Man-machine partial program analysis for malware detection

Thomas Norman Deering

Iowa State University
Man-machine partial program analysis for malware detection

by

Thomas Norman Deering

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Suresh Kothari, Major Professor
Srikanta Tirthapura
Manimaran Govindarasu
David Fernandez-Baca
Samik Basu

Iowa State University
Ames, Iowa
2015
DEDICATION

I would like to dedicate this thesis to my wife, Hannah, for her motivation and support, without which I would not have been able to complete this work.
# TABLE OF CONTENTS

**LIST OF FIGURES** ........................................ vii  
**LIST OF TABLES** ........................................ viii  
**ACKNOWLEDGEMENTS** ....................................... ix  
**ABSTRACT** ................................................ x  
**CHAPTER 1. GENERAL INTRODUCTION** ..................... 1  
  1.1 Software Analysis Platform for Automation and Comprehension ............... 3  
  1.2 Man-Machine Detection of Malware ......................................... 4  
  1.3 Library Summaries for Partial Program Analysis .............................. 5  
**CHAPTER 2. REVIEW OF LITERATURE** ....................... 7  
  2.1 Atlas .................................................... 7  
    2.1.1 Program Comprehension ........................................... 8  
    2.1.2 Detecting Defects .............................................. 10  
    2.1.3 Designing Test Cases .......................................... 14  
    2.1.4 Analysis as a Platform ....................................... 14  
  2.2 ISU Security Toolbox .......................................... 15  
    2.2.1 Nature of Android Malware .................................... 15  
    2.2.2 Machine Learning ............................................ 17  
    2.2.3 Dynamic Testing and Sandboxing .................................. 17  
    2.2.4 Changes to the Android Platform ................................ 18  
    2.2.5 Two-Pass Analysis Systems .................................... 19
CHAPTER 5. FLOWMINER: AUTOMATIC EXTRACTION OF LIBRARY DATA-FLOW SEMANTICS FOR PARTIAL PROGRAM ANALYSIS

5.1 Introduction ................................................. 54
5.2 Motivating Example ........................................ 57
5.3 Background: ATLAS Program Analysis Platform .......... 61
5.4 Approach ..................................................... 64
5.5 Automatic Summary Extraction ................................. 68
  5.5.1 Mining Local Flows .................................... 68
  5.5.2 Mining Interprocedural Flows ............................ 71
  5.5.3 Summary Extraction Example ............................ 72
5.6 Implementation .............................................. 74
  5.6.1 A Summary Graph Schema Extension ..................... 77
  5.6.2 A Portable XML Schema ................................. 79
5.7 Discussion .................................................... 79
5.8 Evaluation ..................................................... 80
  5.8.1 Expressiveness ........................................... 80
  5.8.2 Soundness ............................................... 83
  5.8.3 Completeness ........................................... 84
  5.8.4 Experiments .............................................. 86
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Atlas helps comprehend complex software.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>An Atlas graph</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>An example Atlas script</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>ISU detects novel malware using Atlas.</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Graph transformations used to reduce a CFG into an EFG.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Security Toolbox Dashboard Interface</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Smart View showing reverse data flow</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration of Security Toolbox value analysis</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Alignment of two software graphs</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Partial program analysis omits reusable libraries and frameworks.</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Data flows not found due to lack of Android semantics</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Subset of Atlas XCSG schema</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>Partial program graph $G(\mathfrak{P})$ for $\mathfrak{P}$ in Listing 5.3</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>Elided local flow summary $G^S(\mathfrak{P})$ for Figure 5.4</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary extraction results for the Integer class in Listing 5.4</td>
<td>75</td>
</tr>
<tr>
<td>5.7</td>
<td>Architecture of FlowMiner</td>
<td>76</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary nodes and edges in FlowMiner's XCSG extension</td>
<td>78</td>
</tr>
<tr>
<td>5.9</td>
<td>Analysis of malicious app with FlowMiner summaries</td>
<td>81</td>
</tr>
<tr>
<td>5.10</td>
<td>Coarse flow specifications lead to false positive flows</td>
<td>82</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3.1  Artifacts and relationships in Atlas graphs. . . . . . . . . . . . . . . 29

Table 5.1  Experimental results for FLOWMINER . . . . . . . . . . . . . . . . . 85
ACKNOWLEDGEMENTS

Much of the work described in this thesis was developed jointly with colleagues at the Knowledge-Centric Software Lab at Iowa State University. I wish to thank Ben Holland, Sandeep Krishnan, Akshay Deepak, Damanjit Singh, Daniel Stiner, Curtis Ullerich, and Vani Bojja for their contributions to the Security Toolbox, and Ganeshram Santhanam for his contributions to FlowMiner.

I would also like to thank our colleagues at EnSoft, including Jeremias Sauceda, Nikhil Ranade, and Jon Mathews, for developing Atlas [50] and for contributing to the Security Toolbox. Our research at ISU would be much more tedious without this fantastic analysis platform. Thank you for deftly handling support requests and sharing your deep insights into the field of program analysis.

I owe a debt of gratitude to my advisor, Dr. Suresh Kothari, for his unconventional approach, insight, guidance, mentorship, enthusiasm, and sponsorship of my PhD program. I must also thank the ultimate origin of my funding, APAC Program Manager Timothy Fraser and the Defense Advanced Research Projects Agency (DARPA), whose funding supported the work described in this thesis.¹

Finally, my wife, Hannah, deserves credit for keeping me on task and motivated. Without her support and prodding, this work may never have been completed.

¹ This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
ABSTRACT

With the meteoric rise in popularity of the Android platform, there is an urgent need to combat the accompanying proliferation of malware. Existing work addresses the area of consumer malware detection, but cannot detect novel, sophisticated, domain-specific malware that is targeted specifically at one aspect of an organization (e.g., ground operations of the US Military). Adversaries can exploit domain knowledge to camouflage malice within the legitimate behaviors of an app and behind a domain-specific trigger, rendering traditional approaches such as signature-matching, machine learning, and dynamic monitoring ineffective. Manual code inspections are also inadequate, scaling poorly and introducing human error. Yet, there is a dire need to detect this kind of malware before it causes catastrophic loss of life and property.

This dissertation presents the Security Toolbox, our novel solution for this challenging new problem posed by DARPA’s Automated Program Analysis for Cybersecurity (APAC) program. We employ a human-in-the-loop approach to amplify the natural intelligence of our analysts. Our automation detects interesting program behaviors and exposes them in an analysis Dashboard, allowing the analyst to brainstorm flaw hypotheses and ask new questions, which in turn can be answered by our automated analysis primitives. The Security Toolbox is built on top of Atlas, a novel program analysis platform made by EnSoft. Atlas uses a graph-based mathematical abstraction of software to produce a unified property multigraph, exposes a powerful API for writing analyzers using graph traversals, and provides both automated and interactive capabilities to facilitate program comprehension. The Security Toolbox is also powered by FlowMiner, a novel solution to mine fine-grained, compact data flow summaries of Java libraries. FlowMiner allows the Security Toolbox to complete a scalable and accurate partial program analysis of an application without including all of the libraries that it uses (e.g., Android).
This dissertation presents the Security Toolbox, Atlas, and FlowMiner. We provide empirical evidence of the effectiveness of the Security Toolbox for detecting novel, sophisticated, domain-specific Android malware, demonstrating that our approach outperforms other cutting-edge research tools and state-of-the-art commercial programs in both time and accuracy metrics. We also evaluate the effectiveness of Atlas as a program analysis platform and FlowMiner as a library summary tool.
CHAPTER 1. GENERAL INTRODUCTION

As use of the Android platform has exploded, so too has the prevalence of Android malware. The vast majority of Android malware is currently unsophisticated and makes little attempt to hide itself [132, 35, 153]. Many techniques and tools have been developed to detect this garden-variety consumer malware, including signature matching, machine learning, and dynamic behavioral monitoring. However, the US Department of Defense is worried about a very different scenario: novel, sophisticated, domain-specific malware created to damage a military or infrastructure target. We are already beginning to witness instances of this. In 2010, the Stuxnet worm was discovered and found to be specifically designed to cripple Iranian nuclear centrifuges [96]. Potential damage is not confined to hardware. For example, it is not difficult to imagine the loss of life that might result from a military tactical application that malfunctions when the user is within the GPS bounding box of Afghanistan.

Today’s malware detection techniques are woefully unequipped to handle this kind of novel, domain-specific, triggerable malware. For this reason, and to spur advancements in the field of static software analysis, DARPA created the Automated Program Analysis for Cybersecurity (APAC) program. This thesis describes my research contributions at our Knowledge-Centric Software Lab during Iowa State University’s participation from 2011 to 2015. ISU (with subcontractor EnSoft) was a “Blue” team performer\(^1\) tasked with constructing a malware detection tool. The program also included “Red” team performers\(^2\) whose task was to understand the research tools and construct challenging new malware that would be difficult to detect, and a “White” team performer\(^3\) to coordinate controlled malware detection experiments.

\(^1\) Other Blue team performers included MIT, Stanford, UC Berkeley, UC Santa Barbara, University of Utah, University of Washington, and BAE Systems \(^2\) Raytheon BBN and Raytheon Pikewerks \(^3\) Five Directions
Over the course of APAC Phase 1, our ISU/EnSoft team employed our knowledge-centric approach to software analysis to establish ourselves as the top-performing Blue team. We audited 77 challenge applications created by the Red teams, 62 of which contained embedded malware. We successfully pinpointed the malicious functionality in 57 of 62 malicious apps, achieving a detection rate of 93.5% and an average analysis time of only 1.13 hours per app. These accuracy and time metrics were best-in-class among APAC Blue teams, and significantly outperformed a control team using state-of-the-art commercial tooling. Moreover, we frequently discovered unintended vulnerabilities in the apps we were given. Our success on APAC is directly attributable to the identification of, and novel solutions for, the following research questions:

1. **How should a software analysis platform be built to facilitate both automation and human comprehension?** (Section 1.1, Chapter 3)

2. **How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?** (Section 1.2, Chapter 4)

3. **How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?** (Section 1.3, Chapter 5)

**Organization**

This thesis is organized as a collection of papers, each of which addresses one of the research questions presented above. Sections 1.1, 1.2, and 1.3 of this chapter provide general introductions, motivation, brief related work, and solution previews for each research topic. Chapter 2 provides a comprehensive literature review for all three research questions. Chapters 3, 4, and 5 are published and pending papers with supplementary appendices containing additional information, providing treatment of each research question in greater detail. Chapter 6 provides general concluding thoughts.
1.1 Software Analysis Platform for Automation and Comprehension

Novel, sophisticated, domain-specific malware cannot be detected in a fully-automated way. The same behavior may be benign for one application but malicious for another; for example, a mapping application that sends the device location over the Internet to download the correct map tiles would be benign, but an SMS messaging application that leaks this same information would be malicious. While automated tooling can expedite the process of detecting interesting program behaviors, it cannot make domain-specific judgment calls about the appropriateness of a behavior; for this task, a human being is needed. This observation led naturally to our first research question:

How should a software analysis platform be built to facilitate both automation and human comprehension?

Existing frameworks were insufficient for our purposes, providing either automation or static visualizations, but we required a flexible and interactive query-model-refine paradigm. To overcome the limitations of prior work and address this research question, we commissioned our subcontractor, EnSoft, to create **Atlas** [50]. **Atlas** employs a graph-based mathematical abstraction of software. It preprocesses the Abstract Syntax Tree (AST) of a program into a rich, attributed graph data structure in an in-memory graph database. This software graph can be queried in automated and interactive ways. Automation is supported through an embedded Java DSL, allowing automated analyzers to be written on top of **Atlas** using very few lines of code. Interaction and comprehension are supported in several ways. First, analysis results can be viewed using intuitive graph visualizations that have a one-to-one correspondence with the matching source or byte code. Second, **Atlas** provides an Interpreter View that allows the user to compute, query, and visualize results on-demand. Third, analyzers can be invoked automatically in response to user clicks through a configurable Smart View. For example, this view can be configured to instantly display a call graph, type hierarchy, or other artifact whenever the user clicks on a source token or graph element. This potent combination of automation and interaction has the effect of amplifying the intelligence of its users, enabling use cases that would be infeasible to automation or manual effort alone.
This research question and our solution (Atlas) are discussed in Chapter 3.

1.2 Man-Machine Detection of Malware

On its own, a software analysis platform that enables automation and interaction is not sufficient for malware detection— it is a foundation upon which a man-machine detection approach can be constructed. We recognized immediately that automated tooling can be used to point out interesting program behaviors, but a human analyst is required for making domain-specific judgment calls. The design of such a hybrid system necessitates answers to new questions such as (i) what behaviors are important to detect?, (ii) what behaviors can a static analysis feasibly detect?, (iii) how can we present behaviors to an analyst in a comprehensible way?, and (iv) how can we enable an analyst to effectively pose and answer follow-up questions? More generally:

How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?

Question iv is particularly crucial for addressing the shortcomings of traditional, existing two-pass defect detection tools. In a traditional two-pass tool, automation performs the first pass, then a human must manually confirm or reject its alarms. This places an unreasonable burden on the user. Today’s malware detection approaches either fall into the two-pass category, or else they are fully-automated and therefore not suitable for detecting novel, sophisticated, or domain-specific malware.

We used Atlas and its APIs to move beyond prior work and create the Security Toolbox. Unlike conventional two-pass approaches, the Security Toolbox uses an interactive approach. We detect malware using repeated iterations of automation and interaction; automation mines the artifacts to expose program behaviors, and the analyst synthesizes the results and formulates new questions for the automation to answer.

This research question and our solution (the Security Toolbox) are discussed in Chapter 4.
1.3 Library Summaries for Partial Program Analysis

Android applications, like most modern software, are built on top of reusable libraries. Android provides a massive library of functionality that applications can call that includes all of the standard Java library. In addition, the Android framework itself makes callbacks into an application in response to button clicks, interprocedural communication, component lifecycle changes, and many other events. Thus, analyzing an app by itself is a form of partial program analysis, defined as the analysis of a proper subset of a program’s implementation. Due to the sheer size of the Android framework (orders of magnitude larger than an app), including it in order to perform whole program analysis was infeasible. Yet failing to capture its behaviors, particularly information flows, resulted in incomplete results and missed detections from the APAC performers.

The APAC Blue teams tried radically-divergent approaches to solve this problem. Stanford hired a small army of graduate and undergraduate students to hand-write coarse information flow specifications for “important” Android APIs, then later worked to dynamically verify them [40]. This labor-intensive process produced succinct, but coarse, results of varying quality and coverage. At the other extreme, some performers attempted to include the entire Android framework into their analysis. This approach tackled the problems of quality and coverage, but introduced dire problems of computational scalability. Our ISU team felt that the best of both worlds could be captured by an automated, summary-based approach, if we could answer the following research question:

*How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?*

Most prior work on the topic of library summarization focused on strategies for call graph construction, and thus was unhelpful. While at least one other APAC performer, Stanford, attempted to summarize library data flows, their results were too coarse to be used accurately or capture flows involved in callbacks [40]. To aggregate the benefits of their work while avoiding the drawbacks, we designed FlowMiner, an automated tool for extracting fine-grained, compact data flow summaries of Java library bytecode. FlowMiner employs the graph-based
analysis paradigm and APIs of Atlas to perform a one-time static analysis of a Java library. It outputs *sound* data flow summaries as an abstract data flow graph, encoded using a portable XML format. This summary file can be reused by existing static analysis tools to achieve complete and accurate, yet scalable, partial program analysis.

This research question and our solution (*FlowMiner*) are discussed in Chapter 4.
CHAPTER 2. REVIEW OF LITERATURE

This chapter provides a review of literature for each research question.

2.1 Atlas

How should a software analysis platform be built to facilitate both automation and human comprehension?

Over the years, the field of software analysis has exploded into a complex web of loosely-related tools and techniques. One useful way to divide them is to sort them into static and dynamic categories. Static analyses reason about a program without running it, while dynamic techniques generally involve running it. Each has strengths and weaknesses. Static analyses excel at demonstrating hard-to-trigger program behaviors, but often produce false positive alarms. Dynamic analyses excel at demonstrating the presence of behaviors along common execution paths, but often struggle to cover them all, leading to false negatives. Use cases such as defect detection and program comprehension are topics that play to the strengths of static analysis; therefore, Atlas and its most closely-related work fall into the category of static analysis.

Atlas uses a graph-based mathematical abstraction to reason about software—this concept has a rich history. Ferrante et al. introduced the Program Dependence Graph\cite{ferrante} to explicitly capture both control and data dependencies of a program. Horwitz et al. proved the adequacy of program dependence graphs for representing programs. They showed that the expression of abstract syntax as a graph enables compilation, optimization, and even the checking of deep properties such as program equivalence \cite{horwitz}. This proof laid the groundwork for an explosion of subsequent related work, such as the Program Dependence Web\cite{weaver}, to represent programs as
graphs, leading to the modern Abstract Syntax Tree (AST) and eventually to approaches like Atlas.

2.1.1 Program Comprehension

One of the most natural applications of static analysis is program comprehension. Since the introduction of the PDG, researchers have imagined how program graphs could be used to support and empower developers. Ottenstein and Ottenstein imagined how a PDG might be used in an Interactive Development Environment (IDE) to support common development tasks, such as refactoring, debugging, and understanding program structure [109]. They saw visualization as a critical component of program comprehension. In 1983, Tischler proposed MAP, a static program analysis tool for understanding COBOL source programs. MAP includes elements from control flow analysis, data flow analysis, structure, and interactive visualization [136]. Unfortunately, as Frederick Brooks pointed out in 1987, software is very difficult to visualize because it has no inherent spacial representation [30]. The comprehension gap resulting from this difficulty is very real—Parnas and Lawford note that despite sincere efforts to ensure software quality, large software companies routinely release software containing errors. They assert that the problem stems not from lack of effort, but from the sheer complexity of modern software systems. They propose that frequent code inspections and program comprehension are critical to finding bugs prior to release [110].

Unified Modeling Language [124] (UML) diagrams were introduced as a visual way to communicate software requirements and design. Unfortunately, they have proven to be suboptimal for most use cases, including program comprehension [55]. Radfelder et al. pointed out the failings of UML, particularly for representing dynamic program behaviors. They suggested three-dimensional visualization as one way to solve the problem [118]. Kazman et al. presented Dali, a workbench to extract architectural information about an existing system. It extracts architectural patterns and visualizations to serve as ex-post-facto documentation [90]. Neginhal and Kothari proposed CVision, a tool for understanding C system software. CVision, a philosophical ancestor of Atlas, is an interactive tool allowing the user to visualize relationships between program elements at various levels of abstraction. Reduced views (abstractions) are
obtained by applying graph reductions to distill the code down to only relevant relationships. The system was evaluated and shown to be effective in aiding understanding of Xinu and Linux [106]. Recent work on software visualization includes SourceMiner[46], an Eclipse plug-in for software visualization. SourceMiner uses the AST as a base from which to create various structural views of software, as well as software metrics such as coupling and cohesion. It focuses exclusively on program structure and does not visualize runtime relationships; this limits its general usefulness and applicability. Werner et al. introduced EvolTrack[142], an Eclipse plug-in for visualizing the evolution of software. EvolTrack provides various meta-views of software, including mapping to and from UML diagrams, but is primarily intended for visualizing the transformation of software over time rather than deeply comprehending one software artifact. Femmer et al. proposed BusyBorg[66], an alternative to UML for visualizing program structure and runtime behaviors, particularly for embedded system software. It incorporates aspects of static program analysis, network monitoring, and profiling to show system-level interactions between executables and classes within them. ATLAS moves beyond this prior work, providing rich, interactive visualizations of both structure and runtime behaviors. Moreover, the visualizations that ATLAS produces have a one-to-one live correspondence with the backing software graph data structure; visual elements can be selected and directly used as inputs for analyzers.

Visualization is only one requisite component of program comprehension. Tilley and Paul described the features of an ideal reverse engineering tool for understanding programs. Such a system, they said, should create useful semantic abstractions, but should leave room for human insight and input [135]. Biggerstaff et al. presented the DESIRE platform based on the observation that we understand programs by associating implementation concepts with abstract domain concepts from our everyday lives. In order to understand unfamiliar software, they reasoned, we must be able to map implementation details to concepts in our daily lives, a notion they refer to as the “concept assignment problem.” They argued that effective program comprehension platforms must consider this problem by employing familiar concepts such as relationships, clusters, and slices [28]. Bennett and Ward described “middle out programming” for program comprehension. Reinforcing the view that domain concepts are critical for program comprehension, they argued that pure top-down or bottom-up ways of understanding software
are not how comprehension works in the real world [27]. Kothari and Deng also proposed the use of domain concepts to understand software. They described a user-customizable software engineering environment called SeeCORE for deriving a “program skeleton” upon which domain concepts can be applied [51].

In addition to domain concepts, Rajlich argued that they key to program comprehension is understanding and communicating the original developers’ intentions. Without intention, analysts can see what was done, but not necessarily why it was done. Despite the wide variety of program comprehension tools and techniques, he concluded that no one existing tool or technique can facilitate comprehension in a automated way. Instead we must choose trade-offs related to level of automation, level of detail, properties captured, and more [119]. Schots et al. agreed, noting that use of computing resources to aid in understanding software is still a challenge. They pointed out that program comprehension is critical for all parts of the development life cycle. Their survey of tools, methodologies, and current research in the area reinforced the view that no single existing tool seems to fulfill all comprehension needs. However, they did point out the important research trend of proper abstractions to capture details of interest while discarding others [129].

From this philosophical background, Atlas was born. In 2008, Kothari presented a Query-Model-Refine paradigm for understanding large software. His proposal was that a human analyst should use an analysis tool in iterations. Queries would return abstract software models which, upon consideration by the user, could be used to inspire more queries and refine results [91]. Indeed, this philosophy came to be the guiding principle for the construction and use of Atlas, and is the prime factor for its success as a powerful program comprehension tool.

2.1.2 Detecting Defects

Program comprehension is far from the only use for static analysis tools. Automation and the ability to reason about a program globally allow static analysis tools to excel at finding obscure and hard-to-trigger program defects. A huge number of defect-finding tools exist, falling into a number of categories. The first category is that of tools to aid with code inspections. Fagan argued that frequent code and design audits are effective tools to reduce bugs and increase
software quality [62]. Countless others have echoed this sentiment 25 years after Fagan’s initial work [17], and tool support for code inspection has increased dramatically [10]. Indeed, Atlas itself was shaped by the use case of interactive code inspection for security audits.

Another massive category of static defect detection tools is that of automated bug-finding. Perhaps the single best-known bug finding tool is FindBugs, created by Hovemeyer and Pugh. FindBugs works by defining and detecting common idioms and code patterns that often indicate bugs. These patterns, even when extremely simple, tend to find real bugs in production software [82, 19]. Ayewah et al. provided an evaluation of FindBugs for defect detection on production software. They concluded that many of the detected bugs are real, though often trivial, and that many companies do not have systematic strategies for reviewing warnings from tools. They also noted that there is a long tail of bug detection patterns that rarely, if ever, find problems [20]. Others have criticized FindBugs for the number of false alarms it gives. Vetro et al. performed an empirical study of the precision of FindBugs for a body of software developed at a university. They found that very few of the issues that FindBugs reports have high precision—that is, the majority of warnings that FindBugs raises are false positives, and therefore are not particularly useful for ensuring software quality [140]. Among other tools, Bush et al. proposed a static analyzer, PREfix, for finding dynamic programming errors. They noted that 90% of C and C++ bugs involve interactions between multiple functions, so local warnings within functions are not sufficient to find them. PREfix uses a bottom-up approach to create a summary model of the effects of each function. It then performs interprocedural analysis by composing the effects of models along calls [32]. Hovemeyer et al. employed static analysis to detect null pointer dereference bugs in Java programs. They perform a forward intra-procedural data flow analysis from null literals, categorizing variable definitions as definitely null, definitely not null, or possibly null. If a dereference occurs upon a variable that may be null, an alarm is raised. Subsequently, a path feasibility analysis discards some warnings that are likely to be false positives, though this introduces some false negatives [83]. Pienaar and Hundt created JSWhiz, a static analysis tool for detecting memory leaks in JavaScript. They identify five common code patterns leading to memory leaks. JSWhiz extends the Closure JavaScript compiler, using its back end to identify and warn when such patterns are found [114]. Wu et al. employ static
analysis to check exception handling in Java programs [144]. Many, many other bug-finding static analysis tools have been developed, and are beyond the scope of this thesis to list.

Innocent bugs become vulnerabilities when they lead to security problems. Sotinov presented static analysis techniques to detect vulnerabilities in C programs. He identified several code patterns characteristic of vulnerabilities, then employed taint analysis and value range propagation to detect execution paths exhibiting these patterns. The approach is implemented as an extension to the GNU C compiler [131]. Austin et al. performed a comparison survey of vulnerability discovery techniques, including exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis. Each technique had its own strengths and weaknesses. Systematic manual penetration testing found the most design flaws, while static analysis found the most implementation bugs [18]. When comparing penetration tools against static analysis tools for detecting SQL injection vulnerabilities, Antunes and Vieira found that static analysis tools offer better coverage, though both approaches suffer from problems of false positive alarms [11]. Kannavara argued that static analysis tools should be incorporated in early stages of the software development process, immediately after new modules have been implemented. He recommended that client companies run static analysis tools on open source projects before using them in products [89]. McLean compared a number of static analysis tools for detecting vulnerabilities on a body of open-source software. He found that different static analysis tools are useful for detecting different kinds of vulnerabilities. He also found that tools tend to raise a large number of warnings that are false positives [103]. Perhaps most philosophically close to Atlas, Yamaguchi et al. proposed the use of Code Property Graphs [146] for modeling and detecting vulnerabilities using traversals on a unified graph data structure. The CPG combines concepts such as control flow graphs, abstract syntax trees, and program dependence graphs into a single data structure that can be queried. Using a CPG, vulnerabilities can be expressed concisely using traversal templates.

In a review of several bug-finding tools, including Bandera, ESC/Java 2, FindBugs, JLint, and PMD, Rutar et al. found that the tools find different subsets of software bugs, and suggested using them together in a “meta-tool” that provides better accuracy by combining warnings [125]. Meng et al. extended this work by proposing a way to merge the results of separate static
analysis tools. They proposed uniform specifications of defect patterns, as well as policies for prioritizing results [104]. On balance, Johnson and Bowdidge noted that static analysis tools are still woefully underutilized when it comes to finding bugs. They found that developers encounter too many false positive warnings, most of which are difficult to comprehend. In addition, most static analysis tools do not include interactive mechanisms to help developers address their warnings [87]. Heckman and Williams also noted that a large portion of static analysis warnings are false positives, with only a small portion being actionable. They proposed the use of machine learning techniques to classify alerts as actionable or not actionable. By selecting 51 characteristics of warnings, their machine learning algorithms classified 88-97% of alerts correctly. They noted that the algorithms and characteristics that worked best differed by project, so they suggested that alert classification should be tuned by specific project properties [77].

Kothari et al. worked to address this problem of false alarms by proposing a Knowledge Centric Software framework consisting of an eXtensible Common Intermediate Language (XCIL) and an eXtensible Pattern Specification Language (XPSL) for representing domain-specific knowledge and detecting safety property violations in a language-independent way. They proposed bug detection through inspection and comprehension, in which an analyst could encode domain-specific patterns that indicate property violations based on the results of exploration. The automation, then, could be used to discover the existence of these patterns in safety-critical code [92]. As a test of this philosophy, Gui and Kothari examined matching pair properties, wherein an event A should be followed by an event B on every execution path. If there is an execution path on which A is not followed by B, this constitutes a bug. Examples include synchronization and memory allocation and deallocation. They provided an empirical study of the Linux kernel and used program comprehension tooling to experimentally find the programming patterns along which matching pair properties occur. Based on this survey, they created the concept of a Matching Pair Graph, MPG(X), which constitutes the minimum-sized call graph that must be considered in order to verify a matching pair property for X [76]. This philosophy of combining user insight with automation to reduce false alarms lives on with ATLAS today. ATLAS is not itself a bug finding tool; however, it can be and has been used as a base for writing
2.1.3 Designing Test Cases

A widely-recognized challenge in testing is the problem of designing good test cases. Static analysis tools can be used for this purpose. Laski and Korel suggested employing data-flow oriented static program analysis as a means to create test cases. Their first strategy was to examine uses of variables and check the liveness of every definition at these points. A second strategy tests vectors of variables at once, testing the liveness of a permutation of definitions at points of use. They concluded that good test cases can automatically be inferred from these data flow analyses [98]. Bates and Horwitz proposed the use of program dependence graphs and program slices to incrementally test programs. They suggested that, starting from points at which a program has been modified, static program slices can be used to evaluate the impact of changes. This impact directly corresponds to existing tests that need to be updated, and new tests that should be written [24].

A more recent proposal of program analysis for test generation came from Godefried et al. They observed that despite their popularity, static analysis tools are not yet widely used for automatic test generation. They described three tools at Microsoft that are moving towards this goal, all of which incorporate static analysis techniques like symbolic execution, model checking, and constraint solving. One tool, SAGE, generates white-box fuzz testing inputs for finding security problems. Another tool, PEX, synthesizes automatic unit tests for .NET. The third tool, Yogi, combines testing and static analysis to simultaneously work on proofs and counterexamples for whether a desired program property holds [71]. We have not employed Atlas in the area of automatic test case generation, although we have every expectation that the necessary raw materials for doing so are captured.

2.1.4 Analysis as a Platform

At its core, Atlas is a static program analysis platform, providing a software graph database and a powerful query API for allowing others to write analyses. Many other program analysis
platforms have been created. One well-known platform is Soot\cite{95}, a compilation toolchain for the Jimple intermediate representation language. Many bug-finding tools and optimizers have been written on top of the Soot tool chain. Lam et al. created a static analysis database for context-sensitive pointer analysis called bddbddd. It uses a Datalog embedded query language called PQL. The authors spent a man-year optimizing the implementation of bddbddd to scale to large software \cite{94}. Another approach to building static analysis platforms involves encoding program properties as boolean satisfiability (SAT) problem instances and solving them using SAT-solvers. Frameworks that take this approach include CALYSTO\cite{21} and SATURN\cite{145}. Atlas differs from all of these platforms in several ways. First, it reasons about software using a graph-based mathematical abstraction, relying upon reachability properties rather than boolean formulations. Second, it places heavy emphasis on user interaction, visualization, and program comprehension, affording an unprecedented level of usability. Third, Atlas preprocesses the AST, allowing analyzers to be written at a much higher level of abstraction than is typical on other platforms. This allows for incredibly powerful analyses to be written using very few lines of code. For example, after deciding that we needed a symbolic value analysis for our participation on APAC, I created a complete tool in one day using fewer than 4kLOC.

\section{2.2 ISU Security Toolbox}

How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?

\subsection{2.2.1 Nature of Android Malware}

Malware detection for Android applications has been an \textit{extremely} hot topic from 2011 to the present day– this stems from the open nature of the Android platform, its skyrocketing popularity, and the accompanying proliferation of malware. Spreitzenbarth noted that the prevalence of Android malware increased 3000\% in the second half of 2011 alone, and that the vast majority of malware made no attempt to hide or obfuscate itself. He observed that malware was typically profit-driven and simplistic, but was quickly becoming more complex as detection tools improved \cite{132}. A 2011 survey by Castillo et al. at McAfee also observed that
while malware was spreading quickly, it was unsophisticated and made few attempts to evade detection. As detection tools improved, they noted that malware increasingly employed obfuscation, dynamic downloading of payloads, on-the-fly code encryption/decryption, and other well-known techniques from the realm of PC malware. [35]

Unfortunately, the standard signature-based approach to malware from the PC world is not working well. Zhou and Jiang provided a survey of the behaviors of 1,200 malware samples, finding that state-of-the-art commercial anti-virus products detected only 20% to 80% of malware samples. This low detection rate came despite the low sophistication and simple profit-driven motives of Android malware. They too concluded that as malware became increasingly sophisticated, these detection numbers would get worse [153].

Suarez-Tangil et al. recently surveyed the 20 most well-regarded analysis tools, providing a picture of how malware has evolved over several years to evade them. They noted that malware is still profit-driven, but is becoming increasingly flexible and adaptive to evade detection by even state-of-the-art tools [134]. Indeed, researchers have demonstrated that it is still easy to exfiltrate sensitive device data and bypass today’s state-of-the-art tooling [54, 93], and that adversaries have an enormous number of attack vectors at their disposal. Aprille and Strazzere posited that rather than trying to detect malware, end-users might instead evaluate apps using a risk-based statistical scoring system based on API methods called, permissions requested, size of code, contained URLs, and other features [12]. However, many users may not have sufficient risk appetite to take a probabilistic approach to their security.

This is particularly true in a DoD scenario; any non-zero probability of malware in a deployed app could destroy lives and property. Insufficiency of existing tooling to handle these malware trends was a primary motivation of the APAC program, and is the inspiration for our Security Toolbox. Rather than relying upon the failed signature and heuristic-based approaches of the past, the Security Toolbox puts a human-in-the-loop, amplifying his or her natural intelligence.
2.2.2 Machine Learning

When faced with a problem whose solution is difficult to codify, practitioners are inevitably tempted to apply machine learning techniques to develop a classifier. So it was with the detection of Android malware. A number of researchers proposed the use of static feature vectors, such as the size of an application, permissions requested, permissions used, metadata, source or byte code strings, and control flow graph (CFG) features [126, 149, 13, 152, 1, 84, 16, 120, 128]. Others suggested the use of dynamic behaviors for feature vectors, including API calls, fired Intents, and other system events [130, 53]. The effectiveness of the approaches varies. Feizollah et al. surveyed 100 recent machine-learning techniques used to detect Android malware. They found that feature vectors include static, dynamic, metadata, and hybrid features, with static and dynamic features tied for popular use. They concluded that most work did not use data sets of sufficient size for drawing statistically-valid conclusions. [64]

Machine learning techniques for malware detection are not suitable when the threat model assumes that malware will be novel, sophisticated, and very dissimilar from simplistic commercial malware. This was our situation during the APAC program; it was assumed that malware created by a nation-state adversary would be novel, domain-specific, and very difficult to discern from intended application behaviors. Unlike the prior work using machine learning techniques, the Security Toolbox was designed specifically for this scenario.

2.2.3 Dynamic Testing and Sandboxing

Researchers have proposed various categories of testing and emulation for exposing malicious behaviors of Android apps. Burguera et al. created CrowDroid, a crowd-sourced dynamic behavior analysis tool. CrowDroid collects application event traces on many devices and submits them to a central location, where traces can be compared against those from known malware samples [31]. Reina et al. introduced CopperDroid, a dynamic behavior analysis tool. CopperDroid logs and reports application system calls for subsequent post-processing by a user or another tool [121]. Alazab et al. suggested dynamic execution of Android applications within sandboxes to discover malicious behaviors [5]. Weichselbaum et al. created Andrubis,
a fully-automated emulation and test framework that automatically generates test cases using
the results of a static analysis [141]. Enck et al. proposed TaintDroid, a system-wide dynamic
taint tracking system. TaintDroid monitors flows of sensitive information in real-time, reporting
leaks immediately when they are observed [60]. Unfortunately, others have found that detection
by TaintDroid can be easily avoided by a handful of well-known techniques, including implicit
information flow and use of covert channels [127]. Finally, Yan and Yin created DroidScope, a
profiling emulation environment for dynamically reverse-engineering previously-identified mal-
ware [148].

There are several problems with dynamic testing approaches for malware detection. First,
they tend to identify a malicious application only after it has done something malicious. In
security-critical scenarios, this reactive model is unacceptable. Second, it is not difficult for
malware to detect that it is being run in an emulation environment and modify its behavior
to avoid detection. Third, the novel and sophisticated malware considered by DARPA for
its APAC program is *triggerable* by an adversary. As any software test engineer knows, it is
infeasible to cover all possible execution paths and inputs. Therefore, the malware that the
DoD worries about and that the Security Toolbox is designed to detect is very likely to
pass through dynamic testing approaches undetected.

2.2.4 Changes to the Android Platform

A number of security improvements to the Android platform itself have been proposed. Enck
et al. proposed Kirin, a security service for Android that performs install-time certification
against a set of security policies [61]. Poeplau et al. investigated the dynamic code loading
features of the Android platform. They note that up to 16% of the top free applications
dynamically download and load code, suggesting that Android could be made more secure if
the framework included integrity checks for loaded code [116]. Jeon et al. suggested that
Android could be a much safer platform if finer-grained permissions were introduced. They
demonstrated a prototype fine-grained permission model, showing that application permissions
could be auto-detected without changes to functionality [86]. Grace et al. created Woodpecker,
a tool for checking the enforcement of permissions in Android device images. They found that
vendors frequently fail to correctly enforce permissions in system images, allowing permissions
to be “leaked” to applications that should not have them [73]. Au et al. created PScout, a
tool for automatic static extraction of the Android permission model. They noted that between
18% and 26% of permissions could be entirely eliminated if applications were constrained to
call only publically-documented APIs [15].

It is true that many security features of Android could be improved, and that the mapping
of permissions required to use each API method is poorly-documented. In fact, the Security Toolbox
directly leverages permission mappings produced by PScout to overcome this problem. However, an effective malware detection tool should be able to address Android as it exists today. Even if future improvements were introduced by Google, legacy versions of Android
would still be used for years to come. Thus, the Security Toolbox makes no presumptions
of improvements or changes to Android.

2.2.5 Two-Pass Analysis Systems

Our most closely-related work, and the current state-of-the-art, is a group of traditional
two-pass static analysis tools. In the first pass, the tools run in a fully-automated way to gather
evidence. In the second pass, a human must manually review the reports. The vast majority of
static analysis tooling, including malware detection tooling, runs in this manner. Some of the
most widely-regarded two-pass analysis tools for Android vulnerability and malware detection
include, in roughly chronological order:

- SCanDroid, a static analyzer that infers allowed information flows from the application
  manifest, comparing these against statically-computed flows [70]

- Androlyzer, a static report generator for Android APKs that reports APIs, libraries,
  permissions used, and app structure [26]

- ComDroid, a static analyzer for inter-process communication (Intent) vulnerabilities [39]

- Stowaway, a static analyzer to detect requested, but unnecessary, permissions [65]
• CHEX, a static analyzer for detecting component hijacking vulnerabilities, Intents that are assumed to be internal but can be intercepted by external apps [101]

• SmartDroid, a hybrid static/dynamic analysis tool for identifying UI actions that are likely triggers for unusual behaviors [151]

• RiskRanker, a static heuristic-based approach for predicting that an app is malicious [72]

• SAAF, a static analysis framework for identifying suspicious flows with backward program slices from parameters of sensitive methods [78]

• Anadroid, an abstract interpretation approach to malware detection employing techniques to consider all possible event sequences [100]

• AsDroid, a static analyzer that compares UI text labels against actions that are taken during UI callbacks [85]

• Dendroid, a static analyzer using binary edit distance metrics to compute clusters and form a “phylogenic” tree of malware by its similarity [133]

• FlowDroid, the highly-regarded static taint analyzer extending the IDFS algorithm with a lifecycle model for Android apps [14, 69]

While each of these two-pass solutions brings novel contributions to the field, they all share a common weakness— the human analyst is only involved after the automation has finished producing its results. This means the analyst is left to sift through reported problems, which often appear in cryptic formats and with little helpful evidence to justify the report, without tool assistance. In other words, the task of rejecting false positives becomes a time-consuming, manual effort.

By contrast, the Security Toolbox was designed from the ground up to be a human-in-the-loop malware detection system. The analyst and automation work together to synthesize results; analyst questions can be answered by artifacts produced by the automation, which then inspire further questions. This human-in-the-loop approach allows the Security Toolbox to be much better at detecting novel, domain-specific, sophisticated malware that would be missed
by other tools. It also allows our analysts to avoid a burdensome post-automation manual review phase. We credit our out-performance versus other APAC teams to this important distinction.

2.3 FlowMiner

*How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?*

### 2.3.1 Defining Partial Program Analysis

The term *partial program analysis* can be defined in at least two ways. Degenais defined the concept to mean the analysis of a subset of a compiling program P. In this definition, the subset itself need not compile. He described the problem of type inference in partial Java programs, wherein a referenced artifact (a class, field, etc) is simply missing [44]. Degenais et al. then proposed a framework to accomplish this type inference, partially recovering declared types of expressions [45]. Similarly, Balatsouras and Smaragdakis developed a tool called JPhantom to handle references to missing types from libraries using “class hierarchy complementation”. JPhantom generates stand-in superclasses so that type hierarchies can become consistent for other analysis systems, such as Soot [23]. Madsen et al. proposed the analysis of JavaScript libraries with a focus on how a library is used. They observed that the types of values passed as parameters to a library call can be useful for inferring the type of library method return values and application callback parameters [102].

This is different than the standard definition of partial program analysis that we, and most others, use. In our definition, the subset of P to be analyzed *does compile* (all referenced artifacts are defined), but the implementation of libraries is not given. That is, an application is built and analyzed against a set of library *stubs* that express only the signature of the library APIs. This definition and scenario addressed by FlowMiner are much more common in practice than the analysis of a non-compiling program described by Degenais.
2.3.2 Control Flow

The majority of the work on partial program analysis has focused on call and control flow graph construction. Early efforts included Class Hierarchy Analysis[47] and Rapid Type Analysis[22], proposed as structural type hierarchy-based techniques for inferring possible runtime types at dynamic dispatch call sites. These techniques do not summarize control flows in a library, but rather provide a way to conservatively over-approximate for dispatch calculations. Grove and Chambers noted that an interprocedural call graph is a key prerequisite for other kinds of analysis, proposing a framework for call graph construction algorithms [74]. Ali and Lhotak proposed in CGC that all possible library targets be summarized by a single node in the call graph [7]. They later created Averroes, a tool for generating library placeholders for program analysis. Averroes leverages the assumption that libraries must compile in a context that does not include a client application. The placeholders over-approximate all possible library behaviors [8, 6]. Unfortunately, because they massively over-approximate library behaviors, use of CGC and Averroes is extremely inaccurate and produces a huge number of false positive analysis results.

In more closely-related work, Probst noted the problems posed by dynamic dispatches in object oriented languages and of partial program analysis. He proposed a one-time library analysis that uses a type constraint graph for type inference, allowing an interprocedural control flow graph for a library to be constructed [117]. FlowMiner also performs a one-time analysis of a library, but the target of our analysis is the extraction of data flow features rather than control flow features.

Yan et al. proposed summary-based whole-program analysis, wherein library implementation can be replaced with automatically-computed summaries that are independent of subsequent analysis use cases—this is the philosophy of FlowMiner as well. However, they focused on the mechanics of incorporating summary generation support into existing frameworks, particularly Soot, while FlowMiner addresses the algorithms and graph schema used to express summaries, as well as the mechanics of one tool (Atlas) [147].
Cao et al. recently created EdgeMiner, a tool for automatically mining callback/registration pairs from a Java library. “Registration” methods are API methods with which an application can provide an object reference to a library, while callbacks are library call sites from which a polymorphic call can be made on a registered object. EdgeMiner computes a configuration file of possible registration/callback pairs, allowing a partial program analysis tool to capture possible callbacks in its call graph [34]. This information is implicit in the data flow summaries generated by FlowMiner; any call site that could possibly have more than one runtime target (eg. callbacks) is preserved to be resolved at the time of summary application. In addition, we capture the flow of object references to call sites, allowing possible “registration” to be discovered. Hence, EdgeMiner could have been directly implemented as an analysis use case on top of the summaries generated by FlowMiner.

2.3.3 Data Flow

There is very little closely-related work for summarizing data flow properties of libraries. For one partial program analysis use case, Chapman et al. and Amey et al. proposed annotating library API methods with their trusted security levels. If a partial program flow reaches an API whose security level is untrusted, a problem can be reported without the implementation details of flows within the library [36, 9]. Unlike FlowMiner, this approach is manually-cumbersome and applies only to one security use case.

Rountev et al. provided a theoretical framework for Component-Level Data-flow Analysis, a one-time analysis of library source to create a condensed Inter-Procedural Control Flow Graph (ICFG). The ICFG consists of function-level data flow summaries connected by interprocedural calls [122]. Like FlowMiner, the authors recognized the importance of capturing callbacks in summaries. Unlike FlowMiner, they did not provide a concrete implementation, and did not handle class member fields in full generality.

Our most closely-related work was done by Clapp et al., who presented a technique for mining coarse information flow specifications from concrete executions of a Java library. Their approach instruments a library and uses a test driver to make API calls and generate flow traces.
These traces are post-processed into coarse information flow specifications, which are expressed as annotations on library methods. [40]

There are several issues with this work. The first drawback stems from the authors’ use of dynamic analysis to identify information flow relationships between the input and output variables of a method; this inherently makes it infeasible to cover all possible paths in the library, unavoidably introducing false negatives. Second, the information flow specifications produced are too coarse to be used accurately; information flow is tracked at the level of granularity of objects rather than the granularity of fields and other variables. This causes unrelated flows to become conflated. For example, a method parameter which is written to a field of an object is considered to "taint" the entire object. Therefore, flows through unrelated fields are joined to the same taint, producing false positives.

Third, their information flow specifications express flows only between elements of the library’s public surface (API), throwing away all other flows through intermediate variables. This limits the ability of a subsequent static analysis to use the mined specifications in an accurate way. For example, a static analyzer might have used the details of calling contexts and object types to restrict a library flow to valid constraints, but these details are not present in the flow specifications. Fourth, their flow specifications fail to capture potential flows from a library back into an application via polymorphic call sites. For example, an application can define a subtype of a library type, providing overriding method definitions. At runtime, virtual call sites in the library may result in calls and flows back into the application.

FlowMiner addresses all four issues. In contrast to the work by Clapp et al., we use static analysis (built atop the Atlas platform) instead of dynamic analysis to identify possible flows within the library. Hence, we avoid the possibility that some execution paths are not covered. Next, the flow specifications extracted by FlowMiner track and preserve data flows at the granularity of individual variables and definitions within methods and objects, so we avoid falsely merging unrelated flows. Furthermore, our flow specifications express flows among program elements that are not necessarily on the library API. This allows subsequent analyses to be context, field, type, object, and flow-sensitive. Finally, we retain the details of virtual call sites so that flows involving potential callbacks into an application are captured.
CHAPTER 3. ATLAS: A NEW WAY TO EXPLORE SOFTWARE, BUILD ANALYSIS TOOLS

This paper was presented at ICSE in June of 2014 in Hyderabad, India and published in the Companion Proceedings of the 36th International Conference on Software Engineering [50]. Material in Section 3.7 has been added for this thesis and does not appear in the original paper. This work addresses our first research question: How should a software analysis platform be built to facilitate both automation and human comprehension?

Tom Deering123, Suresh Kothari3, Jeremias Sauced4, Jon Mathews4

Abstract

Atlas is a new software analysis platform from EnSoft Corp. Atlas decouples the domain-specific analysis goal from its underlying mechanism by splitting analysis into two distinct phases. In the first phase, polynomial-time static analyzers index the software AST, building a rich graph database. In the second phase, users can explore the graph directly or run custom analysis scripts written using a convenient API. These features make Atlas ideal for both interaction and automation. In this paper, we describe the motivation, design, and use of Atlas. We present validation case studies, including the verification of safe synchronization of the Linux kernel, and the detection of malware in Android applications. Our ICSE 2014 demo explores the comprehension and malware detection use cases.

Video: http://youtu.be/cZOWlJ-IO0k

1 Primary author 2 Author for correspondence 3 Graduate student and advisor, respectively. Department of Electrical and Computer Engineering, Iowa State University. 4 Primary researcher, EnSoft Corp.
3.1 Motivation

Today’s software is growing larger and more complex at an alarming rate, but our cognitive abilities as practitioners are fixed. We increasingly rely on tooling to break through the comprehension threshold shown in Figure 3.1. Many tools remain isolated to academia [42], but Atlas aims to bring practical software analysis to the masses.

![Figure 3.1: Atlas helps comprehend complex software.](image)

Static analysis tools traditionally suffer from false positives due to over-approximation of program behaviors. Prior work seeks to address this with increasingly sensitive and expensive analyses. A common technique is to express analyses properties in terms of satisfiability (SAT), where basic blocks have been labeled with boolean reachability predicates, relying upon advancements in SAT solver scalability. Recent tools pursuing this approach include SATURN[3] and CALYSTO[21]. Reduction techniques such as Binary Decision Diagrams (BDD)[4] and Event Flow Graphs (EFG)[2] may allow the sizes of the necessary control flow graphs and boolean formulations to be greatly reduced; however, not every analysis query can be easily formulated as a SAT problem, and fully-sensitive analyses remain prohibitively expensive.

While it is possible to script a traditional static analysis using Atlas, we propose an alternative. Atlas breaks analysis into two phases. First, a set of conservative, polynomial time static analyzers translate the software AST into a graph database of precomputed artifacts and relationships. Then, a user can directly query and interact with that database to quickly discharge
false positives and iteratively refine the results (see Section 3.3). This approach allows an analyst to supply critical invariants, insights, and software design knowledge which would otherwise be unavailable to a fully-automated approach. Software mining has been explored in the past in work such as CIA [38], GENOA [52], SCRUPLE [111], SCA [112], Software Bookshelf [58], GUPRO [57], Metanool [29], and GReQL [56]. However, to our knowledge, none of these tools offer the same ease of use or blend of automation and interactivity. In this way, Atlas seeks to be an intelligence amplifying system as proposed by Fred Brooks: “If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.” [88]

### 3.2 Audience & Use Cases

![Atlas graph showing flow of data to and from the DIGEST_ALGORITHM field.](image)

Atlas is useful for nearly any analysis task, but its fundamental use case is code comprehension. The precomputed relationships include sufficient material for building call graphs, data flow graphs, type hierarchies, dependency graphs, and many other useful results. A number of out-of-the-box scripts are provided for common queries. These lightweight analyses are invoked automatically in Eclipse Smart Views as the user clicks on code artifacts, providing instant feedback and interactive software graph visualizations. Figure 3.2 shows an example data flow...
graph produced when the user clicks on the field `DIGEST_ALGORITHM`. It shows the field definition, plus that part of the data flow graph which is reachable via forward and reverse traversals on data flow relationships. The declaring control flow, method, class, package and project artifacts are shown automatically to provide visual context. In addition to Smart Views, Atlas provides an Interpreter View which allows the user to make on-the-fly queries using a provided API. These code comprehension features of Atlas are ideal for development, code reviews, bug reports, software audits, documentation, managerial review, and much more.

In addition to its out-of-the-box functionality, Atlas provides a rich API for writing custom analyzers. The library provides capabilities for selecting, traversing, and showing subgraphs from the larger software graph. Figure 3.3 shows a short Atlas script which would produce the data flow graph shown in Figure 3.2. The scope of possible analysis use cases is bounded only by the creativity of the user. For example, a script which performs global type inference, re-resolves dynamic dispatches, and modifies the graph database to reflect the results can be written with only a few hundred lines of Atlas code! We describe two real-world analysis applications, verifying safe synchronization in Linux and detecting malware in Android applications, in Sections 3.5 and 3.4, respectively.

```java
Q field = fields("DIGEST_ALGORITHM");
Q df = edges(Edge.DATA_FLOW);
Q result = df.forward(field).union(df.reverse(field));
show(result);
```

Figure 3.3: An example Atlas script which would produce the graph shown in Figure 3.2.

### 3.3 Design & Architecture

Atlas is available today as a plugin for the popular Eclipse IDE. It is architected to achieve several design goals:

1. Provide sufficient material to solve difficult analysis problems.
2. Provide lightweight built-in analyses which scale to millions of lines of code.
3. Enable both automation and interaction.
To provide sufficient analysis materials, Atlas employs a number of polynomial time static analyzers to index an attributed, directed graph representation of the program’s Abstract Syntax Tree. The nodes in the graph are software artifacts, and the edges are relationships between the artifacts. Each node or edge contains a set of tags and a map of attributes which contain additional information about that element (eg the local alias of the object instance in use). Sufficient raw material is provided to solve arbitrary analysis problems. Table 3.1 shows a high-level view of the artifact types and relationships that Atlas currently indexes.

Table 3.1: Artifacts and relationships in Atlas graphs.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>PROJECT, PACKAGE, CLASS, INTERFACE, ENUM, ANNOTATION, PARAMETER, FIELD, ENUMCONSTANT, DATA_FLOW, METHOD, CONTROL_FLOW, INVOKE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>DECLARES, ANNOTATION, ELEMENTTYPE, OVERRIDES, PARAM, RETURNS, SUPERTYPE, THROWS_CLAUSE, TYPEOF, CALL, INVOKE, CAST, CATCH, CATCH, THROW, READ, WRITE</td>
</tr>
</tbody>
</table>

To achieve scalability, Atlas indexes all of the necessary raw material, but only precomputes a first approximation of relationships. For example, when encountering a method invocation representing a dynamic dispatch, Atlas conservatively indexes CALL edges to all possible method targets. This first approximation allows Atlas to avoid doing the heavy lifting necessary for global type inference, keeps the indexing process fast, and may be good enough to satisfy most use cases. However, if the user really does want to do global type inference to tighten this approximation, he or she can do so by writing an Atlas script. This philosophy allows Atlas to index millions of lines of code in minutes and avoid unnecessary work.

Automation and interaction are provided via multiple Eclipse views. Atlas provides a “Smart View”, which allows for immediate analysis results to be shown to the user in response to clicks on software elements. Atlas also provides an “Interpreter View”, which allows the user to compose on-the-fly queries in a Scala interpreter environment and show results in an Eclipse graph editor. In both cases, the graph layout is customizable, and clicking on graph elements brings the user directly to the corresponding code locations. The Interpreter also allows the user to invoke automated scripts from a special project in the workspace, called a “toolbox”. In Sections 3.4 and 3.5, we describe the use of toolbox projects to perform custom analyses.
3.4 Malware Detection Study

Analysis problems which require high-specific domain knowledge are a natural reason to extend the capabilities of Atlas. Consider the problem of detecting novel malicious behavior in Android applications, as in DARPA’s Automated Program Analysis for Cybersecurity (APAC) program. Iowa State University is a performer on the project, using Atlas as the foundation of its approach. In order to detect malice, we must first define what it means for a behavior to be “malicious”. Unfortunately, this depends on the purpose of the app. For example, it is expected for a navigation app to send the device location to the Internet, but that is unexpected for a podcast player.

Human judgment is required to determine the legitimacy of a behavior, so fully automated detection of malware is not possible. Instead, we seek to amplify the natural intelligence and intuitions of the human analyst by providing the “ISU Security Toolbox”, an analysis suite built on top of Atlas. The Toolbox contains Atlas scripts which detect “smelly” software patterns, such as reflectively calling private library methods and dynamic code loading. We also provide parameterized detection scripts which locate violations of the well-known CIA (Confidentiality, Integrity, Availability) security model. The analyst pre-encodes his domain knowledge of what is security-relevant for a particular application, and the Toolbox runs the selected scripts and parameters in an automated way. After the results are aggregated, the analyst systematically reviews them. An overview of the approach is shown in Figure 3.4.

In the ISU approach, the human provides the creativity and insight, while Atlas performs the mechanical burden of finding the requested behaviors. As with synchronization verification, Atlas frees the ISU analyst to focus on the problem domain. Meanwhile, the analyst provides key insights and domain knowledge that Atlas, by itself, is lacking. In the first four APAC experiments, we analyzed a mix of 76 malicious and benign challenge applications provided by the adversarial challenge performers. To date, we correctly classified 90% of the apps, with an average analysis time of 2 hours. These results exceed those of any other APAC performer.
3.5 Linux Synchronization Study

In Section 3.4 we described the semi-automated task of detecting novel malware, which begins by understanding an app’s purpose. Consider a more concrete problem which lends itself more naturally to automation. Many important software properties can be modeled as two-event problems, wherein we wish to verify that event $B$ follows event $A$ on all possible execution paths. Examples include allocation and deallocation for memory management and locking and unlocking for safe synchronization. In our group’s prior work, we employ a version of Atlas for the C programming language to verify the latter property in the Linux 2.6.31 kernel, where the two events correspond to calls to `mutex_lock()` and `mutex_unlock()`. Taking full advantage of the Atlas man + machine philosophy, we successfully demonstrate the correctness of synchronization in Linux. [75]

Let us refer to a locking event on mutex signature $X$ as $L(X)$, and the unlocking event as $U(X)$. To show that $U(X)$ follows $L(X)$ on every execution path, we must demonstrate that all

Figure 3.4: ISU detects novel malware using Atlas.
paths along which a violation may occur are infeasible. At first, the problem appears to be intractable due to the exponential number of execution paths. However, we observe that the number of ways in which $U(X)$ may follow $L(X)$ is limited:

1. $U(X)$ follows $L(X)$ within the same function.

2. Token $X$ is passed on the stack as a parameter to the forward call graph, which performs $U(X)$.

3. Token $X$ is returned on the stack to the reverse call graph, which performs $U(X)$.

4. Token $X$ is written to a global variable and is later read to perform $U(X)$.

The first scenario is trivial to check, the second and third are a bit more complex, and the fourth is the worst case. If Linux is well-designed, we hypothesize that scenarios 1-3 are the prevailing pattern. Our analysis begins by using Atlas to locate all $L(X)$ and $U(X)$ events in the kernel. We utilize a custom Atlas script to automatically discharge the simple scenarios from category 1. Next, we develop the notion of a Matching Pair Graph (MPG($X$)). MPG($X$) encapsulates categories 2-4 to provide the minimal subset of the kernel’s call graph which must be considered to verify a locking scenario. We compute MPG($X$) for all $X$ automatically using an Atlas script. For a more about MPG($X$), see [75].

We observe that the average MPG($X$) in Linux 2.6.31 contains only 8 functions, and the majority of cases are, in fact, smaller. Only 3 of the 249 scenarios have a size above 50 functions. We also observe that MPG($X$) reduces the size of the event RCG that must be considered by an average of 56%, with reductions as high as 99%. From MPG($X$), the relevant interprocedural control flow graph can be automatically generated. In later work we refine this notion further by using Atlas to compute an Event Flow Graph (EFG), which discards irrelevant branch conditions of the CFG by retaining only the governing conditions which may affect the two-event property. To accomplish this, Atlas is used to perform a series of graph transformations to the CFG, shown in Figure 3.5. We find that the EFG is often 80-90% smaller than the original CFG. [2]
The use of Atlas to verify safe synchronization in Linux is a perfect example of the potency of the man + machine philosophy. The automation features of Atlas can be used to discharge simple cases. For the remaining cases, automation can be used to compute MPG(X) and EFG. From an EFG, we are able to discharge the vast majority of potential synchronization scenarios in a fully-automated fashion. We are left with a tractable number of complex synchronization scenarios which require human insight to check. The quantity of original synchronization scenarios makes manual analysis intractable, while the complexity of the remaining few scenarios makes automated analysis intractable. Our man + machine hybrid system combines the strengths of both approaches to solve the problem in its entirety.

### 3.6 Conclusion

Atlas is a powerful new software analysis platform. It can be used out-of-the-box to facilitate rapid code comprehension, and it can be extended with toolbox projects to create custom analysis tools. We describe two such custom use cases in Sections 3.4 and 3.5, and the success results of each. Our ICSE 2014 demonstration will explore the code comprehension and malware
analysis use cases. By embracing the Fred Brooks hypotheses [88], Atlas brings feasibility to difficult analysis problems which are resistant to manual effort or automation alone.

3.7 Appendix\textsuperscript{5}

3.7.1 Personal Contributions

As the paper notes, Atlas is a commercial product made by EnSoft Corp. EnSoft was an ISU subcontractor during the APAC project; from 2011 to 2015, I worked closely with EnSoft to evolve and shape Atlas to fit our needs. This collaboration included feedback and requests for the software graph schema, new and existing query APIs, bug reports, and much more. I interned at EnSoft during the Summer of 2012. During my internship I added new APIs for graph modification and transformation. I was not one of the original Atlas developers, but I have done a great deal to shape its direction.

3.7.2 Post-Publication Changes

Atlas continued to evolve after this paper was published in 2014. The largest change relates to the software graph schema—Atlas is transitioning to the eXtensible Common Software Graph (XCSG)\textsuperscript{105} schema. During our participation on APAC, the schema evolved in an iterative fashion to meet our needs for writing novel security analyzers for Java programs. As APAC came to a close, there was a need to redesign the schema in a more systematic and rigorous way. Compared to the graph schema used during APAC, XCSG offers the following advantages:

- **Well-defined semantics.** In the past, an Atlas tag had occasionally-ambiguous semantics. For example, the tag corresponding to the “abstract” Java keyword meant something different when applied to a type than when it was applied to a method. In XCSG, such tags have been subdivided so that each tag has unique semantics.

- **Properly-abstracted hierarchy.** As a consequence of well-defined semantics, relationships exposed by XCSG have been organized into a disciplined hierarchy of proper ab-

\textsuperscript{5} Not included in the original publication.
stractions. In the past, more than one node kind could appear at the origin or destination of a single edge kind; this is no longer the case. Every edge kind in XCSG has unique origin and destination node types, which are well-documented.

- **Support for several languages.** Atlas currently has support for Java, Jimple, C, and C++. In order to write analyzers that will work across programming languages, common semantics among languages have been abstracted in XCSG. As an example, we recently implemented the algorithm proposed by the University of Peking for detecting natural and reducible loops in arbitrary CFGs. Because our implementation relies upon the high-level abstractions of XCSG, it works for Java, Jimple, C, and C++. The ability to write a static analyzer that is portable across several programming languages is, to our knowledge, completely novel. This idea should help to usher a new paradigm of “write once, run everywhere” static analysis tools.

Other changes since ICSE 2014 include:

- **Support for Java bytecode.** During APAC, Atlas was able to index and analyze Java source code, but not bytecode. Since APAC, Atlas has added the ability to convert Android APKs (containing Dalvik class files) and JAR archives (containing JVM class files) to Jimple, a portable intermediate bytecode language, using Soot[95]. Indexing support for Jimple has been added. I employed these new capabilities to implement FlowMiner, which is discussed in Chapter 5.

- **Enhanced support for C and C++.**

### 3.7.3 Tools and Documentation

At the time of this writing, the current version of Atlas was 2.1.7. Atlas is available for download from EnSoft Corp, with free academic and trial licenses available:

http://www.ensoftcorp.com/atlas/
More information about the eXtensible Common Software Graph (XCSG) schema can be found at:

http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph
CHAPTER 4. SECURITY TOOLBOX FOR DETECTING NOVEL AND SOPHISTICATED MALWARE

This paper has been accepted to ICSE and will be presented in June of 2015 in Florence, Italy and published in the Companion Proceedings of the 37th International Conference on Software Engineering. Material in Section 4.9 has been added for this thesis and does not appear in the original paper. This work addresses our second research question: How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?

Benjamin Holland\textsuperscript{1,2,3}, Tom Deering\textsuperscript{3,4}, Suresh Kothari\textsuperscript{4}

Abstract

This paper presents a demo of our Security Toolbox to detect novel malware in Android apps. This Toolbox is developed through our recent research project funded by the DARPA Automated Program Analysis for Cybersecurity (APAC) project. The adversarial challenge ("Red") teams in the DARPA APAC program are tasked with designing sophisticated malware to test the bounds of malware detection technology being developed by the research and development ("Blue") teams. Our research group, a Blue team in the DARPA APAC program, proposed a “human-in-the-loop program analysis” approach to detect malware given the source or Java bytecode for an Android app. Our malware detection apparatus consists of two components: a general-purpose program analysis platform called Atlas, and a Security Toolbox built on the Atlas platform. This paper describes the major design goals, the Toolbox components to achieve the goals, and the workflow for auditing Android apps. The accompanying

\textsuperscript{1} Primary author \textsuperscript{2} Author for correspondence \textsuperscript{3} Primary researcher \textsuperscript{4} Graduate student and advisor, respectively. Department of Electrical and Computer Engineering, Iowa State University.
video illustrates features of the Toolbox through a live audit.

Video: http://youtu.be/WhcoAX3HiNU

4.1 Introduction

Searching for novel malware can be like looking for a needle in the haystack, but without knowing what a needle is or having ever seen one. In 2010 we learned of Stuxnet [97], a targeted nation-state level attack against an Iranian nuclear research site. The attack was only detected, some speculate intentionally, when it began to utilize noisy traditional attack vectors such as USB malware propagation. Recently we have seen a proliferation of high-level logic bugs in SSL [137, 138] and even a recently discovered 25-year-old logic bug in the Bash shell [139]. While most would agree that these bugs were honest mistakes, a few have speculated that some may have been added with malicious intent [25]. Since we have no way to determine intent by examining code a security analyst must consider software bugs as potential malice. In either case the consequences can be catastrophic. When the stakes are high, the current practices for malware detection are far from adequate. The DARPA APAC program aims at creating new techniques and tools to detect sophisticated Android malware capable of causing serious damage in a Department of Defense scenario.

USAF Colonel John Boyd described the OODA loop as an iterative decision cycle of observe, orient, decide, and act. Boyd developed this framework as a way to explain the unanticipated, superior agility of US fighter pilots in aerial combat situations. The paradigm of OODA loops applies equally well to the APAC context. To detect malware, our tools must be able to outmaneuver the capabilities of adversaries who will continue to develop new varieties of Android malware. Our Security Toolbox for Android is designed to utilize best-in-class automation and iteration techniques to maximize the odds of emerging victorious from this confrontation. We completed Phase I of the DARPA APAC program as the top performing Blue team.
4.2 Design Goals

4.2.1 Minimizing Human Effort

Goal: Minimize the human effort for (a) cross-verifying automatically detected malware, (b) performing what-if experiments to hypothesize, refine, and postulate application-specific malware that is not on the radar of automated malware detection.

We incorporate a Query-Model-Refine (QMR) program analysis platform, called Atlas [50, 48], developed by EnSoft; it provides the tool mechanics necessary for our human-in-the-loop detection of malware. We use a heterogeneous, attributed, directed graph data structure as an abstraction to represent the essential aspects of the program’s syntax and semantics (structure, control flow, and data flow), which are required to reason about software. Atlas constructs this graph from a set of software projects provided by the user. Atlas offers an expressive query language for users to write composable analyzers. Analyzers compute results in the form of subgraphs relevant to the query (evidence), which can be visualized. Based on the evidence, users can issue further queries, possibly involving information beyond specific program artifacts (e.g., looking for a specific URL). The above iteration continues until the user is satisfied with the analysis. The Security Toolbox includes analyzers using the Atlas query language. These analyzers incorporate Android semantics and they can be invoked programmatically or through interactive “Smart Views,” described in Section 4.4.5.

4.2.2 Incorporating Android Semantics

Goal: Incorporate rich and complex semantics that Android provides to facilitate development of mobile apps.

To address the semantics of Android, the Security Toolbox incorporates the permission mapping between Android APIs and the permissions each API requires. The Toolbox also incorporates semantics of fundamental Android components such as Activities, Services, Content providers, and Broadcast receivers\(^5\), as well as Android specific XML resources\(^6\).


\(^6\) [https://developer.android.com/guide/topics/resources/providing-resources.html](https://developer.android.com/guide/topics/resources/providing-resources.html)
We have developed new algorithms to automatically summarize all the Android APIs. This is work in progress and when completed we will incorporate these summaries in our Toolbox and also make them available to others in a portable format.

4.2.3 Evolution and User-friendly Design

Goal: Develop a detection tool that is evolution-friendly and highly usable.

We have a decoupled architecture to achieve this goal. The malware detection capability is decoupled and built on top of the program analysis platform (Atlas). The underlying design philosophy is similar to platforms like Matlab or Mathematica with domain-specific toolboxes built on top of general-purpose machinery. The low-level static analysis resides inside Atlas, and the malware detection capability resides inside the Toolbox as analyzers using Atlas queries. Refining and extending the existing detection capabilities as well as creating entirely new capabilities is relatively easy because it can all be done through query-enabled analyzers. Since creating a complete list of malware properties is unrealistic, it is imperative that it be relatively simple to expand the cookbook of ready-made properties through the use of adversarial thinking.

4.3 Use Cases for the Security Toolbox

The Security Toolbox is useful for nearly any Android malware detection task, with three main use cases described as follows:

- Automated detection of Android malware that has a clearly defined specification
- Production of evidence to support conclusions of automated analysis
- Enabling the human to perform what-if experiments to hypothesize and detect new malware that cannot be detected automatically because its pattern or specification is not known a priori

4.4 Components

The Security Toolbox is logically separated into several components as detailed below.
4.4.1 Permission Mapping

Android’s sensitive functionalities such as sending and receiving text messages, accessing geo-location information, or accessing user contacts are protected by runtime checks that enforce whether or not an application has been granted permission to invoke such functionalities. The Security Toolbox leverages the permission mapping produced by the Toronto PScout research group [15]. For each API version of Android, we transform the PScout mapping to an XML file that precisely represents the permission protected methods. The Toolbox contains code for parsing an Application’s manifest, and uses the XML file to automatically annotate the correct API mapping onto the Atlas program graph. We have automatically scraped and encoded into Java objects the Google developer documentation for permissions, permission groups, and protection levels to aid in developing analyzers. Additionally we have recovered mappings for Android permissions to protection levels, and permissions to permission groups by mining their relationships from the Android source\(^7\).

4.4.2 Indexers

The Atlas program graph provides much of the information needed to analyze programs, but some information is a conservative estimate. One example is type inference where the dynamic dispatch edges may be conservatively resolved to many potential targets. To address this problem, the Security Toolbox implements a Rapid Type Analysis (RTA) [22] strategy to exclude call edges to methods that should not be possible at runtime based on observed constructor calls. The Type Inference Indexer performs the RTA analysis and annotates the edges in the program graph for use by other analyzers. Since Android makes extensive use of XML for its user interface, manifest, and other resources many important program artifacts are missing in the Java program graph produced by Atlas. The Security Toolbox provides indexers to annotate and add missing program elements from these resources to the Atlas program graph.

\(^7\) https://github.com/EnSoftCorp/android-essentials-toolbox
4.4.3 Analyzers

The Security Toolbox defines an Analyzer Interface that encapsulates the logic for traversing a program graph to extract an "envelope" (a subgraph that is either empty if the security property is satisfied or non-empty containing the necessary information to locate the violation of the security property). Analyzers encapsulate their descriptions, assumptions, and possible continuations to refine results or broaden a traversal. For example, one possible continuation for a data flow based taint analysis between a sensitive source and a sensitive sink that produced a graph that is too large to interpret would be to perform the same taint analysis with call, object, type, and flow sensitivities enabled. Analyzers have been subdivided into property, smell, confidentiality, integrity, and availability analyzers. A property is something the analyst should be aware of, but does not necessarily indicate malice, such as uses of native code. A smell is a heuristic similar to a property that indicates a stronger suspicion, which demands a justification such as using Java reflection to invoke a private API. The confidentiality, integrity, and availability (CIA) analyzers detect violations of CIA properties using taint analysis of sources and sinks, modification operations on sensitive mutables, and loop detection of expensive resources respectively.

Listing 4.1 shows the queries an analyzer could use to detect high priority broadcast blockers, which could be used to intercept and block SMS messages on an Android device. Lines (1-3) select DECLARES, CALL, and OVERRIDE subgraphs; (4-5) selects abortBroadCast methods including overridden methods; (6-7) selects BroadcastReciever onReceive methods including overridden methods; (8) selects classes registered with a high priority in the Android manifest;
(9) selects high priority onReceive methods; (10) selects CALL graphs that have an edge between the high priority onReceive methods and abortBroadcast methods.

4.4.4 Dashboard

The Dashboard (shown in Figure 4.1) is an interface for automating the execution and managing results of the Toolbox’s automated analyzers. The Dashboard accounts for analyzer dependencies to enable the highest amount of parallel computation while running a multitude of analyzers. As results are computed, they are presented to the analyst in the work item queue on the right of the Dashboard. Results can be filtered by category and marked as reviewed. Optionally an analyst can make additional notes on a work item. Since work items correspond to subgraphs of the program graph, they can be named and even colored to help identify separate program subsystems. Program artifacts can be manually added or removed from a work item based on the colors given to program artifacts.
4.4.5 Smart Views

Smart Views are developed from the observation that there are several graph traversal queries that analysts use over and over again during audits, such as forward and reverse control and data flows, or discovering the declarative structures and instantiations of an object. To speed up such tasks, a graph for each of these queries can be automatically generated in response to mouse selection events on relevant source code or existing graph components. Smart Views can be customized for particular Android-specific analysis tasks, such as showing user interface XML button event callbacks.

Figure 4.2 shows a customized Smart View showing a reverse data flow program slice that includes program artifacts in the Android XML resources. The graph got generated when the user clicked on the "destination" field in the source window to inspect its value.

4.5 Workflow

The workflow of an audit follows a comprehension-driven model of an iterative Observe-Orient-Decide-Act (OODA) decision loop. An audit starts by running the Dashboard, which
produces evidence for the human analyst to inspect. This information helps the analyst observe program behaviors and orient that information within the context of the application. Aided by this information, the analyst can prioritize his exploratory hypotheses to discover malice. To aid in testing a hypothesis, Smart Views are used to quickly follow control and data flows or perform a targeted analysis such as a symbolic analysis, Android Intent resolution, or matching exception throw and catch sites. Confirming a hypothesis either results in the discovery of malware or results in more hypotheses to explore, which begins the process anew. If a hypothesis set becomes depleted an audit is halted, and audits of remaining applications are reprioritized. Finally, after the discovery of malware, the Security Toolbox is adapted by writing new analyzers to raise the bar for future automated analysis.

4.6 Evaluation

By the end of Phase I of the DARPA APAC project, our team audited 77 Android applications developed by the Red team, of which 62 contained novel malware able to evade current automatic detection techniques. DARPA employed a control team to use current state of the art tools to audit the apps along side Blue team performers. Our process correctly classified 66 (85.7%) apps as malicious or benign, found unintended malicious behaviours in 6 (7.8%) apps, and missed malware in only 5 (6.5%) of the apps consistently beating the control team. We completed Phase I as the top performing Blue team.

4.7 Related Work

A number of tools and techniques have been developed for detecting malware in Android apps including some based on static analysis [70, 113, 143, 153, 63, 1] and those based on dynamic analysis [59, 121]. These automated detection methods fall into two general categories: 1) signature-based and 2) machine learning-based. Signature-based approaches can be easily evaded by bytecode-level transformation attacks. Learning based approaches extract features from application syntax, rather than program semantics, and are also subject to evasion.
Berkeley [65] was the first to mine a mapping between Android permissions and the corresponding permission protected APIs using a dynamic analysis approach to randomly call APIs. Toronto later improved on Berkeley’s incomplete mapping with a quicker, less involved, static analysis approach that mined complete public and private API mappings from the Android source code [15]. Our Toolbox incorporates the Toronto mapping.

4.8 Conclusion

Our novel human-in-loop approach to detect Android malware minimizes human effort by allowing the human to use the evidence produced by the machine to focus their effort on further machine-assisted reasoning. This affords greater opportunity to detect malware that is not on the radar of an automated analyzer; the what-if experimentation capability provided by the machine enables the user to posit attacker intentions, hypothesize about the attacker’s modus operandi and tailor queries to detect sophisticated malware. Thus, our approach increases automation, reduces human effort and error, and provides valuable machine assistance to detect novel and sophisticated malware.

This demo paper describes the Security Toolbox that implements our novel approach. The accompanying video shows a live audit that brings out various features of the Toolbox including the Dashboard (to run and manage automated analyzers), Permission Usage View (to list permissions and where they are used in the app), and Smart Views (to facilitate what-if experiments). We acknowledge the valuable feedback from our reviewers of this paper. Several components of the Security Toolbox are being open sourced under the MIT License at https://github.com/EnSoftCorp.

4.9 Appendix

4.9.1 Personal Contributions

The Security Toolbox was a joint research effort by our team at ISU and our subcontractor, EnSoft Corp. As a member of the APAC project from its inception, I have had a
major influence on its development. Colleague Ben Holland and I were the primary developers, writing the vast majority of the code that composes the Security Toolbox. I wrote critical and complex components, including an indexer for Android XML resources, a symbolic value analysis engine, a software graph alignment engine, and a state-of-the-art iterative type inference engine. I also wrote a large number of the malware detection scripts themselves.

In addition to spearheading the implementation of the Security Toolbox, I helped to develop, implement, and evaluate its human-in-the-loop philosophy for malware detection. I participated as an analyst during the APAC malware detection experiments, using the Security Toolbox to find, understand, and report sophisticated and novel malware. Most of the iterative improvements were directly informed by our experiences during these experiments.

4.9.2 Noteworthy Unpublished Accomplishments

This subsection provides details about some impressive, though unpublished, engineering tooling that I created while developing the Security Toolbox.

4.9.2.1 Indexer for Android XML Resources

Android encourages developers to make heavy use of resource XML files when designing an application. Rather than defining things like UI layouts and string constants in Java, it is recommended for these elements to be codified in XML and separated from the core application logic. In this way, Android can automatically select the proper layout file for the given device size and orientation, the proper localized strings for the user's language, etc. At runtime, an application can reference XML resources using automatically-generated identifiers, and XML can also reference Java program elements (for example, to name the callback that should be invoked when a button is pressed).

This paradigm presents a problem for static program analysis tools that operate on Java source or byte code, because important program elements are external to these artifacts. To address this problem, I created an XML Resource Indexer to parse and interpret XML, inserting supplementary nodes and edges into the Atlas program graph. With this added representation
```java
class WithValues {
    String suffix = "\n\c\o\m";
    String res = prefix() + "evil" + suffix;
    String prefix() {
        return "www.";
    }
}
```

Listing 4.2: Malicious Android app using AsyncTask library class to leak data.

of XML resources, we were able to develop analyzers with complete knowledge of both the Java and XML portions of an app.

### 4.9.2.2 Value Analysis Framework

For program analysis tasks, including malware detection, we must often answer questions such as *Where can value V appear in the program?*, and *What values can appear at point P in the program?* During APAC, answers to these questions were critical; without them, we could not reliably or automatically determine:

- The destinations of Intent objects
- The contents of URLs, UI labels, and other strings
- The targets of reflective accesses
- Important value parameters of sensitive API calls
- ... much more

To solve this problem, I created a core library of the Security Toolbox called ValueUtils. ValueUtils allows a program data flow graph to be executed symbolically so that potential values can be determined. Figure 4.3 illustrates this symbolic execution for the program in Listing 4.2. The example shows the symbolic reconstruction of a malicious URL value that is dynamically-assembled by the application.
While symbolic value analysis is not an academically-novel concept, the creation of a value analysis library for the Security Toolbox was a critical step forward for our participation in APAC, and enabling both automated and interactive queries value-based queries.

4.9.2.3 Type-Sensitive Global Type Inference

There are two kinds of function invocations in most programming languages. Static function calls have targets that are known at compile time and can be explicitly addressed in the compiled binary. Virtual function calls, known in Java as dynamic dispatches, require runtime information to dynamically-compute the target of a call. In a language like C, this runtime information would be the value of a dereferenced function pointer. In Java, this information is the runtime type of the object upon which a call is made.

Since references in Java (and most languages) can have declared types that are more general than their runtime types, dynamic dispatches present a challenge for constructing sound call graphs. One “safe” approach is to over-approximate and compute the set of all possible targets.
Let \( DF \) be initialized to sound (non-dispatch) data flow edges

Let \( \text{TYPES} \) map be initialized for new statements, references to final types

Initialize \( \text{FRONTIER} \) to references with types (new and final from above)

While element \( \text{FROM} \) exists on the \( \text{FRONTIER} \)

Pop \( \text{FROM} \) from the \( \text{FRONTIER} \)

For each outgoing edge in \( DF \)

Let \( T0 \) be recipient element

If \( T0 \) is a cast

Add \( T0 \) to next frontier iff \( \geq 1 \) compatible \( \text{TYPES} \text{(FROM)} \) added to \( \text{TYPES} \text{(TO)} \)

Else

Add \( T0 \) to next frontier iff \( \geq 1 \) \( \text{TYPES} \text{(FROM)} \) added to \( \text{TYPES} \text{(TO)} \)

If \( \text{TYPES} \text{(TO)} \) was updated and is "this."

Recompute dispatch for corresponding callsite

Add new edges to \( DF \)

Listing 4.3: Pseudocode for a fixed-point, type-sensitive global type inference algorithm.

This is the default approach taken by \textit{Atlas} and many other tools. Unfortunately, an unsound call graph leads to subsequent analyses that produce many false positives. This is because the conservative set of possible targets is usually much larger than the feasible set at runtime.

To compute a tighter and more accurate call graph, we require \textit{type inference}, a technique for inferring the much smaller set of feasible runtime types for each object reference. Type inference works by starting from the type information we know (new statements) and propagates that information forward to infer types of other references. I implemented the state-of-the-art global type inference algorithm shown in Listing 4.2 to do this for the \textit{Security Toolbox}. The algorithm maintains a frontier of nodes in the data flow graph with new type information to communicate to successor nodes. This type information is iteratively propagated forward along sound data flow edges. When new type information reaches a dynamic dispatch call site, the new target (if any) is computed, and new data flow edges are added to the sound edge set. The algorithm terminates when it reaches a fixed point (type information has been fully-propagated). This approach is the current state-of-the-art for global type inference in the literature.

4.9.2.4 Software Graph Alignment

There are many reasons why one might wish to align (determine an equivalence mapping between) two software graphs (shown in Figure 4.4), including smart differencing, plagiarism and clone detection, etc. On the APAC project, we had at least two motivations for this capability. First, given two versions of an application, we wanted to be able to scrutinize the
semantic differences between them. Second, we wanted to be able to identify the public API surface of Android from within the larger program graph of its full implementation, for the purposes of developing FlowMiner.

Figure 4.4: Alignment of two software graphs

To address these needs, I created a software graph alignment algorithm for the Security Toolbox. It consists of (i) a parallelized graph differencing algorithm, and (ii) a set of matching rules describing how to compare two Java software graphs. Graph differencing is an extremely well-studied topic; nevertheless, this work was a major engineering feat that was crucial for achieving our goals on the APAC program.

4.9.3 Tools and Documentation

The various Security Toolbox components are currently under the process of being selectively open-sourced. In an effort to avoid training malware authors, we are reviewing our
tooling prior to public release. Those components that have been released already can be found at:

https://github.com/EnSoftCorp
CHAPTER 5. FLOWMINER: AUTOMATIC EXTRACTION OF LIBRARY DATA-FLOW SEMANTICS FOR PARTIAL PROGRAM ANALYSIS

We are in the process of submitting an abbreviated version of this paper to the 30th IEEE/ACM International Conference on Automated Software Engineering. We also plan to submit a longer version of this paper with enhanced empirical evaluation and discussion as a journal publication. Material in Section 5.11 has been added for this thesis and will not appear in the published papers. This work addresses our third research question: How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?

Tom Deering\textsuperscript{1234}, Ganesh Ram Santhanam\textsuperscript{1}, Suresh Kothari\textsuperscript{4}

Abstract

Static program analysis tools are critical to the field of software engineering, allowing us to compile, refactor, verify, and understand our code. Because modern software is built on top of reusable libraries and frameworks, whole program analysis is prohibitively expensive, hence tools must instead perform partial program analysis - analysis of a proper subset of a program's implementation. Missing data flow semantics of these components introduce problematic gaps for many use cases, including security-critical analyses. Prior attempts to overcome this, including hand-written models, heuristics, and dynamically-inferred specifications, are too coarse for many analysis use cases, introducing inaccuracies.

\textsuperscript{1} Primary author \textsuperscript{2} Author for correspondence \textsuperscript{3} Primary researcher \textsuperscript{4} Graduate student and advisor, respectively. Department of Electrical and Computer Engineering, Iowa State University.
In this paper we propose FlowMiner, a tool to mine expressive data-flow summaries from Java library binaries to enable complete and accurate partial program analysis. We are the first to create fine-grained summaries that can be used in a context, type, field, object and flow-sensitive manner. We also emphasize compaction—flow details that are not critical for accurate use are elided into simple edges between elements which are accuracy-critical. As a result, summaries extracted by FlowMiner are an order of magnitude smaller than the original library in size. The salient features of our technique are: (i) novel algorithms to extract expressive, fine-grained summary data flow semantics from a Java library, (ii) compactness of the summaries with respect to the original libraries, (iii) graph summarization paradigm that uses a multi-attributed directed graph as the mathematical abstraction to store summaries, (iv) open-source implementation (FlowMiner) of the above that saves summaries in a portable format usable by existing analysis tools, and (v) validation of our work on some of the popular Java libraries. We discuss the characteristics of our summaries versus those from other state-of-the-art tooling. We also demonstrate that our work allows our existing analysis tools to accurately handle previously unaddressed data flows in Android applications.

**Website:** [http://powerofpi.github.io/FlowMiner/](http://powerofpi.github.io/FlowMiner/)

### 5.1 Introduction

Static analysis has emerged as a powerful paradigm [41, 107, 43] for the analysis of real world software, as evidenced by the widespread use of static analysis tools in the government and industry. Despite their success, static analysis techniques share a common Achilles heel when it comes to partial program analysis, i.e., the analysis of a proper subset of a program’s implementation. Real-world software applications are built on top of reusable libraries and frameworks (see Figure 5.1) that are often much larger than the applications which use them. For example, Android applications are often three orders of magnitude smaller than the Android framework itself. This makes whole program analysis, wherein such pieces would be included, infeasible. Yet the alternative of excluding these components leads to unsound and/or incomplete results in practice, which is unacceptable for safety and security critical analysis use cases (e.g., malware
Prior work to summarize library functions by relating inputs and outputs provides a better alternative to excluding libraries from analysis of an application altogether; however it is inadequate as it is too coarse (e.g., flows to or from a field in a class are counted as flow to or from the object.) to be used accurately in a future analysis. Hence, there is a pressing need for algorithms and tools that compute fine-grained and application-agnostic summaries of a library’s semantics in a way that can be reused in future analysis contexts.

In this work we present FlowMiner, a novel approach to extract fine-grained yet compact data flow summaries from a library. We employ a graphical summarization paradigm wherein summaries are expressed as multi-attributed directed graphs, which is much more expressive than coarse, binary relationships between inputs and outputs. FlowMiner extracts application-agnostic summary data flow graph semantics through a one-time analysis of library bytecode. This summary is serialized in a portable format, and can be reused by existing analysis tools to accurately and scalably analyze applications of interest.

**Motivation.** Our motivation for FlowMiner comes from a challenge we faced as participants of DARPA’s Automated Program Analysis for Cybersecurity (APAC) program, where we were tasked with creating partial program analyses for Android apps to detect malware. The typical size of apps we were asked to analyze was small (1kLOC - 100kLOC). However, Android apps are effectively plugins to the much larger Android framework – Android 4.4.4 (KitKat) contains over 2mLOC, which is orders of magnitude larger than the size of an app. Interaction between apps and the framework is ubiquitous. For example, there are many information flows...
that pass back and forth between the app and framework, often asynchronously, that must be tracked to uncover possible malicious behaviors. Whole program analysis (including the Android framework itself) solves the problem, but destroys scalability.

**Optimizing Expressiveness and Compactness.** When summarizing the data flow semantics of a library, certain key artifacts in the library will be crucial to its data flow. For example, individual field definitions must be present if a summary is to be used in a field-sensitive way, and individual call sites must be preserved if library callbacks are to be captured. In Section 5.8, for example, we empirically show that 93.07% of summarized field flows will be false positives if field definitions are not retained. Consequently, fields, method call sites, literal values, and formal and informal method parameters and return values are all key artifacts of a flow that must be preserved in a summary data flow.

On the other hand, non-key features such as uninteresting def-use chains of assignments do not add value to the paths in which they participate, and can be abstracted away in the summary. FlowMiner elides (replaces paths with direct edges) uninteresting flow details to arrive at an abstract data flow graph that contains the key artifacts crucial to the data flow and reachability information between them, and is much more compact than the original program graph. This allows us to achieve significant savings and enhanced scalability versus the original library, while preserving soundness. In other words, the flows that are preserved in FlowMiner’s summary are precisely those that are actually possible at runtime. We find that our summaries only contain 26.89% of the nodes and 16.32% of the edges of the original library program graphs, on average.

**Contributions.** In summary, the following are the contributions of this paper.

- We develop a static analysis technique to automatically generate fine-grained, expressive summary specifications given the source or bytecode of any Java library.

  - Our algorithms identify and retain key artifacts of the program semantics necessary to allow context, object, flow, field, and type-sensitive data flow analyses in the future when using our summaries.
– Our summaries use a rich, multi-attribute graph as the mathematical abstraction to encode fine-grained summaries, rather than coarse binary relations between the inputs and outputs of library API.

– The generated summaries are compact and significantly smaller than the original library, as non-key features in the flows of the original library are elided into key paths.

- We provide FlowMiner, an open-source reference implementation [49] of our algorithms that extracts summaries given the source or bytecode of a library and exports them to a portable, tool-agnostic format.

- We validate FlowMiner by demonstrating that our summaries of popular libraries are much smaller than the original programs, yet more expressive and accurate than other state-of-the-art summary techniques.

**Organization.** The rest of the paper is organized as follows. Section 5.2 provides a motivating example of an Android application whose malicious behavior cannot be detected without data flow semantics for the Android library. Section 5.3 provides background about Atlas, the program analysis framework upon which FlowMiner is built. Sections 5.4 and 5.5 provide a precise problem statement, notation, and high-level overview of our solution, while Section 5.6 discusses implementation details. Section 5.7 provides some thoughts, limitations, and discussion of FlowMiner. We revisit our malicious Android app example, describing how summaries from FlowMiner provide a solution to our problem. We evaluate and characterize our work in Section 5.8, compare and contrast it with prior work in Section 5.9, and provide concluding thoughts in Section 5.10.

### 5.2 Motivating Example

In this section, we put forward a motivating example of an Android application with a malicious behavior that cannot be detected without summaries of library (Android) summary data flow semantics. This example is informed by our practical experiences designing tooling to
public class MainActivity extends Activity {
    private String deviceID;
    private String simSerial;
    private AsyncTask<String, Void, Void> at;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        deviceID = tm.getDeviceId();
        simSerial = tm.getSimSerialNumber();

        at = new AsyncTask<String, Void, Void>() {
            @Override
            protected Void doInBackground(String... params) {
                try {
                    String url = "http://evil.com/";
                    for (String s : params) {
                        url += "&" + s;
                    }
                    new URL(url).openConnection();
                } catch (IOException e) {
                    return null;
                }
            }
        };
    }

    @Override
    protected void onPause(){
        at.execute(deviceID, simSerial);
    }
}

Listing 5.1: Malicious Android app that uses Android’s AsyncTask library class to leak data

Malicious App. Consider the Android app shown in Listing 5.1. MainActivity is a subclass of Activity, so it defines an application screen. It overrides two lifecycle methods; the Android framework will call onCreate when MainActivity is initialized for the first time, and it will call onPause when MainActivity loses user focus. Therefore, at some point when this app is run, there will be a call to onCreate followed by a call to onPause. This triggers a latent malicious behavior.

Consider the onCreate method. On lines 9-10, the app retrieves the device ID and SIM card serial number, writing them to member fields. Lines 12-24 define and instantiate an anonymous AsyncTask, which is a threading mechanism defined by the Android library. A call to
AsyncTask.execute(params) causes Android to run the object’s doInBackground(params) method in a new thread, passing along the same arguments. Line 12 writes this anonymous AsyncTask object to a member field.

If we examine onPause(), we see that the AsyncTask is asynchronously executed with the device ID and SIM card serial number as arguments. The doInBackground method constructs a shady URL for a server operated by an attacker on lines 16-19, appending the sensitive information to the URL. Line 20 opens a connection, causing an HTTP GET request to be issued to the malicious server. This application behavior clearly will leak sensitive device data to http://evil.com.

**Analysis Without Summaries.** Consider how an analyst would hope to detect the malicious flow using a state-of-the-art static analysis tool. The analyst would first define

TelephonyManager.getDeviceId and TelephonyManager.getSimSerialNumber to be sensitive information sources, and any constructor of URL to be a sensitive information sink. The analyst would then run a static analysis tool, hoping to detect data flows from any of the sources to any of the sinks. Figure 5.2 presents a software graph with the data flows between the app and the Android framework that will be tracked by a static analysis tool such as Atlas [50] (without access to summaries of external library (Android) semantics). Observe that an Atlas-based analysis is able to follow the data flows from Android’s

TelephonyManager into the onCreate method, then through member field definitions, leading to the parameters of a call to AsyncTask.execute\(^5\) (defined by Android). At this point, the analyzer can follow the flow no further, as it has no information about the internal (private) implementation of AsyncTask. In other words, the static analysis fails to detect the malicious data flow because data flow semantics for the Android library are unavailable.

To solve this problem and identify the malicious flow via static analysis, we either have to (a) resort to whole-program analysis by including the entire Android implementation along with the app as input to the static analyzer, which is prohibitively expensive for most real-world libraries; or (b) include summary data flow semantics for Android that precisely define the data flow information between Android components necessary to track data flow through Android.

\(^5\) This partial flow can be tracked by many state-of-the-art tools, including Atlas.
Figure 5.2: Data flows inferred by static analysis of a malicious app without Android semantics fails to detect the malicious flow (created by an ATLAS script to follow data flows forward to detect source/sink flows.)
In this example, we require a summary of how data passed to `AsyncTask.execute` flows through the private implementation of Android and back into the app via an asynchronous callback.

In Section 5.4, we provide an overview of our solution for computing precise summaries of a library. We perform an automatic, one-time extraction of summary data flow semantics within a given library (such as Android). We demonstrate how these summaries can be grafted into the partial program analysis context, enabling us to detect the malicious program behavior presented in the example above. The resolution of this example is described in Section 5.7.

5.3 Background: Atlas Program Analysis Platform

In this section, we provide the reader with background knowledge about the Atlas [50] program analysis platform. Atlas is a static program analysis platform for C, Java source code, and Java bytecode developed by EnSoft Corp, which can be used to develop custom and sophisticated software analyzers [80]. Before we describe our approach to summary generation, we briefly describe the necessary infrastructure for developing FlowMiner that is provided by Atlas. For the remainder of this paper, we assume the analysis of Java bytecode and use terminology considered standard in the Java Language Specification.

**Atlas Program Graph.** Given a program $\Psi$, *Atlas* starts with the abstract syntax tree (AST) and employs a number of polynomial-time static analyzers to construct a rich multi-attributed software graph called the *Atlas Program Graph* (APG, denoted $G(\Psi)$). Using attributes, elements are arranged and expressed according to the eXtensible Common Software Graph (XCSG) [105] schema. Software artifacts such as variables, parameters to a method, call sites, classes, methods, etc. in $\Psi$ compose the nodes of the $G(\Psi)$, and relationships between those artifacts are expressed as edges. XCSG expresses both structural relationships (contains, overrides, extends, etc.) and runtime relationships (data flow, call, control flow, etc.) between program artifacts. $G(\Psi)$ is compactly stored using an optimized in-memory graph database, allowing for fast retrieval and traversal of artifacts. A subset of the important nodes and structural edge kinds is shown in Figure 5.3.
Nodes

- **Project**: Java project
- **Library**: Java library
- **Method**: Method of a class
- **Type**: Type definition
  - **Primitive**: Primitive type
  - **Array Type**: Array type
  - **Annotation**: Annotation type
  - **Interface**: Interface type
  - **Class**: Class type
    - * **Enum**: Enum type
- **Variable**
  - **Field**: Field of a type
  - **Parameter**: Parameter of a method
  - **Identity**: Implicit "this" parameter
  - **Return**: Return value of a method
  - **Enum Constant**: Constant enum value
  - **Local**: Local variable

Structural Edges

- **Declares**: Origin declares dest
- **Overrides**: Origin method overrides dest method
- **Supertype**: Dest is a supertype of origin
  - **Extends**: Origin extends dest
  - **Implements**: Origin implements dest
- **Typeof**: Origin is of type dest
- **Returns**: Origin method return type is dest
- **Element type**: Type of array components

Tags

- **Public**: Public keyword
- **Protected**: Protected keyword
- **Private**: Private keyword
- **Abstract**: Abstract keyword
- **Native**: Native keyword
- **Static**: Static keyword
- **Synchronized**: Synchronized keyword
- **Transient**: Transient keyword
- **Volatile**: Volatile keyword

Attributes

- **Name**: Name of element
- **Identifier**: Unique identifier
- **Parameter Index**: Index of parameter in method
- **Array Dimension**: Dimension of array type

Figure 5.3: Subset of ATLAS’ eXtensible Common Software Graph (XCSG) schema.
Listing 5.2: Atlas script to find parameters named "p" of methods of local types.

Attributes and Tags in $G(\mathcal{P})$. Atlas graph elements contain tags (labels) and attributes (properties). Atlas uses tags to express a hierarchy of node and edge kinds, a subset of which appear in Figure 5.3, and also to communicate boolean properties, such as the presence or absence of language keywords. Attributes express multi-valued properties of an element, such as its name or parameter index, and thus provide a mechanism for filtering results. A subset of common attributes are also shown in Figure 5.3.

Querying $G(\mathcal{P})$. Atlas provides a rich and expressive language to query $G(\mathcal{P})$ with respect to constraints on specific tags and attributes. One can develop custom scripts on top of Atlas to select graph elements, traverse nodes and edges, and construct non-trivial subgraphs of $G(\mathcal{P})$ that reveal specific structural or behavioral aspects of $\mathcal{P}$. Common tasks include the computation of call graphs, control flow graphs, data flow graphs, and type hierarchies. More exotic uses are also easy to encode; for example, one could identify the parameters named $p$ of methods of local types (types declared within a method) by selecting the nodes tagged as methods, traversing forward on contains edges to find local types and their declarations, filtering to nodes tagged as parameters, then filtering by those whose name is $p$. This example script is shown in Listing 5.2.

Queries can be issued in an interactive shell environment, or may be composed programatically using a Java API and packaged conveniently as Eclipse plugins. Intermediate and final script analysis results, which are subgraphs of $G(\mathcal{P})$, can be conveniently visualized through an Atlas graph editor view. The graph in Figure 5.2 is an example of a result shown in a graph editor visualization. For more details on Atlas, its features and its query language, we refer the reader to [50] and the online tutorial [79].

We use the program graph $G(\mathcal{P})$ as a starting point to extract summaries. We then represent the extracted summaries using an extension of the node and edge kinds in XCSG (to distinctly
represent summary flows). The precomputed artifacts in $G(\mathcal{P})$ that serve as raw material for our summary extraction approach include:

- Program declarative structure
- Type hierarchy relationships (type points to a type it extends or implements)
- Method override relationships (method points to a method definition that it overrides)
- Static type relationships (variable points to its declared type)
- Call site information
  - Method signature
  - Type to search
  - Informal parameters
- Pre-computed data flow relationships (variable points to its flow destination)
  - Field reads and writes
  - Local def-use chains
  - Local array accesses

The algorithms for extracting data flow summaries of a program $\mathcal{P}$ are described in the following sections in terms of the program graph $G(\mathcal{P})$ that ATLAS provides.

### 5.4 Approach

In this section we provide a high-level overview of our novel approach to automatically-extract summary library data flow semantics. Our approach has the following desirable attributes:

- Targets JVM bytecode for wide applicability
- Automatically extracts summaries without manual effort
• Retains details needed for context, object, field, and type-sensitive use

• Uses portable encoding to allow use by any analysis tool

• Summaries are much smaller than a library itself

**Notation.** We introduce the following notation and concepts needed to explain the algorithmic aspects of our approach. Let $\Psi$ be a program, and $G(\Psi)$ be its corresponding program graph pre-processed by Atlas. Let $M$ be the set of methods defined in $\Psi$. For each method $m_i \in M$ let the set $P_i = \{p_1^i, p_2^i, \ldots, p_{P_i}^i\}$ denote the formal parameters to $m_i$, and $r_i$ its return. We denote a method call site by $c := \langle m_j, t^c, P^c, r^c \rangle$ with $P^c$ denoting the set of arguments (parameters passed) from the call site $c$ to $m_j$ and $r^c$ denoting the returned type from $m_j$. $t^c$ denotes either the `class` where $m_j$ is defined (if $c$ is a static dispatch), or else the stated type of the reference on which $m_j$ is invoked (if $c$ is a dynamic dispatch). Statically-dispatched call sites do not require runtime information to calculate the target of the call. These include calls to static methods and constructors. Dynamically-dispatched call sites do require runtime information to calculate the destination, as is the case for calls to general member methods.

**Remark 1.** An interesting case arises when an application defines a subtype of a library type – this may introduce new potential runtime targets in the application for dynamic dispatch call sites in the library (callbacks). For example, an application may define custom subkinds of the `java.util.List` interface and pass instances of these types as parameters of calls to the library. Hence, in order for the computed data flow summaries of the library to be strictly application-agnostic and complete, they cannot pre-resolve a dynamically-dispatched callsite a priori. Our approach to computing data flow summaries adheres to this principle, which we call the **open world assumption** for computing summaries.

It is worth mentioning here that all the artifacts of the program $\Psi$ noted above are stored explicitly, but compactly, in the program graph $G(\Psi)$ constructed by Atlas, and can be queried via the Atlas API.

**Illustration of Approach.** To illustrate the approach taken to extract summaries from $G(\Psi)$, consider the two methods, `sum` and `average`, defined in Listing 5.3. A subset of the Atlas program
Listing 5.3: Malicious Android app using AsyncTask library class to leak data.

```
static int average(List<Integer> l){
    int lSum = sum(l);
    int lLength = l.size();
    return lSum / lLength;
}
```

```
static int sum(List<Integer> l){
    int s = 0;
    for(Integer i : l) s += i;
    return s;
}
```

graph $G(P)$ for the corresponding code is shown in Figure 5.4. Our goal is to arrive at the data flow summaries in Figure 5.5. Observe that the summary graph is derived from the original program graph provided by ATLAS; undistinguished nodes from $G(P)$ are removed to simplify the summary flow semantics. However, the summary graph retains critical features of the flows, such as literal values, call sites, method signature elements, and the flows between them.

To get from $G(P)$ in Figure 5.4 to $G^S(P)$ in Figure 5.5, we perform the following high-level steps:

1. Compute the program graph $G(P)$
2. Identify the set of key nodes in $G(P)$ (highlighted in cyan in Figure 5.4)
3. Compute flows between key nodes, eliding paths through non-key nodes into simple edges.
4. Compute inter-procedural summary flows by analyzing callsites

There are important differences between $G(P)$ and the summaries $G^S(P)$ that are produced. Nodes in the program graph that are important or key features of a data flow, such as formal method parameters, method return nodes, and literal values, are all retained in the summary graph. On the other hand, intermediate nodes and edges in the program graph between key nodes are elided in the summary. For Listing 5.3, the key nodes for the program graph are colored cyan in Figure 5.4. These are the nodes retained in the summary graph (Figure 5.5).

When intermediate nodes along a flow from key node $k_1$ to $k_2$ are removed from the program graph, a summary edge is introduced between $k_1$ to $k_2$ to convey the existence of a summary data flow. For example, in the summary of the `average` method, the nodes corresponding to the
Figure 5.4: Partial program graph $G(P)$ for $P$ in Listing 5.3 with key nodes highlighted in cyan

Figure 5.5: Elided local flow summary $G^S(P)$ for Figure 5.4
variables `Sum`, `Length`, and the operator `/` are intermediate nodes in Figure 5.4 that are *elided* in the summary in Figure 5.5. In their place are direct summary flow edges from the callsites of `sum` and `List.size` to the return value of the method.

**Problem Statement.** Given a program `P`, we formulate the problem of summary extraction as the procedure of automatically generating the summary graph `G^S(P)`. In the next section, we describe algorithms for each high-level step to automatically compute the summaries from `G(P)`.

### 5.5 Automatic Summary Extraction

Let `P` be a Java library for which we would like to extract summary data flows. We perform a one-time analysis of `P` using the *Atlas*, which constructs the *Atlas* program graph `G(P)` for us. We explain our technique for summary computation in two parts. The first subsection describes in detail our algorithm for computing summaries of (local) data flows within each method, and the following subsection describes the corresponding algorithms for summarizing interprocedural data flows.

#### 5.5.1 Mining Local Flows

Before we proceed to describe the algorithm to mine summary data flows local to a method, we introduce the notion of *key* nodes in `G(P)`.

**Key Nodes.** We define *key* artifacts as precisely those nodes in the `G(P)` that must be preserved in the summary graph `G^S(P)`. For the language of Java, the nodes we consider key include:

- Method signature elements
  - Formal parameters
  - Formal implicit identity parameter
  - Return node
- Call sites
Informal parameters

Informal implicit identity parameter

Return value

- Fields
- Literal values
- Definitions written to fields
- Definitions read from fields
- Array access operators and operands
  - Array reference operand
  - Array index operand
- For-each loop iterables and receivers
- Array Components

Remark 2. The key nodes in $G(\Psi)$ will differ based on the language of the library, and hence the notion of key nodes must be well defined for the library’s language prior to using our approach. For example, $G(\Psi)$ for a library written in the C language will not contain the implicit identity parameter, but may contain other key nodes such as pointers to fields and functions that have to be preserved in the summary data flow.

The algorithm for extracting a summary of local data flows (i.e., within a method) is based on the idea of eliding pre-processed def-use chains with respect to a set of key nodes in the method. Given the program graph $G(\Psi)$, we begin by identifying the set $K$ of key nodes in the graph, and then reduce $G(\Psi)$ by preserving only the nodes in $K$ and the reachability information among them. As a result, all intermediate data flow nodes and edges that occur on paths between key nodes are elided for each method, resulting in a summary graph $G^S(\Psi)$ that is much smaller than $G(\Psi)$. Def-use paths occurring between key nodes in a method are merged into simple edges, but key nodes are never elided.
**Extracting Summary Flows.** Given the set $K$ and a *pre-processed* data flow graph of def-use chains provided by Atlas, the algorithm to compute elided summary data flows with respect to $K$ is shown in Algorithm 1. The procedure $\text{MineFlow}$ iterates over the key nodes in $K$. For each key node $k \in K$, $\text{MineFlow}$ finds all other key nodes $K' \subset K$ that are reachable along data flow paths that *do not include other key nodes as intermediates*, using the procedure $\text{ElidedFlow}$ (Line 3). For each such key node $k' \in K'$, $\text{MineFlow}$ introduces a summary flow edge from $k$ to $k'$ (Lines 4-5).

**Eliding Intermediate Nodes.** The procedure $\text{ElidedFlow}$ computes the set of nearest-reachable key nodes $K'$ for a given key node $k$ by exploring the data flow graph breadth-first starting from $k$. The procedure maintains a *frontier* containing the set of nodes that have to be processed, initialized to $\{k\}$. In each iteration, it adds each node $f'$ in the frontier that has a key node successor to the return value (Lines 13-15); and otherwise, it is added to the *frontier* so that further key nodes potentially reachable from $k$ via $f'$ can be searched in a future iteration (Lines 13,16-17). $\text{ElidedFlow}$ terminates when all nodes in the *frontier* have been processed (Line 12). The set of nodes returned by $\text{ElidedFlow}$ is exactly the set of key nodes that are reachable from $k$ via non-key intermediate nodes.

**Remark 3.** The attributes labeling each summary edge are determined based on the kind of summary relationship being represented. For instance, if the origin or destination is a field definition, then the edge will be labeled with attributes indicating that it is a data flow from or to a field. Labels are critical for distinguishing nodes and edges in a useful way; they allow us to select and filter specific kinds of elements for traversal.

Our summary schema, described in Section 5.6.1, defines other kinds of relationships as well, including array accesses, dynamic callsite information information, for-each iteration, and resolved flows to methods. Mining these relationships is straightforward, as they can be taken directly from $G(\Psi)$ for inclusion in $G^S(\Psi)$. 
Algorithm 1 Mining summary data flows

1: procedure MineFlow($K, G(P)$)
2:     for all $k \in K$ do
3:         reachable = $\leftarrow$ ElidedFlow($k, K, G(P)$)
4:     for all $k' \in$ reachable do
5:         Add summary flow edge from $k$ to $k'$
6:     end for
7: end procedure

9: procedure ElidedFlow($k, K, G(P)$)
10:     frontier $\leftarrow \{k\}$
11:     result $\leftarrow \{\emptyset\}$
12:     for all $f \in$ frontier do
13:         frontier $\leftarrow$ frontier - $f$
14:     for all $f'$ s.t. $(f, f')$ is a data flow edge in $G(P)$ do
15:         if $f' \in K$ then
16:             result $\leftarrow$ result $\cup f'$
17:         else if $f' \notin$ frontier then
18:             frontier $\leftarrow$ frontier $\cup f'$
19:         end if
20:     end for
21: end for
22: return result

5.5.2 Mining Interprocedural Flows

The task of mining interprocedural flows involved in method calls, as well as dynamic call site information, is somewhat more complex. First, we must decide which call sites to resolve at present (during summary generation) and which cannot be resolved until summaries are applied in the context of an analysis. If a potential target of a call site may lay outside of the library after an application is introduced into the analysis context, then we must not resolve targets of the call site at this time. Clearly static dispatches can be resolved during summary generation, because the targets are unambiguous even with an open-world assumption about future analysis contexts (see Remark 1).

**Resolvable and Unresolvable Call Sites.** It is important to distinguish between call sites that can be statically-resolved and those which cannot at the time of summary generation. By pre-resolving those which are statically-resolvable to their targets, we generate sound data flow relationships that a client can use, and prevent future rework by clients. Additionally, direct interprocedural flows are more compact to express than leaving a callsite description in the
summaries. Thus, it is preferable to identify and resolve statically-dispatchable callsites at the time of summary generation.

Although dynamic dispatches are not statically-resolvable in general, they become so under certain circumstances. For instance, a call to a member method marked \texttt{final} or \texttt{private} cannot possibly have polymorphic behavior, even under an open-world assumption. Similarly, a call to a member method within a type that is marked \texttt{final} or \texttt{anonymous} is also unable to result in polymorphism.

The algorithm to mine interprocedural summary flows is shown in Algorithm 2. The procedure \texttt{MineCallsiteSummaries} in Algorithm 2 calls the procedure \texttt{ClassifyCallsites} to partition the set $C$ of call sites as described above and returns (a) $R^+$ containing call sites for which targets may be unambiguously resolved even in the face of an open-world assumption at the time of summary generation, and (b) $R^-$ containing call sites for which multiple targets (presently, or in a future analysis context), may be resolved.

Next, the procedure \texttt{MineMethodFlows} is called for $R^+$. For each call site, this procedure resolves the target using a dispatch calculation\footnote{Recall that each call site in $R^+$ can be resolved to a single target.} (line 23) and adds summary flow edges in $G^S(P)$ connecting the informal call site parameters $P_c$ to the corresponding formal parameters $P_j$ in the (resolved) target method $m_j$’s definition (lines 24-27). \texttt{MineMethodFlows} concludes by connecting the return flows from the return value in the resolved method $m_j$ to the receiving variable at the call site (line 29). Finally, the procedure \texttt{MineDynamicDispatch} is called for $R^-$, wherein the dynamic dispatch information for each call site in the $G(P)$ is retained in the summary $G^S(P)$ (lines 34-37) so that a client can resolve them in a future analysis context.

\subsection{Summary Extraction Example}

It may be easiest to understand the summary extraction process by way of an example. Consider the \texttt{Integer} class from the Java standard library, a subset of which we show in Listing 5.4. The corresponding summaries are shown in Figure 5.6, where elements of $G^S(P)$ are differentiated from $G(P)$ using magenta highlighting.
Algorithm 2 Mining method flows and dynamic callsite information relationships

```
procedure MineCalleeSummaries(C)
2:   \((R^+, R^-) = \text{ClassifyCallsites}(C)\)
3:   MineMethodFlows(R+)
4:   MineDynamicDispatch(R-)
end procedure

6: procedure ClassifyCallsites(C)
8:   \(R^+ \leftarrow \emptyset\)
9:   \(R^- \leftarrow \emptyset\)
10: for all \(c \in C\) do
11:     if \(c\) is a static dispatch then
12:        \(R^+ \leftarrow R^+ \cup c\)
13:     else if \(m_i\) is final \(\lor\) private \(\lor\) constructor then
14:        \(R^+ \leftarrow R^+ \cup c\)
15:     else if \(t\) is final \(\lor\) private \(\lor\) anonymous \(\lor\) array then
16:        \(R^+ = R^+ \cup c\)
17:     else
18:        \(R^- = R^- \cup c\)
19:     end if
20: end for
21: return \((R^+, R^-)\)
end procedure

procedure MineMethodFlows(C)
22: for all \(c := (m_i, P_c, r^c, t^c) \in C\) do
23:     \(m_j \leftarrow \text{dispatch}(c)\) \(\triangleright \text{Unambiguous resolution of } c \text{ to } m_j\)
24:     \(P^c \leftarrow \{p_1^c, p_2^c, \ldots, p_{|P^c|}^c\}\)
25:     \(P_j \leftarrow \{p_1^j, p_2^j, \ldots, p_{|P^c|}^j\}\) \(\triangleright \text{Arguments passed at callsite } c\)
26:     for all \(p_k^c \in P^c\) do
27:         Add method flow summary edge \((p_k^c, p_k^j)\) to \(G(\Psi)\)
28:     end for
29:     Add return flow summary edge \((r_j, r^c)\) to \(G(\Psi)\)
30: end for
end procedure

procedure MineDynamicDispatch(C)
32: for all \(c := (m_i, P_c, r^c, t^c) \in C\) do
33:     Add dynamic callsite method edge \((c, m_i)\) to \(G(\Psi)\)
34:     Add dynamic callsite type edge \((c, t)\) to \(G(\Psi)\)
35:     for all \(p_k^c \in P^c\) do
36:         Add dynamic callsite param edge \((p_k^c, c)\) to \(G(\Psi)\)
37:     end for
38: end for
end procedure
```
public final class Integer extends Number implements Comparable<Integer> {
    private final int value;

    public Integer(int value) {
        this.value = value;
    }

    public byte byteValue() {
        return (byte) value;
    }

    public int compareTo(Integer object) {
        return compare(value, object.value);
    }

    public static int compare(int lhs, int rhs) {
        return lhs < rhs ? -1 : (lhs == rhs ? 0 : 1);
    }
    ...
}

Listing 5.4: Partial implementation of Integer from the Java standard library

We see the effects of Algorithm 1 within each summarized method. For example, the conditional operators and intermediate definitions in the compare method have been elided; the flow is simplified with respect to three possible literals that may flow to the return. The effects of Algorithm 2 can be seen in the compareTo method, where there is a statically-resolvable call to compare. FlowMiner has resolved the call automatically, showing the flow of the two informal parameters in compareTo to the formal parameter and identity parameters of compare, and the corresponding flow of the return value back to compareTo. Finally, this example also illustrates field reads and writes, which were imported directly to $G^S(\mathcal{P})$ from $G(\mathcal{P})$ during mining. Using this summary graph, we can succinctly and accurately track flows through the Integer class.

5.6 Implementation

In this section we describe the implementation details of our approach for statically-extracting, expressing, and subsequently employing data flow summaries of Java libraries.

Architecture. FlowMiner is implemented as a plugin for the popular Eclipse IDE. This choice was natural because Atlas, our underlying analysis platform, is also an Eclipse plugin. As shown in the architectural diagram of Figure 5.7, FlowMiner takes Java library bytecode as input, typically in the form of a JAR archive. Library bytecode is passed to Atlas for indexing, a process which converts an AST into a rich, queryable graph database. After in-
Figure 5.6: Summary extraction results for the `Integer` class in Listing 5.4
Figure 5.7: Architecture of FlowMiner
dexing, FlowMiner runs the polynomial-time algorithms described in Section 5.5 to extract a summarized version of the library’s data flow semantics. This summary data flow graph is packaged into a portable XML format for later use by existing static analyzers.

5.6.1 A Summary Graph Schema Extension

To express our summary data flow semantics, we propose an extension to the XCSG schema from Figure 5.3. We introduce several new kinds of local variables (nodes), as well as new relationships (edges), shown in Figure 5.8. This schema extension allows us to express the semantics of data flows within methods, flows to and from fields, as well as flows involved in static and virtual calls between methods. Like XCSG itself, our schema extension is organized hierarchically and expressed via the use of tags for kinds and subkinds.

Important Features. There are several important features to note about our summary schema extension. First, it is strictly an extension of the XCSG. The node types we introduce are specialized subtypes of local variables. These specializations represent literal values, array components and access operators, method call sites, and other important local definitions. The edge types we introduce represent summary data flows, array accesses, for-each iteration, and call site information.

Second, our summaries pertain only to data flow. While a flow edge \((A, B)\) implies the existence of a control flow path along which this flow happens, we do not retain control flow nodes and edges from \(G(P)\). This allows \(G^S(P)\) to be much more compact than the library itself, which was one of our design goals.

Third, our summaries retain sufficient information to be used with context, type, field, object, and flow sensitivity. The client using the summaries for subsequent analysis is able to decide which categories of sensitivity to employ in order to achieve the desired level of accuracy and speed. One consequence of this philosophy is that we only resolve flows for method call sites when the target can be unambiguously resolved to a single possibility with an open-world assumption. That is, no matter what other types and methods are introduced into an analysis context by an application, the resolution decision for the call site cannot be changed. We leave dynamic dispatch call sites to be resolved when summaries are applied to an analysis context,
Summary Nodes

- **Array Component**: Array components (on heap) of a referenced array
- **Array Index Op**: Array access operator, takes array reference and idx as operands and selects a component
- **Call Site**: Represents a method call, as well as the value returned.
  - **Call Site Resolved**: Statically-resolved call site
  - **Call Site Unresolved**: Dynamic call site
- **Literal**: Literal primitive or String value

Summary Edges

- **Array Access**: Connects array index operator to an array component node
- **Dynamic Callsite**: Describes an aspect of a dynamic dispatch call site
  - **Dynamic Callsite Param**: Origin stack param is informal parameter for destination call site
  - **Dynamic Callsite This**: Origin object reference is implicit identity parameter for destination call site
  - **Dynamic Callsite Signature**: Call site points to its invoked method signature
  - **Dynamic Callsite Type**: Call site points to its stated identity param type
- **Flow**: Data flow relationship
  - **Array Flow**: Flow to or from array component
  - **Field Flow**: Flow to or from a field
  - **Local Flow**: Flow between local variables
  - **Resolved Method Flow**: Flow to method parameters or from a method return
- **For Each**: Iteration over Iterable or array type to local receiver variable

Figure 5.8: Summary nodes and edges in FlowMiner’s XCSG schema extension
since we cannot know ahead of time if that context may introduce new possibilities for the target of the call site. However, we do provide the signature of the call site, as well as the informal stack parameters involved in the call, so that clients may resolve it later.

Our summary schema extension borrows philosophically from the rich XCSG schema of Atlas [50]. Our research group uses Atlas for summary extraction, application, and subsequent program analysis. Background about Atlas is provided in Section 5.3. It should be noted that the concepts discussed in this paper are general; other static analysis tools besides Atlas may be used to mine, apply, and subsequently use our summaries.

5.6.2 A Portable XML Schema

In order to serialize $G^S(\mathcal{P})$ for subsequent use in a partial program analysis, we require a portable document format. XML is a well-known, widely-used format for creating structured documents. There are many mature libraries available for verifying, parsing, and writing XML documents. Therefore, encoding XCSG with our schema extension described in Subsection 5.6.1 in terms of an XML document format seemed an obvious choice for portability and convenience. An XML schema document (XSD) defining the grammar for expressing software graphs is provided with the open source reference implementation of FlowMiner [49].

5.7 Discussion

Using Summaries. Having used Atlas and FlowMiner to perform a one-time analysis of a library to extract its data flow semantics, an existing static analyzer can apply these summaries to achieve a complete and accurate program analysis. What it means to "apply" summaries will differ based on the tooling used by the client. For analyzers implemented on top of the Atlas platform, applying summaries means translating the portable XML summary document into additional nodes and edges from $G^S(\mathcal{P})$ for insertion into the program graph $G(\mathcal{P})$ of an application. Once inserted, these supplementary data flow semantics can be used for subsequent analysis.
Recall the example malicious Android app from Section 5.2, for which a static analyzer was unable to detect the malicious behavior. The application asynchronously leaks the user’s device ID and SIM card number to an attacker. We defined the values returned by \texttt{TelephonyManager.getSimSerialNumber} and \texttt{TelephonyManager.getDeviceId} to be sensitive information, and asked our analyzer to track forward data flows from these artifacts. The result, shown in Figure 5.2, ran into a dead end as soon as the flow disappeared into the private implementation of Android’s \texttt{AsyncTask.execute} API.

After applying $G^S(P)$ extracted from a one-time analysis of Android 4.4.4, we are able to obtain the result in Figure 5.9. Summary nodes and edges ($G^S(P)$) are highlighted in magenta to distinguish them from elements of the original \textsc{Atlas} program graph ($G(P)$). By employing $G^S(P)$, our static analyzer is able to detect the entirety of the malicious flow! After the sensitive information enters \texttt{AsyncTask.execute}, our summaries of Android track the asynchronous data flow involving local flows, a method call, a write and read of a field, and finally a callback into the application (\texttt{MainActivity$\_1$.doInBackground}) on a new thread. From there, our analyzer uses $G(P)$ to follow the flow through an enhanced for loop, string concatenation, and ultimately to the \texttt{URL} constructor, completing the leak.

Thanks to our extracted summaries of Android data flow semantics, we are now able to accomplish complete and accurate partial program data flow analyses of Android applications. As mentioned previously, the techniques involved are not specific to Android; \textsc{FlowMiner} can summarize data flows within \textit{any} Java library. The Android example is motivated by our original use case for DARPA’s APAC program.

5.8 Evaluation

In this section, we evaluate the expressiveness, soundness, completeness, and scalability properties of \textsc{FlowMiner} and its summaries.

5.8.1 Expressiveness.

The data flow summaries extracted by \textsc{FlowMiner} are extremely rich and expressive. For example, the coarse information flow specifications at the granularity of object tainting
Figure 5.9: Partial program analysis of malicious app from Listing 5.1 with FlowMiner summaries of Android
generated by Clapp et al. [40] can be directly inferred from our summaries. When information in a FlowMiner summary reaches a member field definition, the corresponding "taint" on the object is implied. When information flows from a member field to a method return, it is implied that the object "taints" the method return. Therefore, FlowMiner summaries are strictly more expressive than the most closely-related prior work.

More importantly, our summaries can be used more accurately. Figure 5.10 shows how coarse specifications that taint entire objects can lead to an exponential number of implied false positive flows. The figure shows three types with two fields each. Dashed arrows represent transfer of taint at the granularity of objects, while solid arrows represent transfer of taint with field granularity. While a subset of the flows implied by object granularity are true positives (black), the majority of flows will be false positives (red). In general, a flow involving object-granularity summaries that traverses through $N$ classes (with $K$ unrelated fields each) will produce on the order of $K^N$ false positive flows!

Similarly, the presence of registration/callback pairs identified by EdgeMiner [34] can also be inferred from FlowMiner summaries. Because FlowMiner encodes details of virtual callsites for which multiple runtime targets may exist, the set of callback sites will be a subset of the callsites in $G^S(\mathfrak{P})$. A polynomial-time structural analysis of application subtypes and
method override relationships can be used to identify this subset. Then, for each potential callback site, a reverse data flow analysis on our summary semantics can be used to find APIs that "register" callbacks. Hence, the summaries extracted by FlowMiner could directly be used by EdgeMiner to identify callback and registration pairs.

5.8.2 Soundness.

We observe that the data flow summaries produced by FlowMiner are sound, i.e., there is never a false positive; if a flow is indicated, it *can* actually occur at runtime. This follows from the way in which our summaries are generated (see Section 5.4 for details).

*First*, local flows within a method follow directly from definition-use chains that are preprocessed by Atlas in a flow-sensitive manner along valid control flow paths. By definition, these flows can and do occur at runtime within a method. Our algorithm to elide (simplify) these flows with respect to key nodes preserves soundness; if a flow path from A to B was present in the original sequence of def-use relationships, it is preserved after we have elided the flow. This process never introduces spurious new flows, but rather simplifies existing flows, thus preserving soundness.

*Second*, flows to and from field definitions are sound. Unlike the information flow specifications produced by Clapp et al. [40], FlowMiner retains the granularity of individual field definitions and variables. Thus, we avoid conflating flows through unrelated fields. Additionally, FlowMiner preserves object alias attributes on field flow edges; this allows our summaries to be leveraged in an object-sensitive way to avoid false positives.

*Third*, the manner in which FlowMiner resolves interprocedural flows involved in method calls is sound. A method call site is pre-resolved to a target by FlowMiner if and only if the target can be statically resolved in the presence of an open-world assumption. That is, when the library is used by an application (which may create arbitrary subtypes and overriding method definitions), the call sites pre-resolved by FlowMiner remain statically-resolvable. For all other call sites, our summaries capture the structural details of the call site (method signature, stated type, informal parameters) that will be needed to resolve the call in the future. But we cannot and do not attempt to pre-suppose one or more runtime targets. In fact, some runtime
targets may exist only in external application contexts, so resolution is only possible given a specific client application context. Therefore, we avoid false positive flows for virtual method calls which are not statically-resolvable.

As a consequence of the above observations, the data flow summaries generated by FlowMiner are sound. Put differently, the removal of any summary flow edge would remove critical information needed later to compute a data flow in some partial program analysis context.

5.8.3 Completeness.

In the sense that it does not miss any true flows, FlowMiner provides complete summaries of data flow semantics with a few exceptions. The completeness properties of FlowMiner follow directly from the completeness properties of the Atlas analysis platform on which it is built. At this time, Atlas fully supports the features of the Java 7 programming language. In this sense, all local, field, and method flows between Java program elements will be captured. However, there are a few minor challenges to completeness.

Java, like many other languages, contains an introspection mechanism known as reflection. Java’s reflection APIs allow a program to dynamically locate and access arbitrary program elements, even allowing the violation of language-imposed information hiding. The Atlas program graph captures calls to Java’s reflection APIs but does not attempt to statically-resolve their effects. For example, a class can write to a private member field of an unrelated class with a series of calls to reflection APIs. In this case, Atlas will show these calls, but will not directly index a field write. Introspection mechanisms such as reflection present a well-known challenge for static analysis techniques in general.

Another challenge to completeness involves interactions between programming languages. For example, a Java program may call into a native C binary, which may itself call back into the Java program. While cross-language interaction is not an issue for pure Java libraries, it presents a challenge for mixed-language libraries. Our research group is currently working on an analysis solution for mixed Java and C/C++ libraries, though this paper does not describe that work.
Table 5.1: Experimental results showing the performance of FlowMiner on popular Java libraries in terms of compactness and accuracy. (* Percentage of object-granularity flows that are false positives compared to those produced by FlowMiner)

| Library              | $|V|$ | $|E|$ | $|V^s|$ | $|E^s|$ | $|V^s|/|V|$ (%) | $|E^s|/|E|$ (%) | Field Flows | Object Flows | % False Positives* eliminated |
|----------------------|-----|-----|-------|-------|----------------|----------------|--------------|--------------|-----------------------------|
| junit-4.12           | 74169 | 308696 | 22135 | 66612 | 29.84%          | 21.57%          | 890          | 3019         | 70.52%                      |
| testng-6.8.21       | 241021 | 1161421 | 84001 | 260351 | 34.85%          | 22.41%          | 366          | 4050         | 90.96%                      |
| wicket-core-6.19.0   | 390099 | 1940647 | 140786 | 431677 | 36.08%          | 22.24%          | 11172        | 80173        | 86.07%                      |
| spring-webflow-2.4.0 | 138972 | 587270 | 44399  | 134074 | 31.94%          | 22.83%          | 608          | 3620         | 83.20%                      |
| struts2-core-2.3.20  | 171171 | 763587 | 56208  | 168307 | 32.83%          | 22.04%          | 4731         | 61382        | 92.29%                      |
| commons-collections4-4.0 | 152745 | 988275 | 51735  | 159147 | 33.87%          | 16.10%          | 13248        | 69679        | 80.99%                      |
| commons-io-2.4       | 72754  | 285713 | 20984  | 63450  | 28.84%          | 22.20%          | 1711         | 8354         | 79.52%                      |
| commons-lang3-3.3.2  | 178319 | 752137 | 57006  | 174955 | 31.96%          | 23.26%          | 10280        | 34743        | 70.41%                      |
| Android 4.2.2        | 6651277 | 33964070 | 2467991 | 7664280 | 37.11%          | 22.57%          | 1129523      | 16053060     | 92.96%                      |
| Android 4.3.1        | 6867245 | 351653616 | 2547558 | 7915450 | 37.10%          | 22.51%          | 1206542      | 16816490     | 92.83%                      |
| Android 4.4.4        | 7707688 | 44150241 | 2850104 | 8855585 | 36.98%          | 20.06%          | 1216178      | 17069468     | 92.88%                      |
| Android 5.0.2        | 8684208 | 45649066 | 3217476 | 10010647 | 37.05%          | 21.93%          | 1556027      | 21874691     | 92.89%                      |
5.8.4 Experiments.

With the goal of evaluating FlowMiner’s accuracy and compactness when summarizing real world Java software libraries, we summarized a selection of popular Java libraries, including utilities, test frameworks, and web client frameworks, including the recent versions of the Android operating system\(^7\). We ran our experiments on a multi-core computer with 64 GB RAM, with Eclipse Luna installed with Atlas 2.1.5 and FlowMiner. In Table 5.1 we present the libraries we used (\(\mathcal{P}\)) and results of our experiments. We downloaded the Java libraries listed in column 1 of Table 5.1 along with their dependencies listed by the Maven central repository. We created a simple Atlas analyzer to gather the summary statistics listed in Table 5.1.

Comparing the size of the original program graph provided by Atlas (\(G(\mathcal{P})\)) to the size of the summary graph (\(G^S(\mathcal{P})\)), we see that the sizes of the summary graphs are around 30\% as large as the original program graphs, indicating that summaries produced by FlowMiner are significantly compacted.

The table also shows the number of data flow edges induced in the summary by FlowMiner (fine-grained approach that tracks data flows at field level granularity) ranges from about 7\% to 30\% of that induced by the coarse-grained approach (that tracks data flows at object level granularity). This means that over 70\% of the flows induced by coarse-grained approach are false positives as explained earlier. In particular, for Android 4.4.4, this metric (92.88\% false positives) indicates the significance of our approach to malware detection, as the Android operating system is critical for the analysis of Android apps. A similar trend is seen for three other recent versions of the Android operating system.

**Correctness** We also empirically verified the soundness and completeness of our FlowMiner implementation for each of the Java libraries (column 1 of Table 5.1) chosen for our experiments as follows. We first computed both the Atlas program graph \(G(\mathcal{P})\) and the summary graph \(G^S(\mathcal{P})\), and then successfully verified the property that there is a data flow path from one key node \(k\) to another key node \(k'\) in \(G^S(\mathcal{P})\) if and only if there is a corresponding data flow path from \(k\) to \(k'\) in \(G(\mathcal{P})\).

\(^7\) We generated the JVM bytecode for the Android operating system from the Android build for the aosp_arm-user device configuration.
5.8.5 Scalability.

We have tested FlowMiner on very large libraries. For example, Android 4.4.4 (KitKat) contains roughly 2 million lines of Java code, even after comments and white space have been omitted. At this scale, Atlas requires around 30 minutes and 20GB of memory to index the library with commodity PC hardware. FlowMiner completes its one-time analysis and export of data flow summary semantics within an additional 45 minutes. For the smaller libraries that we tested, FlowMiner needed 1 minute or less (including the time Atlas takes to construct the original program graph). Hence, FlowMiner scales well even for summarizing extremely large libraries.

In addition to the scalability of summary mining, the compactness of extracted summary artifacts is important for practical use. For example, $G^S(\Psi)$ for Android 4.4.4 contains only 36.98% of the nodes and 20.06% of the edges of $G(\Psi)$. Hence, our summaries provide an order-of-magnitude savings versus a fully-detailed program graph of a library, yet retain the critical details for use in a partial-program data flow analysis. This is a key justification for the use of summaries versus full library implementation.

5.9 Related Work

Summarizing Call Graphs. There has been a lot of recent interest in the extraction of API summaries of software libraries. Much of the work in this area focuses on extracting summaries that describe control flow transitions within the library. Such control-flow summaries are useful for routine static analysis tasks such as call graph generation [74, 150, 147, 7], tracking of non-trivial calling relationships between application and the library (e.g., asynchronous callbacks in Android) [34] and visualization of control flows from the application to the library and vice-versa [99].

Summarizing Data Flow Graphs. Despite the interest in library summaries, there is not much closely-related work in the area of mining data flows from software libraries, a task that is particularly crucial for security-critical analyses. Malware detection in Android apps [68], for
example, requires tracking the flow of sensitive information (source, e.g., IMEI number) from the mobile device to potentially harmful destinations (sinks, e.g., a location on the internet).

Callahan first proposed the program summary graph as implemented in PTOOL [33] as a way to compactly represent the inter-procedural call and data flow semantics of the whole program. Routev et al. [122] pointed out the need to use summaries of data flow semantics when analyzing applications that are dependent on large libraries. They proposed a general theoretical framework for summarizing data flow semantics of large libraries, using pre-computed summary functions per library component and building on the work of Pnueli [115].

Similarly to Routev et al., Chatterjee et al.[37] compute a summary function for each procedure in the bottom up traversal order of the call graph such that the summary of a caller is expressed in terms of the summary of the callee component(s). More recently, Routev et al. [123] described an approach for summarizing libraries by using a graph representation of dataflow summary functions, and by abstracting away redundant dataflow facts that are internal to the library, in a similar vein to our concept of eliding flows.

Some approaches compute summary information for a software component independently of the callers and callees of that component. For example, Ali et al. developed a tool, AVERROES[8], to generate a placeholder that overapproximates the behaviour of a given library. Their overapproximation may be too coarse to be useful in some scenarios such as malware detection, where we need summaries to retain enough information for various kinds of sensitive analyses.

**Summarizing Android Flows.** To the best of our knowledge, the most closely related work in summarizing libraries in the context of Android is by Clapp et al. [40], who employ a dynamic analysis approach to mine information flows from Android. In their two-part system, Droidrecord provides an instrumented Android emulation environment in which concrete execution traces can be recorded. Modelgen post-processes these trace logs to infer information flow specifications. Their approach successfully recovers 96% of a set of hand-written information flow specifications. The strengths of this work are its automation, the small size of its information flow specifications, and the fact that statically-difficult flows (reflection, cross-language flows) are captured by dynamic analysis. While this work lays a strong foundation for the
kind of library data flow summaries we desire, the following aspects limit the usefulness of the approach and suggest an alternative.

First, the authors’ use of dynamic analysis to identify information flow relationships between the input and output variables of a method inherently makes it infeasible to cover all possible paths in the library, unavoidably introducing false negatives. Second, the information flow specifications produced are too coarse to be used accurately; information flow is summarized at the level of granularity of objects rather than the granularity of fields and other variables. This causes unrelated flows to become conflated, as in Figure 5.10. For example, a method parameter that is written to a field of an object is considered to "taint" the entire object. Therefore, flows through unrelated fields are join the same taint, producing false positives.

Third, their information flow specifications express flows only between elements of the library’s public surface (API), throwing away all other flows through intermediate variables and functions. This limits the ability of a subsequent static analysis to use the mined specifications in an accurate way. For example, a static analyzer might have used the details of calling contexts and object types to restrict a summary flow to valid constraints, but these details are not present in the flow specifications. Finally, their flow specifications fail to capture potential flows from a library back into an application introduced by polymorphic call sites. For example, an application can define a subtype of a library type, providing overriding method definitions. At runtime, virtual call sites in the library may result in calls and flows back into the application.

Our work addresses all these issues. We use static analysis (built on top of the Atlas platform) instead of dynamic analysis to identify possible flows within the library. Hence, we avoid the possibility that some execution paths are not covered. Next, the flow specifications extracted by FlowMiner track and preserve data flows at the granularity of individual variables and definitions within methods and objects, so we avoid falsely merging unrelated flows. Furthermore, our flow specifications express flows among program elements that are not necessarily on the library API. This allows subsequent analyses to be context, field, type, object, and flow-sensitive. Finally, we retain the details of virtual call sites so that flows involving potential callbacks into an application are captured.
5.10 Conclusion

In this paper we have presented FlowMiner, a novel solution to the common problem of incomplete results for partial program data flow analyses. We propose a one-time analysis of a library to extract a summary of its data flow semantics. Once extracted, the summary can be leveraged to achieve complete and accurate partial program analysis. We have shown that our summaries are compact, containing on average 26.89% of the nodes and 16.32% of the edges of the original library program graphs. Yet they are also fine-grained—retaining individual field definitions, for example, avoids 93.07% of the false positive flows indicated by tainting entire objects. We have illustrated the usefulness of summary data flow semantics with a motivating malicious Android application whose malice cannot be detected without summary semantics for Android, shown that FlowMiner’s summaries solve this scenario, and compared our work against existing state-of-the-art tools. Our contributions include:

- We develop a static analysis technique to automatically generate fine-grained, expressive summary specifications given the source or bytecode of any Java library.
  
  - Our algorithms identify and retain key artifacts of the program semantics necessary to allow context, object, flow, field, and type-sensitive data flow analyses in the future when using our summaries.
  
  - Our summaries use a rich, multi-attributed graph as the mathematical abstraction to encode fine-grained summaries, rather than coarse binary relations between the inputs and outputs of library API.

  - The generated summaries are compact and significantly smaller than the original library, as non-key features in the flows of the original library are elided into key paths.

- We provide FlowMiner, an open-source reference implementation [49] of our algorithms that extracts summaries given the source or bytecode of a library and exports them to a portable, tool-agnostic format.
• We validate FlowMiner by demonstrating that our summaries of popular libraries are much smaller than the original programs, yet more expressive and accurate than other state-of-the-art summary techniques.

There are a number of future research directions that should be investigated. Data flow semantics are only one aspect of software; other aspects include control flow, domain-specific resource use (file system, network, etc.), potential error codes or exceptions thrown, permissions required, and many more. Ultimately, the desired summary of a library will depend upon the intended program analysis use cases. For example, an analysis to determine whether an application has properly-implemented error-handling would require a summary of error codes and exceptions for any libraries used. Finally, we note that the implementation described in this paper focuses on summaries of data flow semantics for Java libraries. Future work might generalize our proposed XCSG schema extension to apply to other languages and multi-language software stacks.

Acknowledgements

We would like to thank the talented team at EnSoft Corp who created the Atlas platform, including Jeremias Sauceda, Jon Matthews, and Nikhil Ranade. We would also like to thank the members of the ISU APAC team who helped to develop our malware detection tooling, including Ben Holland, Daniel Stiner, Curtis Ullerich, Akshay Deepak, Sandeep Krishnan, Damanjit Singh, Vani Bojja, and others. Finally, we would like to thank DARPA for funding this work.

5.11 Appendix

5.11.1 Personal Contributions

I served as the primary developer and author of FlowMiner, though its approach and philosophy were shaped by discussions with Suresh Kothari and Jeremias Sauceda. Experimental evaluation and authorship assistance was provided by Ganesh Ram Santhanam.

8 Not included in the original publication.
5.11.2 Tools and Documentation

FlowMiner has been open-sourced under the MIT License. Its source code can be obtained on GitHub:

https://github.com/powerofpi/FlowMiner

Instructions, support, and an update site are available for download at:

http://powerofpi.github.io/FlowMiner/
CHAPTER 6. GENERAL CONCLUSIONS

This thesis describes the identification of, solutions for, and evaluation of three important research questions motivated by our group’s participation in DARPA’s APAC program. This chapter provides a summary of my contributions, results, and suggestions for future research on each research topic.

6.1 Atlas

How should a software analysis platform be built to facilitate both automation and human comprehension?

Chapter 3 presents Atlas, a novel program analysis platform employing an attributed graph-based software abstraction. While prior work provides either automated analyses or static program visualizations, Atlas simultaneously supports automation, interaction, and program comprehension. For automation, Atlas provides a powerful embedded Java DSL for writing analyzers to query an optimized in-memory graph database. For comprehension, it provides interactive software graph visualizations with one-to-one source code correspondence, a Smart View for invoking configured scripts in response to user clicks on source tokens and graph elements, and an Interpreter View for making on-demand queries. As a sophisticated user of Atlas at ISU on the APAC program, and as a former intern at EnSoft, my contributions include:

- An API for modifying and extending the software graph using client-side indexers

- Refinements and improvements to the Smart View for dynamically invoking analysis scripts in response to clicks
• Suggestions, feedback, and improvement requests for analysis APIs and software graph schema features

• Experimental evaluation and feedback for Atlas as a platform for writing analyzers, including the Security Toolbox and FlowMiner.

• Experimental evaluation and feedback for Atlas as a program comprehension tool, as an analyst during malware detection experiments

Atlas has been experimentally-validated in several ways. Our group at ISU tested its potential as a framework for writing analyzers by successfully constructing and demonstrating the effectiveness of both the Security Toolbox and FlowMiner. The analysis APIs that Atlas provides allowed us to create powerful, fast, and accurate analyzers with minimal effort and very few lines of code. As a platform to facilitate program comprehension, Atlas was extensively tested during the APAC malware detection experiments. In conjunction with the Security Toolbox, our analysts employed the Interpreter and Smart Views to understand unfamiliar code and interactively discover malware, achieving best-in-class detection performance (see Section 6.2). Finally, other students in our research group have used Atlas to verify safe synchronization and memory safety properties of the Linux kernel with much less time and effort than other state-of-the-art defect detection tools.

As mature as Atlas is already, there are several directions for future research. First, its software graph schema needs to be semantically-formalized and abstracted to apply to many programming languages. EnSoft is already pursuing this goal with its new graph schema, eXtensible Common Software Graph (XCSG)[105]. Second, there is an opportunity for the memory footprint and performance of Atlas APIs to be further optimized. Query optimization is a well-studied topic; however, there is the potential to exploit the specific structure of software graphs for speed. Finally, Atlas would be an even more potent program comprehension tool if support were added for on-the-fly graph simplifications. Program subgraphs are often too large for users to quickly comprehend; there is an opportunity to provide simplified views with a “click to drill down into details” paradigm.
6.2 Security Toolbox

How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?

Chapter 4 presents the Security Toolbox, a human-in-the-loop system for detecting novel, sophisticated, domain-specific Android malware. Prior work lacks the flexibility to detect novel malware, or else employs a traditional two-pass approach in which a user must manually review and reject a large number of false alarms produced by the automation phase. The Security Toolbox solves this problem with a human-in-the-loop approach. Our automated smell and property analyzers uncover an initial set of interesting program behaviors, presenting them to the user in a convenient Dashboard view so that they can be systematically-reviewed and annotated. The analyst is able to use these artifacts to formulate domain-specific flaw hypotheses and issue follow-up questions using a set of provided analysis primitives. As a security analyst and primary developer of the Security Toolbox, my contributions include:

- Creation of “smell” and “property” analyzers for interesting program behaviors
- Sophisticated analysis utilities for:
  - Indexing Android XML resources
  - Symbolic value analysis
  - Software graph alignment
  - State-of-the-art type inference
- Experimental validation as a security analyst for APAC experiments

During Phase I of APAC, we employed the Security Toolbox and the program comprehension features of Atlas to audit 77 Android applications, 62 of which contained novel, state-of-the-art malware. We successfully identified the malicious functionality in 57 of the 62 apps, with an average analysis time of only 1.13 hours per app. These results dramatically outperformed a control team using only state-of-the-art commercial tooling, and were best-in-class among
APAC performers. Additionally, we discovered many unintended bugs and vulnerabilities that were not counted in the official data. As tempting as it may be to attribute our success to the intellect of our analysts, we credit our outperformance to our human-in-the-loop malware detection philosophy and symbiotic relationship with the Atlas platform.

There are a number of possible follow-up research directions for the Security Toolbox. First, it is widely-known that most Android malware in the wild is unsophisticated and makes little attempt to hide itself from detection. There is an opportunity to create a fully-automated version of the Security Toolbox for detecting everyday, simplistic, profit-driven malware. Second, improvements can and should be made to further integrate the human analyst into the analysis loop, by prioritizing detected program behaviors and automatically suggesting domain-specific flaw hypotheses. The Security Toolbox could be enhanced with a taxonomy or “cookbook” of related flaw hypotheses in order to assist the user with the process of brainstorming. Finally, the Security Toolbox should fully utilize the summaries of Android produced by FlowMiner to increase its automation potential. Analysis sub-steps whose automation was not previously possible due to partial program analysis should be revisited.

6.3 FlowMiner

*How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?*

Chapter 5 presents FlowMiner, a tool for mining expressive yet compact data flow summaries of Java libraries for partial program analysis. Prior work in this area creates summaries that are too coarse to be used accurately and do not capture the semantics of callbacks. Summaries produced by FlowMiner do capture callbacks, and also retain sufficient detail to be used with context, field, flow, type, and object-sensitivity. Yet, they are compacted to be an order of magnitude smaller than the original program graph of a library. My contributions include:

- Expressive, fine-grained summary flow specifications with sufficient detail for context, field, flow, type, and object-sensitive use
Summary compaction, allowing flow specifications to be losslessly-compact ed with respect to key features

- A novel graph-based summary extraction algorithm to mine elided data flows

- An open-source reference implementation (FlowMiner) of our summary extraction algorithm

- Empirical evaluation of FlowMiner data flow summaries on a handful of the most popular third-party Java libraries.

In our evaluation of FlowMiner for some of the most popular third-party Java libraries, we found that our summary graphs had an average of 26.89% as many vertices and 16.32% as many edges as the original library program graphs. We also empirically demonstrated the importance of fine-granularity—The most closely related work tracks flows at the level of object granularity rather than field granularity, which produces an average of 93.07% false positive field read/write flows.

While data flow summaries of library components are very useful for security-critical analyses, future work should investigate the potential for use of other diverse kinds of library summaries. Examples might include summaries of control flows, error codes or exceptions, permissions used, and domain-specific resource usage. The nature of the summary desired will depend upon the intended subsequent analysis use cases. Finally, FlowMiner should be extended to analyze libraries written in languages other than Java.
BIBLIOGRAPHY


[38] Y. Chen. The c information abstraction system. . ., *IEEE Transactions on*, 16(3), 1990.

27


19


[68] T. Fraser. Automated program analysis for cybersecurity (apac), July 2011. 87


[79] B. Holland, March 2015. 63


[119] V. Rajlich. Intensions are a key to program comprehension. In *ICPC*, pages 1–9, 2009. 10


