
APAC

**5e. TASK NUMBER**
97

**5f. WORK UNIT NUMBER**
75

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

University of Utah – School of Computing
50 S Central Campus Drive, Room 3190
Salt Lake City, Utah 84112

**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-2017-008

**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**

Applications deployed on mobile devices play a critical role in the fabric of national cyberinfrastructure. They carry sensitive data and have capabilities with significant social and financial effect. Yet while it is paramount that such software is trustworthy, these applications pose challenges beyond the reach of current practice for low-cost, high-assurance verification and analysis. This effort investigates a systematic and scalable approach to the fully automatic analysis and verification of applications deployed on mobile devices.

**15. SUBJECT TERMS**

Malware Detection for Android applications, Binary 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 K. WILLIAMS** |
| U | U | U | UU | 20 | 19b. TELEPHONE NUMBER (*Include area code*) **N/A** |

## TABLE OF CONTENTS

# 1.0 SUMMARY

## 1.1 Problem Description

Applications deployed on mobile devices play a critical role in the fabric of national cyberinfrastructure. They carry sensitive data and have capabilities with significant social and financial effect. Yet while it is paramount that such software is trustworthy, these applications pose challenges beyond the reach of current practice for low-cost, high-assurance verification and analysis. These programs are large, modular, and interactive. They communicate with distributed services that are fundamentally unavailable for analysis. They are written in expressive high-level, higher-order programming languages, for which traditional "Fortran-style" approaches to analysis simply do not apply.

We propose to investigate a systematic and scalable approach to the fully automatic analysis and verification of applications deployed on mobile devices.

To make program analysis for cybersecurity economically feasible, scalability and precision must both improve by orders of magnitude. We boldly conjecture scalability and precision—often seen as competing concerns—are inseparable instances of the same problem. Based on recent work, we hypothesize that to significantly improve scalability, precision must be *increased* to such a degree that swaths of false-positive analytic state-space are eliminated.

## 1.2 Research Goals

Our primary goal is to enable sound, secure, automatic program analysis for the elimination of security vulnerabilities in mobile applications written in high-level programming languages. Our research goal at the beginning of this project was to develop the following techniques and artifacts:

### 1.2.1 A method for the systematic abstraction of object-oriented languages.

Van Horn and Might's systematic abstraction exposes the full inventory of parameters to tune the precision of a static analysis. These parameters induce an analytic framework that spans a continuum from the null analysis up to the concrete semantics, with intermediate points that include classical data-flow analysis; rich abstract interpretations; and symbolic execution.

### 1.2.2 A family of locally non-monotonic techniques for abstract transfer functions.

We propose "locally non-monotonic transfer functions," and argue that they are essential to reducing false positives. Locally non-monotonic analyses can revoke judgments made across transition between states, thereby avoiding the inevitable merging that creates false positives. Global monotonicity still guarantees termination.

### 1.2.3 A mobile contract infrastructure for Java, with corresponding higher-order generalizations of relational abstract domains.

Contracts are executable specifications that sit at the boundary between software components. Contracts are a run-time enforcement mechanism, but our approach will leverage

contracts as symbolic values for compile-time symbolic execution. By carrying out symbolic execution with contracts, we can verify rich behavioral properties with minimal false-positives. A higher-order generalization of relational abstract domains—entangled abstract domains—will enable us to conduct such symbolic execution.

### 1.3 Expected Impact

We aim for simultaneous orders-of-magnitude improvements over state- of-the art in the scalability and precision of static analysis. The differentiating keystone of our technical strategy is to assault scalability through *increased* precision. Conventional wisdom in the field holds that precision comes only at the cost of additional analysis time.

The argument goes that increasing context-sensitivity—allocating more abstract variants of concrete addresses to boost polyvariance—will increase the size of the abstract state- space, and therefore, the worst case (and, in practice, the actual) analysis time.

Though linked, *context-sensitivity is not the same as precision*.

The leveragable hypothesis of our proposed effort is that cracks in the conventional wisdom on static analysis can and will be pried open. Early cracks such as Wright and Jagannathan's work on polymorphic analysis showed analysis time can fall even as precision (and worst-case complexity) increase. The message of that work was clear: it is not the number of contexts that matter—it is when they're allocated (and just as critically, when they are not). Co-PI Van Horn extended this understanding by proving that Shivers' venerable context-sensitive k-CFA occupies a point in the design space that runs in exponential time, yet can only learn a polynomial number of true facts about a program. In other words, k-CFA is *hard and imprecise*; it spends its time computing junk. A more recent crack—PI Might's work on abstract garbage collection—makes more efficient use of any finite set of contexts under any allocation strategy, yielding simultaneous order-of-magnitude improvements in both time and precision.

Extrapolating from these points, we argue that there is a region, in which locally non-monotonic methods are employed, where more precision cleaves so much false positive state-space and spurious rediscovery that analysis time falls.

## 2.0 METHODS ASSUMPTIONS AND PROCEDURES

Making automated program analysis for software security economically feasible requires substantial improvements to analytic precision and scalability. At a high level, our technical plan is to assault scalability through targeted strikes on precision. In brief, we seek to augment precision as necessary to amputate strictly over-approximating false-positive analytic state-space and to avoid spurious rediscovery of facts.

Our experience in static analysis of higher-order systems leads us to conjecture that traditional higher-order analyses waste so much time in over-approximating territory that considerable effort can be profitably expended in its elimination, raising both precision and performance simultaneously.

**Technical keystone: Local non-monotonicity**. In the heap-cloning, path-sensitive, context-insensitive, whole-program analysis of a ten-line doubly nested loop constructed from higher-order functions, the abstract transition graph is disproportionately large. A manual inspection of this state-space finds that more than 90% of it is strictly over- approximating; that is, it is considering interleavings of control-flow (and therefore data- flow) that cannot occur. The fundamental cause of this over-approximation is the finite abstraction inflicted on data and control combined with the monotonic growth of judgments during analysis.

By making individual transitions in a small-step abstract interpretation able to revoke judgments (for example, those of the form variable f may be bound to procedure p), we can collapse the transition graph during construction. In this case, the locally non-monotonic refinement was PI Might's abstract garbage garbage collection.

This is a radical departure from the traditional analysis of higher-order programs. We intend to develop and leverage locally non-monotonic small-step abstract interpretations to achieve extreme precision, and with it, performance.

### 2.1 Summary of Project Milestones

- 6 months: A specification for core Java.
- 12 months: An abstract machine for core Java.
- 12 months: A contract calculus for core Java.
- 14 months: A systematic abstraction of core Java.
- 18 months: A security auditor for core Java.
- 24 months: A contract calculus for full Java.
- 28 months: An abstract machine for full Java.
- 32 months: A systematic abstraction of full Java.
- 36 months: A security auditor for full Java.
- 42 months: A web-deployed service for security auditing.

## 4.0 RESULTS AND DISCUSSIONS

The work we performed during the APAC project improved the state of the art in static analysis, with regards to both basic science and applications of static analysis to security. This section reviews a number of the technical contributions we made through the APAC project and their importance.

### 4.1 Hash-Flow Taint Analysis of Higher-Order Programs

As web applications have grown in popularity, so have attacks on such applications. Cross-site scripting and injection attacks have become particularly problematic. Both vulnerabilities stem, at their core, from improper sanitization of user input.

We propose static taint analysis, which can verify the absence of unsanitized input errors at compile-time. Unfortunately, precise static analysis of modern scripting languages like Python is challenging: higher-orderness and complex control-flow collide with opaque, dynamic data structures like hash maps and objects. The interdependence of data-flow and control-flow make it hard to attain both soundness and precision.

In one of our papers[1], we apply abstract interpretation to sound and precise taint-style static analysis of scripting languages. We first define lambda-H, a core calculus of modern scripting languages, with hash maps, dynamic objects, higher-order functions and first class control. Then we derive a framework of k-CFA-like CESK-style abstract machines for statically reasoning about lambda-H, but with hash maps factored into a “Curried Object store.” The Curried object store—and shape analysis on this store—allows us to recover field sensitivity, even in the presence of dynamically modified fields. Lastly, atop this framework, we devise a taint-flow analysis, leveraging its field-sensitive, interprocedural and context-sensitive properties to soundly and precisely detect security vulnerabilities, like XSS attacks in web applications.

We have prototyped the analytical framework for Python, and conducted preliminary experiments with web applications. A low rate of false alarms demonstrates the promise of this approach.

### 4.2 Introspective Pushdown Analysis of Higher-Order Programs

In the static analysis of functional programs, pushdown flow analysis and abstract garbage collection skirt just inside the boundaries of soundness and decidability. Alone, each method reduces analysis times and boosts precision by orders of magnitude. Our work[2] illuminates and conquers the theoretical challenges that stand in the way of combining the power of these techniques. The challenge in marrying these techniques is not subtle: computing the reachable control states of a pushdown system relies on limiting access during transition to the top of the stack; abstract garbage collection, on the other hand, needs full access to the entire stack to compute a root set, just as concrete collection does. Introspective pushdown systems resolve this conflict. Introspective pushdown systems provide enough access to the stack to allow abstract garbage collection, but they remain restricted enough to compute control-state reachability, thereby enabling the sound and precise product of pushdown analysis and abstract garbage collection. Experiments reveal synergistic interplay between the techniques, and the fusion demonstrates "better-than-both-worlds" precision.

### 4.3 Structural Soundness Proof for Shivers's Escape Technique: A Case for Galois Connections

Shivers's escape technique enables one to analyse the control flow of higher-order program fragments. It is widely used, but its soundness has never been proven. In this paper[3] we present the first soundness proof for the technique. Our proof is structured as a composition of Galois connections and thus rests on the foundations of abstract interpretation.

### 4.4 Higher-Order Symbolic Execution via Contracts

In this paper[4] we present a new approach to automated reasoning about higher-order programs by extending symbolic execution to use behavioral contracts as symbolic values, thus enabling symbolic approximation of higher-order behavior.

Our approach is based on the idea of an abstract reduction semantics that gives an operational semantics to programs with both concrete and symbolic components. Symbolic components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is an executable semantics that soundly predicts program behavior, including contract failures, for all possible instantiations of symbolic components. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. Supporting this rich language of specifications leads to powerful symbolic reasoning using existing program constructs.

We then apply our approach to produce a verifier for contract correctness of components, including a sound and computable approximation to our semantics that facilitates fully automated contract verification. Our implementation is capable of verifying contracts expressed in existing programs, and of justifying contract-elimination optimizations.

### 4.5 Monadic Abstract Interpreters

Recent developments in the systematic construction of abstract interpreters hinted at the possibility of a broad unification of concepts in static analysis. We deliver that unification by showing context-sensitivity, polyvariance, flow-sensitivity, reachability-pruning, heap-cloning and cardinality-bounding to be independent of any particular semantics. Monads become the unifying agent between these concepts and between semantics. For instance, by plugging the same "context-insensitivity monad" into a monadically-parameterized semantics for Java or for the lambda calculus, it yields the expected context-insensitive analysis.

To achieve this unification, we develop a systematic method for transforming a concrete semantics into a monadically-parameterized abstract machine. Changing the monad changes the behavior of the machine. By changing the monad, we recover a spectrum of machines-from the original concrete semantics to a monovariant, flow- and context-insensitive static analysis with a singly-threaded heap and weak updates.

The monadic parameterization also suggests an abstraction over the ubiquitous monotone fixed-point computation found in static analysis. This abstraction makes it straightforward to instrument an analysis with high-level strategies for improving precision and performance, such as abstract garbage collection and widening. While our paper[5] itself runs the development for continuation-passing style, our generic implementation replays it for direct-style lambda-calculus and Featherweight Java to support generality.

### 4.6 AnaDroid: Malware Analysis of Android with User-Supplied Predicates

Today's mobile platforms provide only coarse-grained permissions to users with regard to how third-party applications use sensitive private data. Unfortunately, it is easy to disguise malware within the boundaries of legitimately-granted permissions. For instance, granting access to "contacts" and "internet" may be necessary for a text-messaging application to function, even though the user does not want contacts transmitted over the internet. To understand fine-grained application use of permissions, we need to statically analyze their behavior. Even then, malware detection faces three hurdles: (1) analyses may be prohibitively expensive, (2) automated analyses can only find behaviors that they are designed to find, and (3) the maliciousness of any given behavior is application-dependent and subject to human judgment. To remedy these issues, we propose[6] semantic-based program analysis, with a human in the loop as an alternative approach to malware detection. In particular, our analysis allows analyst-crafted semantic predicates to search and filter analysis results. Human-oriented semantic-based program analysis can systematically, quickly and concisely characterize the behaviors of mobile applications. We describe a tool that provides analysts with a library of the semantic predicates and the ability to dynamically trade speed and precision. It also provides analysts the ability to statically inspect details of every suspicious state of (abstract) execution in order to make a ruling as to whether or not the behavior is truly malicious with respect to the intent of the application. In addition, permission and profiling reports are generated to aid analysts in identifying common malicious behaviors.

### 4.7 Optimizing Abstract Abstract Machines

The technique of abstracting abstract machines (AAM) provides a systematic approach for deriving computable approximations of evaluators that are easily proved sound. In this paper[7] we contribute a complementary step-by-step process for subsequently going from a naive analyzer derived under the AAM approach, to an efficient and correct implementation. The end result of the process is a two to three order-of-magnitude improvement over the systematically derived analyzer, making it competitive with hand-optimized implementations that compute fundamentally less precise results.

### 4.8 Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation

Sound malware analysis of Android applications is challenging. First, object-oriented programs exhibit highly interprocedural, dynamically dispatched control structure. Second, the Android programming paradigm relies heavily on the asynchronous execution of multiple entry points. Existing analysis techniques focus more on the second challenge, while relying on traditional analytic techniques that suffer from inherent imprecision or unsoundness to solve the first.

We present[8] Anadroid, a static malware analysis framework for Android apps. Anadroid exploits two techniques to soundly raise precision: (1) it uses a pushdown system to precisely model dynamically dispatched interprocedural and exception-driven control-flow; (2) it uses Entry-Point Saturation (EPS) to soundly approximate all possible interleavings of asynchronous entry points in Android applications. (It also integrates static taint-flow analysis and least permissions analysis to expand the class of malicious behaviors which it can catch.)

Anadroid provides rich user interface support for human analysts which must ultimately rule on the “maliciousness” of a behavior.

To demonstrate the effectiveness of Anadroid’s malware analysis, we had teams of analysts analyze a challenge suite of 52 Android applications released as part of the Automated Program Analysis for Cybersecurity (APAC) DARPA program. The first team analyzed the apps using a version of Anadroid that uses traditional (finite-state-machine-based) control-flow-analysis found in existing malware analysis tools; the second team analyzed the apps using a version of Anadroid that uses our enhanced pushdown-based control-flow-analysis. We measured machine analysis time, human analyst time, and their accuracy in flagging malicious applications. With pushdown analysis, we found statistically significant ($p < 0.05$) decreases in time: from 85 minutes per app to 35 minutes per app in human plus machine analysis time; and statistically significant ($p < 0.05$) increases in accuracy with the pushdown-driven analyzer: from 71% correct identification to 95% correct identification.

### 4.9 Entangled Abstract Domains for Higher-order Programs

Relational abstract domains are a cornerstone of static analysis for first-order programs. In this paper[9] we explore challenges in generalizing relational abstract domains to higher-order program analysis. We find two reasonable, orthogonal and complementary interpretations of relational domains in a higher-order setting. The first technique, locally relational abstract domains, are relational abstract domains that travel with the environments found within closures. These abstract domains record invariants discovered within a given scope. The second technique, globally entangled abstract domains, allows relational abstract domains to quantify over the concrete constituents of an abstract resource. This approach enables the discovery of interprocedural, interstructural and intrastructural program invariants. We develop the techniques for a lambda calculus enriched with structs. We structurally abstract the concrete semantics; we develop a logic for both the local and global generalizations; and then we integrate both logics into the abstraction. By restricting the logics, existing relational abstract domains (or their entangled, higher-order generalizations) are recoverable. To demonstrate the applicability of the framework, higher-order variants of both octagon and polyhedral domains are formulated as such restrictions.

### 4.10 Multi-core Parallelization of Abstracted Abstract Machines

It is straightforward to derive well-known higher-order flow analyses as abstract interpretations of well-known abstract machines. In our paper[10], we explore multi-core parallel evaluation of one such abstract abstract machine, the CES machine. The CES machine is a variant of CESK machines that runs Continuation Passing Style (CPS) lambda-calculus. Using k-CFA, the concrete semantics for a CES machine can be turned into abstract semantics. Analyzing a program for this machine is a state graph walk, which can be run in parallel to increase performance.

### 4.11 A Unified Approach to Polyvariance in Abstract Interpretations

In this paper[11] we describe an approach to exploring polyvariance in abstract interpretations by exposing the allocation of abstract bindings as an analysis parameter. This allocation policy is a method for selecting abstract addresses based on the current state of

execution. As addresses are chosen from a finite set, the allocation policy is responsible for determining the exact degree of merging which occurs between different values at a given point in the analysis. This approach allows us to select any kind of polyvariance desired through the selection of an abstract allocation function. We show how this can be done for an intermediate representation of Scheme, implementing a sound parametric framework for such analyses. We distinguish three disparate interpretations of the k-CFA hierarchy and compare them, instantiating each within our framework and motivating our approach.

**4.12 Pushdown flow analysis with abstract garbage collection.**

In the static analysis of functional programs, pushdown flow analysis and abstract garbage collection push the boundaries of what we can learn about programs statically. Our work[12] illuminates and poses solutions to theoretical and practical challenges that stand in the way of combining the power of these techniques. Pushdown flow analysis grants unbounded yet computable polyvariance to the analysis of return-flow in higher-order programs. Abstract garbage collection grants unbounded polyvariance to abstract addresses which become unreachable between invocations of the abstract contexts in which they were created. Pushdown analysis solves the problem of precisely analyzing recursion in higher-order languages; abstract garbage collection is essential in solving the "stickiness" problem. Alone, our benchmarks demonstrate that each method can reduce analysis times and boost precision by orders of magnitude. We combine these methods. The challenge in marrying these techniques is not subtle: computing the reachable control states of a pushdown system relies on limiting access during transition to the top of the stack; abstract garbage collection, on the other hand, needs full access to the entire stack to compute a root set, just as concrete collection does. Conditional pushdown systems were developed for just such a conundrum, but existing methods are ill-suited for the dynamic nature of garbage collection. We show fully precise and approximate solutions to the feasible paths problem for pushdown garbage-collecting control-flow analysis. Experiments reveal synergistic interplay between garbage collection and pushdown techniques, and the fusion demonstrates "better-than-both-worlds" precision.

**4.13 Fast Flow Analysis with Gödel Hashes**

Flow analysis, such as control-flow, data-flow, and exception-flow analysis, usually depends on relational operations on flow sets. Unfortunately, set related operations, such as inclusion and equality, are usually very expensive. They can easily take more than 97% of the total analyzing time, even in a very simple analysis. We attack this performance[13] bottleneck by proposing Gödel hashes to enable fast and precise flow analysis. Gödel hashes is an ultra compact, partial-order-preserving, fast and perfect hashing mechanism, inspired by the proofs of Gödel’s incompleteness theorems. Compared with array-, tree-, traditional hash-, and bit vector-backed set implementations, we find Gödel hashes to be tens or even hundreds of times faster for performance in the critical operations of inclusion and equality. We apply Gödel hashes in real-world analysis for object-oriented programs. The instrumented analysis is tens of times faster than the one with original data structures on DaCapo benchmarks.

### 4.14 Pruning, Pushdown Exception-Flow Analysis

Statically reasoning in the presence of exceptions and about the effects of exceptions is challenging: exception-flows are mutually determined by traditional control-flow and points-to analyses. We tackle[14] the challenge of analyzing exception-flows from two angles. First, from the angle of pruning control-flows (both normal and exceptional), we derive a pushdown framework for an object-oriented language with full-featured exceptions. Unlike traditional analyses, it allows precise matching of throwers to catchers. Second, from the angle of pruning points-to information, we generalize abstract garbage collection to object-oriented programs and enhance it with liveness analysis. We then seamlessly weave the techniques into enhanced reachability computation, yielding highly precise exception-flow analysis, without becoming intractable, even for large applications. We evaluate our pruned, pushdown exception-flow analysis, comparing it with an established analysis on large scale standard Java benchmarks. The results show that our analysis significantly improves analysis precision over traditional analysis within a reasonable analysis time.

### 4.15 Deletion: The curse of the red-black tree

Okasaki introduced the canonical formulation of functional red-black trees when he gave a concise, elegant method of persistent element insertion. Persistent element deletion, on the other hand, has not enjoyed the same treatment. For this reason, many functional implementations simply omit persistent deletion. Those that include deletion typically take one of two approaches. The more-common approach is a superficial translation of the standard imperative algorithm. The resulting algorithm has functional airs but remains clumsy and verbose, characteristic of its imperative heritage. (Indeed, even the term insertion is a holdover from imperative origins, but is now established in functional contexts. Accordingly, we use the term deletion which has the same connotation.) The less-common approach leverages the features of advanced type systems, which obscures the essence of the algorithm. Nevertheless, foreign-language implementors reference such implementations and, apparently unable to tease apart the algorithm and its type specification, transliterate the entirety unnecessarily. Our paper’s goal[15] is to provide for persistent deletion what Okasaki did for insertion: a succinct, comprehensible method that will liberate implementors. We conceptually simplify deletion by temporarily introducing a "double-black" color into Okasaki's tree type. This third color, with its natural interpretation, significantly simplifies the preservation of invariants during deletion.

### 4.16 Concrete and Abstract Interpretation: Better Together

Recent work in abstracting abstract machines provides a methodology for deriving sound static analyzers from a concrete semantics by way of abstract interpretation. Consequently, the concrete and abstract semantics are closely related by design. We apply[16] Galois-unions as a framework for combining both concrete and abstract semantics, and explore the benefits of being able to express both in a single semantics. We present a methodology for creating such a unified representation using operational semantics and implement our approach with and A-normal form (ANF) lambda-calculus for a CESK style machine in PLT Redex.

### 4.17 A Linear Encoding of Pushdown Control-Flow Analysis

We describe a linear-algebraic encoding for pushdown control-flow analysis of higher-order programs. Pushdown control-flow analyses obtain a greater precision in matching calls with returns by encoding stack-actions on the edges of a Dyck state graph. This kind of analysis requires a number of distinct transitions and was not amenable to parallelization using the approach of EigenCFA. Recent work has extended EigenCFA, making it possible to encode more complex analyses as linear-algebra for efficient implementation on SIMD architectures. We[17] apply this approach to an encoding of a monovariant pushdown control-flow analysis and present a prototype implementation of our encoding written in Octave. Our prototype has been used to test our encoding against a traditional worklist implementation of a pushdown control-flow analysis.

### 4.18 Environment Unrolling

We propose[18] a new way of thinking about abstract interpretation with a method we term environment unrolling. Model checkers for imperative languages will often apply loop unrolling to make their state space finite in the presence of loops and recursion. We propose handling environments in a similar fashion by putting a bound on the number of environments to which a given closure can transitively refer. We present how this idea relates to a normal model of abstract interpretation, give a general overview of its soundness proof in regards to a concrete semantics, and show empirical results demonstrating the effectiveness of our approach.

### 4.19 Strong Function Call

This work[19] presents an incremental improvement to abstract interpretation of higher order languages, similar to strong update, which we term strong function call. In an abstract interpretation, after a function call, we know which abstract closure was called and can deduce that any other values found at the same abstract address in the abstract store could not possibly exist in the corresponding concrete state. Thus they can be removed from the abstract store without loss of soundness. We provide the intuition behind this analysis along with a general overview of its soundness proof.

### 4.20 Static analysis of non-interference in expressive low-level languages

Early work in implicit information flow detection applied only to flat, procedureless languages with structured control-flow (e.g., if statements, while loops). These techniques have yet to be adequately extended and generalized to expressive languages with interprocedural, exceptional and irregular control-flow behavior. We present[20] an implicit information flow analysis suitable for languages with conditional jumps, dynamically dispatched methods, and exceptions. We implement this analysis for the Dalvik bytecode format, the substrate for Android. In order to capture information flows across interprocedural and exceptional boundaries, this analysis uses a projection of a small-step abstract interpreter's rich state graph instead of the control-flow graph typically used for such purposes in weaker linguistic settings. We present a proof of termination-insensitive non-interference. To our knowledge, it is the first analysis capable of proving non-trivial non-interference in a language with this combination of features.

## 5.0 CONCLUSIONS

Primary contributions of our work on APAC include not only theoretical improvements, but also software. AnaDroid[6,21], a static analysis system designed to be used in conjunction with a human analyst, was developed by Utah graduate student Shuying Liang--this work lead directly to her PhD thesis[21].

AnaDroid's static analysis framework includes many advanced features for static analysis and malware detection, including: efficient encodings using Gödel hashes[13], abstract garbage collection combined with liveness analysis, push-down exception-flow analysis [14], and a high-precision taint-flow analysis designed for security applications[1].

Future work for our approach to static analysis for malware detection includes developing library summarization techniques, along with techniques for dynamically adjusting precision of the static analysis.

Analysis of code that comes from a large library can requires an inordinate amount of resources. However, since these libraries are commonly shared among multiple applications, it makes sense to analyze each library once, and share the results of the analysis between all the applications that use that library. This task is known as "library summarization." Library summarization is known to be a difficult task, but should be investigated in the context of our approach to static analysis in order to enhance our technique's ability to run on large code bases.

Both excessively precise and excessively imprecise analyses can make the analysis intractable. The ideal precision for an analysis can differ between different modules of a large application's code base. In order to keep analyses tractable, it will be important to investigate techniques that allow the analysis tool to dynamically adjust its precision within different parts of the application being analyzed, to have high-precision in the parts of the code that benefit from that, and low-precision in the parts of the code that benefit from that.

## 6.0 REFERENCES


[1] Liang, Shuying, Might, Matthew, “Hash-Flow Taint Analysis of Higher-Order Programs,” Programming Languages and Security 2012 (PLAS 2012), Beijing, China, June, 2012.

[2] Earl, Christopher, Sergey, Ilya, Might, Matthew, Van Horn, David, “Introspective Pushdown Analysis of Higher-Order Programs,” International Conference on Functional Programming 2012 (ICFP 2012), Copenhagen, Denmark, September 2012.

[3] Midtgaard, Jan, Adams, Michael D., Might, Matthew. “A Structural Soundness Proof for Shivers's Escape Technique: A Case for Galois Connections,” Static Analysis Symposium 2012 (SAS 2012), Deauville, France, September 2012.

[4] Van Horn, David, Tobin-Hochstadt, Sam, “Higher-Order Symbolic Execution via Contracts,” Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA 2012), Tucson, Arizona, October 2012.

[5] Sergey, Ilya, Devriese, Dominique, Might, Matthew, Midtgaard, Jan, Darais, David, Clark, Dave, Piessens, Frank, “Monadic Abstract Interpreters,” Proceedings of the 34th Annual Conference of Programming Language Design and Implementation (PLDI 2013), Seattle, Washington, 2013.

[6] Liang, Shuying, Might, Matthew, Van Horn, David, “AnaDroid: Malware Analysis of Android with User-Supplied Predicates,” Proceedings of Tools for Automatic Program Analysis 2013 (TAPAS 2013), Seattle, Washington, June 2013.

[7] Johnson, J. Ian, Labich, Nicholas, Might, Matthew, Van Horn, David, “Optimizing Abstract Abstract Machines,” Proceedings of the International Conference on Functional Programming 2013 (ICFP 2013), Boston, Massachusetts, September, 2013.

[8] Liang, Shuying, Keep, Andrew W., Might, Matthew, Van Horn, David, Lyde, Steven, Gilray, Thomas, Aldous, Petey, “Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation,” Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2013), Berlin, Germany, November 2013.

[9] Liang, Shuying, Might, Matthew, “Entangled Abstract Domains for Higher-order Programs,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

[10] Andersen, Leif, Might, Matthew, “Multi-core Parallelization of Abstracted Abstract Machines,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

[11] Gilray,Thomas, Might, Matthew, “A Unified Approach to Polyvariance in Abstract Interpretations,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

[12] Johnson, J. Ian, Sergey, Ilya, Earl, Christopher, Might, Matthew, Van Horn, David, “Pushdown flow analysis with abstract garbage collection,” Journal of Functional Programming, **24**(2-3), pp. 218-283.

[13] Liang, Shuying, Sun, Weibin, Might, Matthew, “Fast Flow Analysis with Gödel Hashes,” 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), Victoria, BC, Canada, September 2014.

[14] Liang, Shuying, Sun, Weibin, Might, Matthew, Keep, Andrew, Van Horn, David, “Pruning, Pushdown Exception-Flow Analysis,” 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), Victoria, BC, Canada, September 2014.

[15] Germane, Kimball, Might, Matthew, “Deletion: The curse of the red-black tree,” Journal of Functional Programming, **24**(4), July 2014, pp. 423-433.

[16] Jenkins, Maria, Andersen, Leif, Gilray, Thomas, Might, Matthew, “Concrete and Abstract Interpretation: Better Together,” Workshop on Scheme and Functional Programming, Washington, D.C., November 2014.

[17] Lyde, Steven, Gilray, Thomas, Might, Matthew Might, “A Linear Encoding of Pushdown Control-Flow Analysis,” Workshop on Scheme and Functional Programming, Washington, D.C., November 2014.

[18] Lyde, Steven, Might, Matthew, “Environment Unrolling,” Workshop on Higher-Order Program Analysis 2014 (HOPA 2014), Vienna, Austria, July 2014.

[19] Lyde, Steven, Might, Matthew, “Strong Function Call,” Workshop on Higher-Order Program Analysis 2014 (HOPA 2014), Vienna, Austria, July 2014.

[20] Aldous, Peter, Might, Matthew, “Static analysis of non-interference in expressive low-level languages,” International Static Analysis Symposium (SAS 2015). Saint-Malo, France. September, 2015.

[21] Liang, Shuying, "Static Analysis of Android Applications," PhD Dissertation, University of Utah, August 2014.

## 7.0 ACRONYMS/GLOSSARY

| | |
|---|---|
| AAM | abstracted abstract machine |
| ANF | administrative normal form |
| API | application program interface |
| CFA | control-flow analysis |
| CES | Control, Environment, Stack |
| CESK | Control, Environment, Stack, (K)ontinuation |
| DoD | Department of Defense |
| DVM | Dalvik Virtual Machine |
| EPS | entry point saturation |
| GUI | graphical user interface |
| HTTP | hypertext transfer protocol |
| JVM | Java Virtual Machine |
| OS | operating system |
| VM | virtual machine |

## APPENDIX: PUBLICATIONS


Aldous, Peter, Might, Matthew, “Static analysis of non-interference in expressive low-level languages,” International Static Analysis Symposium (SAS 2015). Saint-Malo, France. September, 2015.

Andersen, Leif, Might, Matthew, “Multi-core Parallelization of Abstracted Abstract Machines,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

Earl, Christopher, Sergey, Ilya, Might, Matthew, Van Horn, David, “Introspective Pushdown Analysis of Higher-Order Programs,” International Conference on Functional Programming 2012 (ICFP 2012), Copenhagen, Denmark, September 2012.

Germane, Kimball, Might, Matthew, “Deletion: The curse of the red-black tree,” Journal of Functional Programming, 24(4), July 2014, pp. 423-433.

Gilray,Thomas, Might, Matthew, “A Unified Approach to Polyvariance in Abstract Interpretations,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

Jenkins, Maria, Andersen, Leif, Gilray, Thomas, Might, Matthew, “Concrete and Abstract Interpretation: Better Together,” Workshop on Scheme and Functional Programming, Washington, D.C., November 2014.

Johnson, J. Ian, Labich, Nicholas, Might, Matthew, Van Horn, David, “Optimizing Abstract Abstract Machines,” Proceedings of the International Conference on Functional Programming 2013 (ICFP 2013), Boston, Massachusetts, September, 2013.

Johnson, J. Ian, Sergey, Ilya, Earl, Christopher, Might, Matthew, Van Horn, David, “Pushdown flow analysis with abstract garbage collection,” Journal of Functional Programming, 24(2-3), pp. 218-283.

Liang, Shuying, Keep, Andrew W., Might, Matthew, Van Horn, David, Lyde, Steven, Gilray, Thomas, Aldous, Petey, “Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation,” Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2013), Berlin, Germany, November 2013.

Liang, Shuying, Might, Matthew, Van Horn, David, “AnaDroid: Malware Analysis of Android with User-Supplied Predicates,” Proceedings of Tools for Automatic Program Analysis 2013 (TAPAS 2013), Seattle, Washington, June 2013.

Liang, Shuying, Might, Matthew, “Entangled Abstract Domains for Higher-order Programs,” Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, D.C., November 2013.

Liang, Shuying, Might, Matthew, “Hash-Flow Taint Analysis of Higher-Order Programs,” Programming Languages and Security 2012 (PLAS 2012), Beijing, China, June, 2012.

Liang, Shuying, Sun, Weibin, Might, Matthew, Keep, Andrew, Van Horn, David, “Pruning, Pushdown Exception-Flow Analysis,” 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), Victoria, BC, Canada, September 2014.

Liang, Shuying, Sun, Weibin, Might, Matthew, “Fast Flow Analysis with Gödel Hashes,” 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), Victoria, BC, Canada, September 2014.

Lyde, Steven, Gilray, Thomas, Might, Matthew Might, “A Linear Encoding of Pushdown Control-Flow Analysis,” Workshop on Scheme and Functional Programming, Washington, D.C., November 2014.

Lyde, Steven, Might, Matthew, “Environment Unrolling,” Workshop on Higher-Order Program Analysis 2014 (HOPA 2014), Vienna, Austria, July 2014.

Lyde, Steven, Might, Matthew, “Strong Function Call,” Workshop on Higher-Order Program Analysis 2014 (HOPA 2014), Vienna, Austria, July 2014.

Midtgaard, Jan, Adams, Michael D., Might, Matthew. “A Structural Soundness Proof for Shivers's Escape Technique: A Case for Galois Connections,” Static Analysis Symposium 2012 (SAS 2012), Deauville, France, September 2012.

Sergey, Ilya, Devriese, Dominique, Might, Matthew, Midtgaard, Jan, Darais, David, Clark, Dave, Piessens, Frank, “Monadic Abstract Interpreters,” Proceedings of the 34th Annual Conference of Programming Language Design and Implementation (PLDI 2013), Seattle, Washington, 2013.

Van Horn, David, Tobin-Hochstadt, Sam, “Higher-Order Symbolic Execution via Contracts,” Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA 2012), Tucson, Arizona, October 2012.