AFRL-RI-RS-TR-2015-186

COMPREHENSION-DRIVEN PROGRAM ANALYSIS (CPA) FOR MALWARE DETECTION IN ANDROID PHONES

IOWA STATE UNIVERSITY

JULY 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2015-186 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /	**/ S /**
MARK K. WILLIAMS	WARREN H. DEBANY, JR.
Work Unit Manager	Technical Advisor, Information Exploitation and Operations Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JULY 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2012 – JUN 2015

4. TITLE AND SUBTITLE COMPREHENSION-DRIVEN PROGRAM ANALYSIS (CPA) FOR MALWARE DETECTION IN ANDROID PHONES	**5a. CONTRACT NUMBER** FA8750-12-2-0126
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 61101E
6. AUTHOR(S) Suraj Kothari	**5d. PROJECT NUMBER** APAC
	5e. TASK NUMBER 97
	5f. WORK UNIT NUMBER 76
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Iowa State University 1350 Beardshear Hall Ames, IA 50011-2025	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGB 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-186

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The DARPA APAC program gave us an opportunity to make three important technological advances: (a) A graph database program analysis platform and a graph schema for representing program semantics that together facilitate both automation and human comprehension. (b) Malware analysis techniques and its incorporation in a security toolbox to provide a man-machine analysis system to detect novel, sophisticated Android malware. (c) An innovative library summarization technique and its incorporation in the FlowMiner tool that mines expressive, compact information flow summaries from a library for accurate and scalable partial program analysis. The challenge apps were very useful in evolving our technologies and understanding its limitations. Details of technological advances, our experiences and observations are outlined in this report.

15. SUBJECT TERMS
Android malware, graph paradigm for software analysis, man-machine systems to reason about large software, automated summarization of software libraries

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **MARK K. WILLIAMS**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	33	**19b. TELEPHONE NUMBER** ***(Include area code)*** **N/A**

Table of Contents

List of Figures

List of Tables

1. Executive Summary

Mobile malware detection can be extremely challenging in the presence of cross-cutting control and data dependencies, invisible control switches due to multithreading or event processing. Moreover, inherent ambiguities of an application's intent make it difficult to separate malicious behavior from legitimate functionality. Given that provable automation is not possible in all cases, our proposed novel comprehension-driven graph-based approach enables an iterative refinement process capable of quickly discovering sophisticated malware. We were the top-performing Blue team in Phase I and among the top 3 performers in Phase II. We exceeded Phase I and Phase II BAA goals in terms of analysis time, and also in terms accuracy in Phase I. The Red team, however, has clearly shown that malware detection is still an unsolved problem and more research is needed. The program has brought to the surface some of the hard problems of static analysis and the need for a program comprehension technology to enable humans to develop better hypotheses of potential malware.

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?*
2. *How can a man-machine analysis system detect novel, sophisticated, and domain-specific malware?*
3. *How can expressive, compact information flow summaries be mined from a library for accurate and scalable partial program analysis?*

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 advance Atlas and its graph schema to meet our need. 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 a Shell 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.

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

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, and 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.

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

Android applications, like most modern software, are built on top of reusable libraries. Android provides a massive library, including the entire standard Java library, which applications can call. 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 divergent approaches to solve this problem. As reported in their research paper, Stanford had 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. 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.

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, we found that their results were too coarse to be used accurately or capture flows involved in callbacks. 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 byte-code. 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. Static analysis tools can use this portable summary file to achieve complete and accurate, yet scalable, partial program analysis.

2. Introduction

This work was performed by Iowa State University (ISU) and was issued by the Air Force Research Laboratory (AFRL) under Cooperative Agreement No. FA8750-12-2-0126, Comprehension-Driven Program Analysis (CPA) for Malware Detection in Android Phones. The Defense Advanced Research Project Agency (DARPA) program manager was Tim Frasier and the AFRL Program Manager was Mark Williams. The PI was Dr. Suraj Kothari from ISU and the subcontractors were EnSoft. Corp and North Carolina State University (NCSU participated only during the early part of Phase I).

As described by DARPA [1] this program aims to address the following challenges.

> The Automated Program Analysis for Cybersecurity (APAC) program aims to address the challenge of timely and robust security validation of mobile apps by first defining security properties to be measured against and then developing automated tools to perform the measuring. APAC will draw heavily from the field of formal-methods program analysis (theorem proving, logic and machine proofing) to keep malicious code out of DoD Android-based application marketplaces. APAC will apply recent research breakthroughs in this field in an attempt to scale DoD's program analysis capability to a level never before achieved with an automated solution.
>
> ..
>
> The second challenge APAC aims to address is producing practical, automated tools to demonstrate the cybersecurity properties identified. Successful tools would minimize false alarms, missed detections and the need for human filtering of results to prove properties.

2.1 Team

Our team is composed of three sub teams, each of which brought a unique capability to this project.

Suraj Kothari, the principal investigator, and his team at Iowa State University have a track record of innovative advancements in applications of program analysis.

Jeremias Sauceda, the co-principal investigator and his team at EnSoft have experience in building world-class easy-to-use engineering tools that apply sophisticated algorithms.

In the first year of Phase I, Xuxian Jiang and his team at North Carolina State University provided their expertise in Android malware in the wild.

2.2 Goals and Progress Towards Goals

The Automated Program Analysis for Cybersecurity (APAC) program was designed to find malware in Android phones. The goals as set in the program BAA [2] are listed below.

1. Develop a practical program analysis tool to keep malicious applications written in Java out of the DoD Android-based mobile marketplaces.

2. Conduct novel research that leverages advances in static program analysis to develop innovative capabilities, far beyond existing practices, to detect highly sophisticated malware attacks that the DoD needs to be prepared for.

3. To meet the DoD needs for practical deployment, the proposed tool will surpass the following minimum performance requirements:

   A. False alarms less than 30% in Phase I and less than 5% in Phase II
   B. Missed detection less than 30% in Phase I and less than 5% in Phase II
   C. Manual labor required per mobile app less 160 hours in Phase I and less than 80 hours in Phase II
   D. An analyst with basic knowledge of software development and malicious techniques should be able to operate the tool effectively

When evaluating our success of goal 1, we consider that we were the top performing Blue team in Phase I and among the top 3 performers in Phase II, but on the other hand, a missed detection rate of even 5% may have dire consequences when dealing with DoD level software audits. The Red team has shown that even under somewhat ideal conditions, the problem of detecting malware is far from solved. Additionally even though we performed well within our analysis time goals, the APAC performers received feedback from DoD analysts during PI meetings that human analysis time may need to be decreased further to meet current analysis demands. That being said, our average analysis time was significantly under the proposed analysis time goals proposed in the BAA.

In response to goal 2, our team has published peer-reviewed papers on novel program analysis techniques, which are detailed in the Publications section at the end of this report.

With regard to goal 3A, our team found unintended malware (malicious behavior not purposely crafted by the Red teams, nonetheless found to exist in a challenge app). In Phase I, we found 6 unintended malwares and 35 unintended malwares in Phase II. Our human-in-the-loop process did not produce false alarms in Phase I or II.

Pertaining to goal 3B, we exceeded our Phase I goal of limiting missed detections to 30 % with a missed detection rate of 6.49 %. In Phase II, as a result of the changing nature of the challenge applications our missed detection rate increased to 25% and so we did not meet our Phase II goal of 5 %. The Red team has clearly shown that malware detection is still an unsolved problem and more research is needed. The APAC program has brought to the surface some of the hardest problems of static analysis and the need for new technology to address those problems. Specifically, there is a critical need for new static analysis based program comprehension techniques to enable humans to better hypothesize the candidates for potential malware. In almost all challenge apps in Phase II where we did not find the planted malware, our failure was because of our inability to come up with the right hypothesis. In many of those case we hypothesized and found malware but it was not the one that was planted by the Red Team. Thus, in phase II although we found a lot more unintended malware (35 instances of unintended malware in Phase II as compared to 6 instances in Phase I), our missed detection rate increased was worse in Phase II (25% missed detection rate in Phase II compared to 6.49% in Phase I).

Our 3C goal of analyzing applications under 160 hours per application in Phase I and under 80 hours in Phase II was achieved. In Phase I we averaged 1.13 hours per application and an average of 9.19 hours per application in Phase II. Our analysis time increased between Phase I and Phase II, the bulk of the increase is due to the time that analysts needed for hypothesizing the malware. The time for automated analysis was not the issue. In fact the apps in Phase II were typically much smaller than the apps in Phase I and the scalability of automated analysis was not the issue.

As for goal 3D, we believe we have met this goal. Throughout APAC Phase I and II, ISU has employed undergraduate students to provide feedback and assist with development tasks of the Security Toolbox. And an undergraduate course in software engineering that used Atlas for homework on program analysis showed that users with limited background in software analysis and malware could operate the tool successfully. The Red team continually praised our tool for its usability and maturity in the field.

A further discussion of these results can be found in the Results and Discussions section.

3. Methods, Assumptions, and Procedures

Our research focused on the challenges not addressed, novel and sophisticated malware that unlike the malware reported in the wild, pose significant program analysis challenges. Unlike the other security attacks that are immediately noticeable through their denial of service, malware apps can silently leak sensitive information without revealing themselves.

Detecting sophisticated and novel mobile malware can be extremely challenging in the presence of the following program analysis challenges:

- Non-local data and control dependencies in the malware that cut across several functions and data structures.
- Invisible control switches due to multithreading or event processing - these control switches are not directly visible in static analysis as function calls or as control statements.
- Inherent ambiguities of an application's intent make it difficult to separate malicious behavior from legitimate functionality.

Users of existing defect analysis (not just malware detection) tools employ two passes: (1st pass) the tool works automatically to produce a list of potential problems in the code, (2nd pass) tedious manual inspection to validate the problems. This approach runs into the following difficulties. Without any on-the-fly human intelligence to guide its trajectory, the tool makes wrong or highly conservative decisions resulting in many false negatives and/or positives. Moreover, the results produced by defect analysis tools lack evidence for humans to reason with to confirm or reject the tool findings. The existing 2-pass approach is shown in Figure 1.

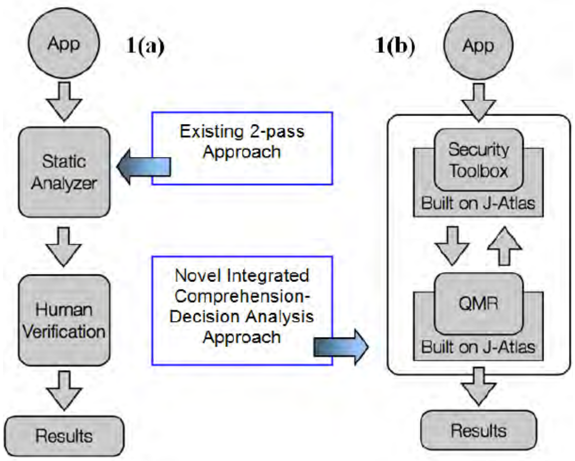


Figure 1 - Existing 2-pass vs. Integrated Comprehension Driven Analysis Approach

A tool by itself cannot deal with the program analysis challenges listed above, so we incorporate human guidance to tweak the trajectory of the tool to improve its precision in detecting malware. A human cannot guide the tool effectively without comprehending the application software. Nor can he comprehend the large and complex application software without an inordinate amount of effort, so we incorporate tool assistance by searching and extracting relevant software artifacts as evidence for the analyst to reason about the application's intent and validate the malware detection results from the tool. In short, we proposed and researched a novel integrated comprehension-driven analysis-based approach.

As shown in Figure 1, we have a tightly coupled human-in-loop approach with iterative refinement as opposed to the existing 2-pass approach. Our approach succeeded reasonably well because it provided a viable alternative to address sophistications of attacks that make program analysis difficult.

We leveraged the Query-Model-Refine (QMR) framework developed by EnSoft; it provides the tool mechanics necessary for our tightly coupled human-in-loop approach. Like the database language SQL, the framework incorporates a composable query language that can be used either interactively or embedded as a Java program to create programs to analyze programs. Because of the query language, the user is freed from the lower-level details of static analysis and enabled to focus efforts on the malware detection strategies at higher-level semantics. The “model” and “refine” capabilities work in conjunction with the query language to facilitate human comprehension by creating effective abstractions of large software. We developed Atlas, a QMR platform for Java.

Another important aspect of our approach was to develop a tool that is evolution-friendly and highly usable, i.e., it is fairly easy to refine and extend its malware detection capabilities without requiring expertise in building static analysis tools. This is the case because our approach amounts to having the malware detection capabilities incorporated as a toolbox built on top of Atlas. The low-level details of static analysis reside inside Atlas, and the malware detection capability resides inside the toolbox as compact analysis programs using Atlas queries. Refining and extending the existing detection capabilities as well as creating entirely new capabilities is relatively easy because it can be done through query-enabled analysis programs. The underlying design philosophy is similar to environments like Matlab where the heavy lifting is done behind the scenes, making it much easier for the user to develop domain-specific programs. Since creating a complete list of 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. An evolution-friendly technology provides a cost effective path for DoD to maintain state-of-the-art in malware detection for years to come.

3.1 Technological Advances

The DARPA APAC program gave us an opportunity to make three important technological advances that we will describe here.

3.1.1 Graph Schema and Program Analysis Platform

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

We have made significant advances to answer this key question. We came up with a graph schema, called the eXtensible Common Software Graph (XCSG) to provide an attributed directed graph as a common medium to express rich structural and behavioral semantics of programs in Java, Java byte code, C and C++. We advanced Atlas as the graph database platform to write program analyzers based on the XCSG schema. We have advanced the Atlas platform that enables one to write software analysis verification and transformation programs in minutes or hours that otherwise would take days or months.

Atlas parses C, C++, Java, and Java bytecode to capture complex program semantics in a graph database. It provides APIs to mine, traverse, and transform the graph database. Atlas APIs and program graph visualization capabilities enable quick prototyping of tools to experiment with and advance fundamental techniques to reason about complex problems of large software. Atlas is free for academic use. The XCSG schema is available online.

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 our ICSE 2014 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.

Demo Video: http://youtu.be/cZOWIJ-IO0k

3.1.1.1 Graphs and Queries

Graphs are a natural way to represent programs and program analysis results, where nodes typically correspond to entities such as methods and variables, and edges correspond to relationships such as control or data flow. The Atlas database extracts and stores rich program semantics as a unified graph representation that includes structural relationships (types, methods, fields, etc.) and control and data flows derived by conservative analyses as the base knowledge for writing refined path, object, context, and field sensitive analyses.

Encoding the program semantics in a unified graph has the advantage of lending itself to composable analyses. The result of a program analysis is usually another graph, which can then be used as the input to the next analysis. In Atlas, graphs can also be displayed, but the visualization is not necessarily the end; the nodes and edges can be selected, and used as inputs for the next iteration of analysis.

Many program analysis questions can be encoded as reachability queries, and so Atlas provides a query language to make these common queries easy to write. Writing queries also involves knowing the graph's schema, or how the program and analysis information is encoded. The Atlas schema, called the eXtensible Common Software Graph (XCSG), will be discussed later.

3.1.1.2 Human Interaction

An unprecedented human interaction capability, far beyond any other existing program analysis tool, to reason about complex problems of software is enabled by: (a) a capability to visualize and interact with large program graphs in a way that fosters human comprehension of complex program semantics, (b) a correspondence with the code for program artifacts and the corresponding graphs depicting relationships between those artifacts - a correspondence that enables scalable navigation through large code, (c) a query interpreter shell that enables composition of powerful queries to mine complex cross-cutting program semantics and its visualization.

From our experiences during the first few engagements analyzing Android apps, we found that many queries were variations on data flow queries. To help accelerate the iterative analysis process, "Atlas Smart Views" were introduced to help reduce common queries to a point-and-click operation, wherein the analyst selects an analyzer from a drop-down menu and that analyzer is applied automatically and the corresponding result is shown whenever the analyst clicks on an appropriate source code entity. For example, the analyst selects the "call graph" analyzer, clicks on a method invocation in the source code being viewed, instantly the corresponding call graph is shown. As another example, the analyst selects the "data flow graph" analyzer, clicks on a parameter in for the method invocation in the source code being viewed, instantly the corresponding data flow graph is shown. New analyzers can be added to the "Atlas Smart Views." We added Android-specific analyzers.

3.1.1.3 Extensibility

Atlas serves as a platform to build domain-specific toolboxes such as the Security Toolbox we built for the APAC project. The Security Toolbox incorporates Android semantics by extending the graph database. For example, the graph database is extended to include the Graphical User Interface (GUI) semantics derived from the Android XML files. The extension must be designed to follow the eXtensible Common Software Graph (XCSG) schema.

The XCSG schema defines a semantically rich graph representation of software (i.e. source code or binaries) to support program applications such as mining software for patterns, malware and defect detection, building static analysis tools, and code comprehension. XCSG is based on the eXtensible Common Intermediate Language (XCIL) developed by Kothari and his team for the DARPA Software Enabled Control (SEC) program [3].

Like XCIL, XCSG has semantically precise definitions for program artifacts to enable a harmonious representation of software written in different languages. Without precise semantics, analysis tools can easily develop a language bias that leads to incorrect processing of other languages, especially while analyzing software written in multiple languages. For example, the keyword “static” in C and Java have overlapping but incompatible uses, which XCSG disambiguates. XCSG improves upon XCIL by tailoring it for a graph database, and by encompassing representations of analysis results such as control flow and data flow graphs.

3.1.2 Android Security Toolbox

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

Our research to address this question led to several interesting innovations. Using the program analysis platform Atlas, we have incorporated these innovations in a domain-specific toolbox, called the Android Security Toolbox for detecting malware in Android Apps. The Security Toolbox is designed with following goals:

1. 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.
2. Incorporate the rich and complex Android semantics of API permissions, components such as Activities, Services, Content providers, Broadcast receivers, and XML resource files.

3. Provide a decoupled architecture for an evolution and user-friendly malware detection tool. 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.

In our ICSE 2015 paper, we describe the design and use of the Android Security Toolbox.

Video: http://youtu.be/WhcoAX3HiNU

3.1.2.1 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. 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.

3.1.2.2 Analyzers

An analyzer conforms to specifications defined by the Security Toolbox. Specifically an analyzer encapsulates a name, description, set of analysis assumptions, and the analysis program to be executed. The programs written in Java invoke Atlas APIs to access the information in the graph database and typically its purpose is to check one or more security properties. The result of the analyzer, called "envelope," is an Atlas graph that captures the program semantics relevant to the property. The graph can be empty if the property is undetected and non-empty if the security property is detected. The graph may be shown to be interacted with by the human analyst or used as input into another analysis. For instance a confidentiality analyzer might first do a cheap insensitive taint leak (reachability test) between an automatically detected source and sink pair (e.g. a flow from the SIM card number to the Internet). If the resulting graph is non-empty but very large we could pass the graph to a more expensive sensitive (call, object, type, flow, etc.) taint leak analysis to prune false positives from the graph. The toolbox refers to this type of recommended analyzer chaining as "continuations".

Analyzers have been subdivided into five categories: properties, smells, confidentiality, integrity, and availability. 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 Application Programming Interface (API) method. The Security Toolbox takes the conservative approach of making smells trend towards tighter heuristics that only report with high confidence. 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. Sources, sinks, mutables, and resources are inputs to the CIA Analyzers.

The analyzers in the Security Toolbox are general-purpose analyzers. We can only add analyzers and input models (e.g. sources and sinks) that can be written *a priori*. Domain specific knowledge such as the fact that the result of a certain sensitive calculation should be treated as a source of information for confidentiality leaks still needs to be determined at runtime by a human analyst. Once the new confidentiality source is discovered however, it is a trivial task to run the various taint leak detection analyzers with the new information.

3.1.2.3 Indexers

Since Android makes extensive use of extensible markup language (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.

In another use case for custom indexers, Atlas provides a conservative open-world approximation to resolve dynamic dispatches, but leaves the necessary raw information for type-sensitive answers to be computed. This arrangement is ideal because it allows the Security Toolbox to explicitly choose a desired speed vs. accuracy tradeoff that suits the situation. For conservative dynamic dispatches, the Security Toolbox implements a type inference indexer that reduces the set of conservative edges by tagging edges that it can show are likely runtime behaviors. To enable object sensitivity, Atlas provides unique object instance ids, which can be used to maintain call site histories and perform sensitive data flow traversals.

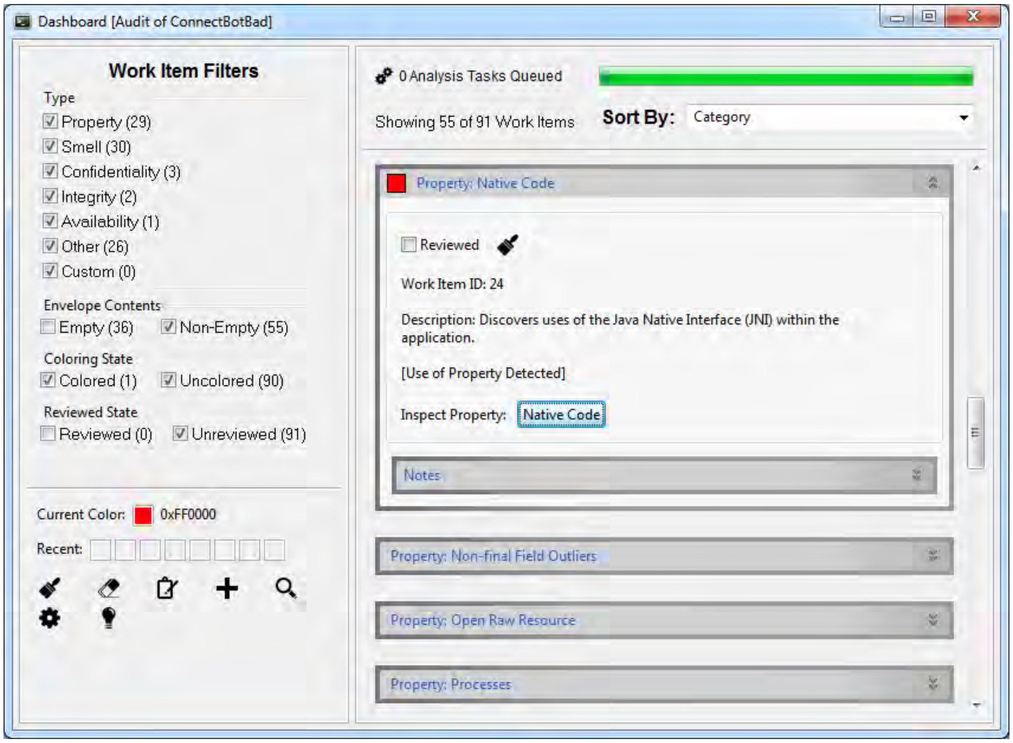
3.1.2.4 Dashboard

The Dashboard (shown in Figure 2) 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.

Since some analyses depend on the results of another analysis, such as type inference or resource indexing, and other analyses do not have prerequisite analyses some results can be computed in parallel. The Dashboard builds a precedence graph and prioritizes the analyses that enable the maximum amount of parallelization.

Results can be inspected as soon as they are available, and the analyst can sort and filter results by type, contents, and state. State can either be reviewed or un-reviewed and colored or uncolored. An analysis result can optionally be assigned a color. The color is simply an Atlas tag for all of the graph elements inside of the work item. Additional elements outside of the original result set can be colored with the same coloring to manually add elements to the work item.

Each analysis work item has a note-taking field that allows the analyst to record time-stamped notes. Using a utility developed for the Security Toolbox called AuditMon, the Dashboard can record selection events directly into the Atlas program graph. This information can later be used to review the audit.



Dashboard [Audit of ConnectBotBad]

Work Item Filters

Type

☑ Property (29)
☑ Smell (30)
☑ Confidentiality (3)
☑ Integrity (2)
☑ Availability (1)
☑ Other (26)
☑ Custom (0)

Envelope Contents

□ Empty (36) ☑ Non-Empty (55)

Coloring State

☑ Colored (1) ☑ Uncolored (90)

Reviewed State

□ Reviewed (0) ☑ Unreviewed (91)

Current Color: 🟥 0xFF0000

Recent:

0 Analysis Tasks Queued

Showing 55 of 91 Work Items **Sort By:** Category

🟥 Property: Native Code

□ Reviewed

Work Item ID: 24

Description: Discovers uses of the Java Native Interface (JNI) within the application.

[Use of Property Detected]

Inspect Property: Native Code

Notes

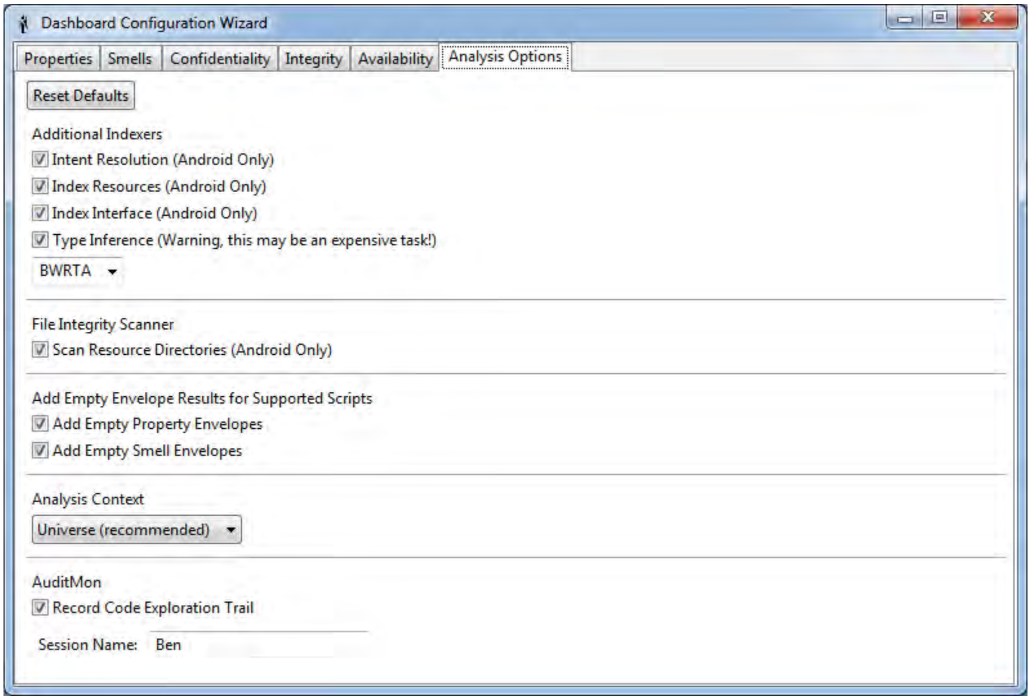
Property: Non-final Field Outliers

Property: Open Raw Resource

Property: Processes

Figure 2 - Dashboard

Since analyses contain configurations options, such as selecting trade offs on accuracy vs. time, the Dashboard provides a configuration wizard (Figure 3) to enable or disable individual analysis programs or select analysis strategies.



Dashboard Configuration Wizard

Additional Indexers

☑ Intent Resolution (Android Only)
☑ Index Resources (Android Only)
☑ Index Interface (Android Only)
☑ Type Inference (Warning, this may be an expensive task!)

BWRTA

File Integrity Scanner

☑ Scan Resource Directories (Android Only)

Add Empty Envelope Results for Supported Scripts

☑ Add Empty Property Envelopes
☑ Add Empty Smell Envelopes

Analysis Context

Universe (recommended)

AuditMon

☑ Record Code Exploration Trail

Session Name: Ben

Figure 3 - Dashboard Wizard

3.1.3 FlowMiner

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

Our research to address this question has led a technological advance for analyzing programs that depend on software libraries.

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.

Supported by the additional seed funding in 2014, we started developing FlowMiner, a tool to mine expressive data flow summaries from Java library binaries to enable complete and accurate partial program analysis. This work was recently completed and it is a part of Tom Deering's Ph.D. research. As far as we know, 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 that 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 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 by on some of the most 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.

3.1.3.1 Balancing 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. 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 some 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.

3.1.3.2 Validation

We have validated 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. 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.

3.1.3.3 Open Source

We provide FlowMiner, an open-source reference implementation of our algorithms that extracts summaries given the source or byte code of a library and exports them to a portable, tool-agnostic format. The FlowMiner research is described in more detail in Tom Deering's Ph.D. thesis and a paper based on this work will be submitted for publication. The tool is available at the following site: http://powerofpi.github.io/FlowMiner/

4. Results and Discussions

4.1 Summary

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. A control team was employed beginning with engagement 1C to use current state of the art tools to audit the apps alongside Blue team performers. Our process correctly identified malice in 57 (91.94 %) of the malicious apps and correctly classified 66 (85.71 %) apps as malicious or benign. We found 6 unintended malicious behaviors, and missed malware in only 5 (6.49 %) of the apps consistently beating the control team. We completed Phase I as the top performing Blue team.

At the end of Phase I (starting with engagement 2B) the nature of the malware in the challenge applications began to change from what was primarily seen in Phase I. The malware became much more difficult and we saw the performance of all teams including the control team reduced. Over the course of Phase II, our team audited 28 challenge applications developed by the Red team, of which 25 contained novel malware. Our process correctly identified malice in 18 (72 %) of the malicious apps and correctly classified 25 (89.29 %) apps as malicious or benign. We found 35 unintended malicious behaviors, and missed malware in 7 (25 %) of the apps beating the control, but not by as much in Phase I.

According to metrics provided at the PI meetings, we determined that we maintained our position in Phase II among the top 3 performing teams and weren't far behind the leading team.

4.2 Analyst Backgrounds

The participating analysts included throughout the engagements included graduate students, a staff member from the ISU research group (the staff member has an MS in Computer Engineering), undergraduate research assistants, and software engineers from EnSoft. During engagement 1A there were 6 analysts, but for every engagement thereafter there were only 2-4 analysts working on challenge application audits per engagement.

4.3 Analysis Process

Before the actual analysis, one person spent 1-2 hours to survey application size and prepare all apps as Eclipse projects to be ready to audit. After preprocessing apps, two analysts were assigned to each app. Each analyst worked independently and did not have access to the work of other analysts. A coordinator who oversaw the process but did not audit apps reviewed the results of the analysis done by each analyst. An additional analyst was assigned to audit an app by the coordinator if the results from the original two analysts were deemed to be in conflict or inconclusive. All analysts worked independently without access to the results of analysis done by other analysts. A single report for each application was chosen at the end of the experiment to submit in the collection of final reports.

An emulator was used to verify the malicious behavior after the malware and triggers were discovered and reported by the analyst. Emulation was not used to detect malware; it was done strictly to observe the malicious behavior predicted by our analysis. In some cases, we could not observe the malicious behavior, but we have still reported the application as malware because we were certain of the presence of a malicious payload based on our analysis.

4.4 Engagement Results

It is difficult to interpret the results of the engagements for many reasons. The experimental setup of each engagement evolved as engagements progressed. A control team was not added until engagement 1C and the difficulty of the malware increased drastically starting with engagement 2A. In early Phase I some malicious applications contained more than one malware, while later engagements reduced the malicious surface area by limiting the malware to a single malicious behavior. In engagement 1B we were tasked with finding all malwares. Starting with engagement 2B we were given access to a human oracle that confirmed malice when presented with the correct evidence. While the main idea of the oracle persisted to the end of Phase II, the oracles official response policy changed over engagements. The oracle was intended to stop teams from "short circuiting" on unintended malice. Engagements 1C, 2B, and 3B were onsite and had limits on the number of analysts (4, 3, and 3 respectively) and limits on time (8, 4, and 5 hours respectively). A few select applications in engagement 3B and 4A were "updates" to existing applications in previous engagements so that previous knowledge could be drawn upon for during the audits. Beginning with engagement 1B, application support libraries came "packed" into the source instead of as dependent JAR libraries and were considered in scope of the audit. During engagement 4A all apps were known to be malicious, as opposed to previous engagements where the classification of the application was unknown.

Table 1 - Challenge Application Distribution

Engagement	# Apps	# Malicious Apps	# Benign Apps	# Malwares
1A	26	21	5	29
1B	16	14	2	16
1C	10	7	3	7
2A	16	13	3	15
2B	9	7	2	8
3A	10	8	2	8
3B	4	3	1	3
4A	14	14	0	14
Phase I	**77**	**62**	**15**	**75**
Phase II	**28**	**25**	**3**	**25**
Total	**105**	**87**	**18**	**100**

Not counting a dry run practice engagement, which had 8 challenge applications, our team completed 105 application audits. Of the 105 applications, 87 were officially classified as malware containing 100 individual officially malicious behaviors.

Table 2 - Challenge Application Metrics

Engagement	Average Java Files	Average Java LOC	Average XML Files	Average XML LOC	Average Other Files	Average Other LOC	Average Comments	Average Libraries
1A	65.65	7128.08	24.46	1778.77	4.08	922.77	3145.46	0.81
1B	80.5	8841.44	49.31	2233.25	0.56	83.94	4498.31	0.5
1C	28	3417.8	27.1	1965.3	0	0	2548.4	0
2A	50.44	6310.19	18.44	400.75	0	0	2095.81	0
2B	70.56	9144.33	17.89	530.56	0	0	9807.56	0
3A	53.8	5743.7	24.3	813.8	0.1	25.5	2686.5	0
3B	18	2083.5	16.75	383	1.75	62.25	194.25	0
4A	89.86	8819.29	17.79	604.79	0	0	4190.5	0
Phase I	**66.71**	**6758.61**	**19.96**	**647.75**	**0.29**	**18**	**3082.46**	**0**
Phase II	**61.26**	**7067.96**	**27.95**	**1465.19**	**1.49**	**329.03**	**3909.61**	**0.38**
Total	**62.71**	**6985.47**	**25.82**	**1247.21**	**1.17**	**246.09**	**3689.04**	**0.28**

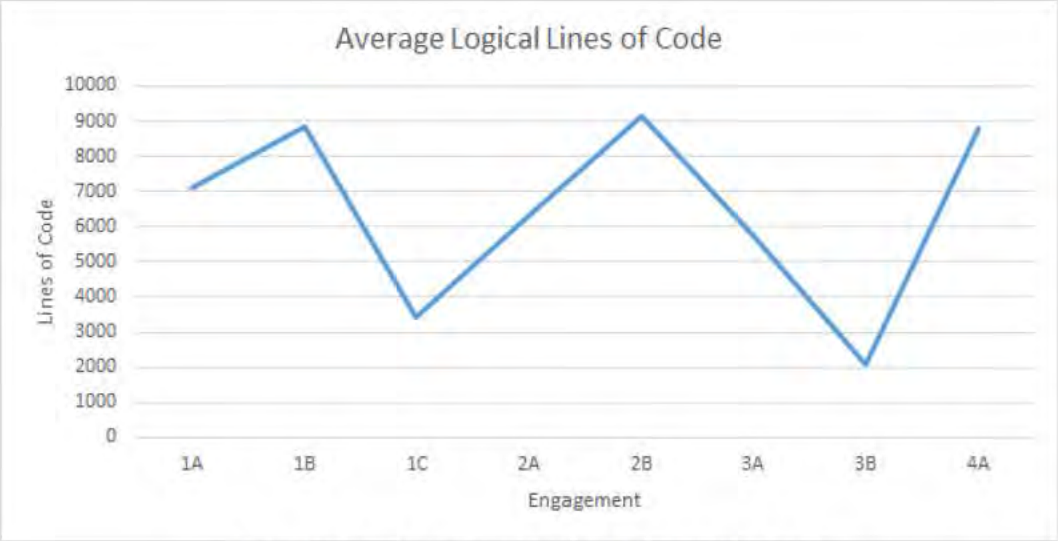


Figure 4 - Average Logical Lines of Code

Table 3 - Challenge Application Detection Rates

Engagement	Correct Classification	Correctly Identified Malice	Missed Detections	Unintended Malware
1A	88.46 %	95.24 %	3.85 %	2 issues
1B	87.5 %	100 %	0 %	2 issues
1C	90 %	100 %	0 %	1 issue
2A	81.25 %	84.62 %	12.5 %	1 issue
2B	77.78 %	71.43 %	22.22 %	0 issues
3A	80 %	62.5 %	30 %	4 issues
3B	100 %	0 %	75 %	10 issues
4A	92.86 %	92.86 %	7.14 %	21 issues
Phase I	**85.71 %**	**91.94 %**	**6.49 %**	**6 issues**
Phase II	**89.29 %**	**72 %**	**25 %**	**35 issues**
Over Both	**86.67%**	**86.31 %**	**11.43 %**	**41 issues**

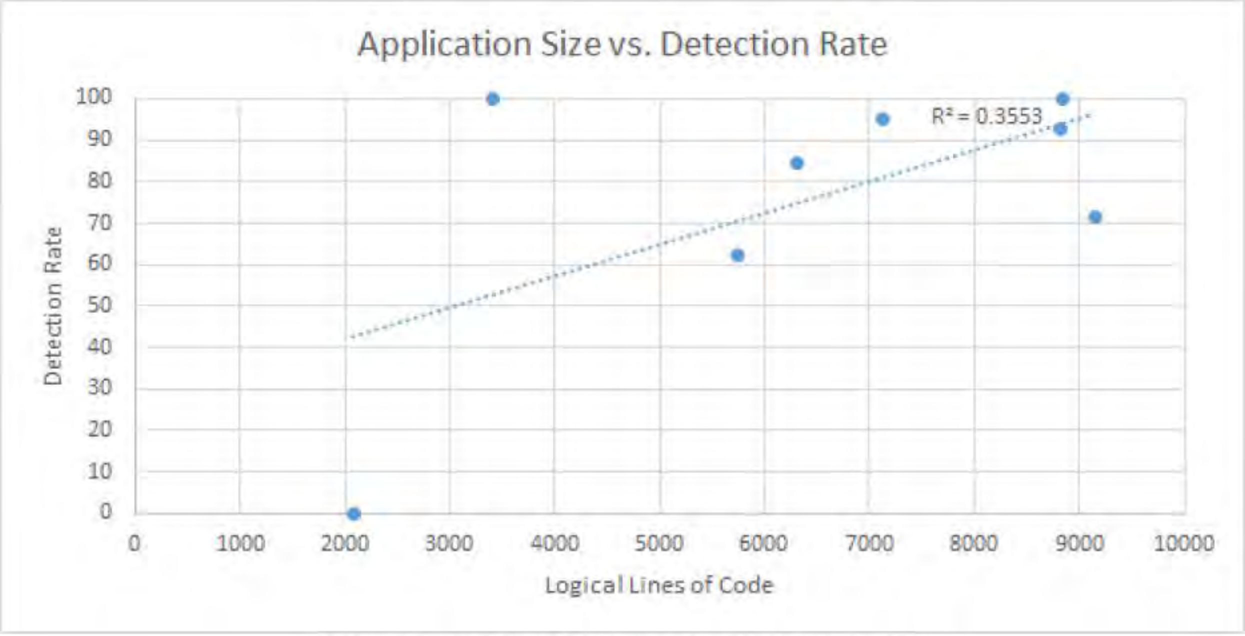


Figure 5 - Application Size vs. Detection Rate

Table 4 - Challenge Application Analysis Time

Engagement	Total Analysis Time	Average Analysis Time
1A	29.72 hours	1.14 hours
1B	13.44 hours	0.84 hours
1C	8.24 hours	0.82 hours
2A	26.35 hours	1.65 hours
2B	9.03 hours	1.00 hours
3A	83.98 hours	8.40 hours
3B	13.01 hours	3.25 hours
4A	160.20 hours	11.44 hours
Phase I	***86.78 hours***	***1.12 hours***
Phase II	***257.19 hours***	***9.19 hours***
Over Both	***343.98 hours***	***3.28 hours***

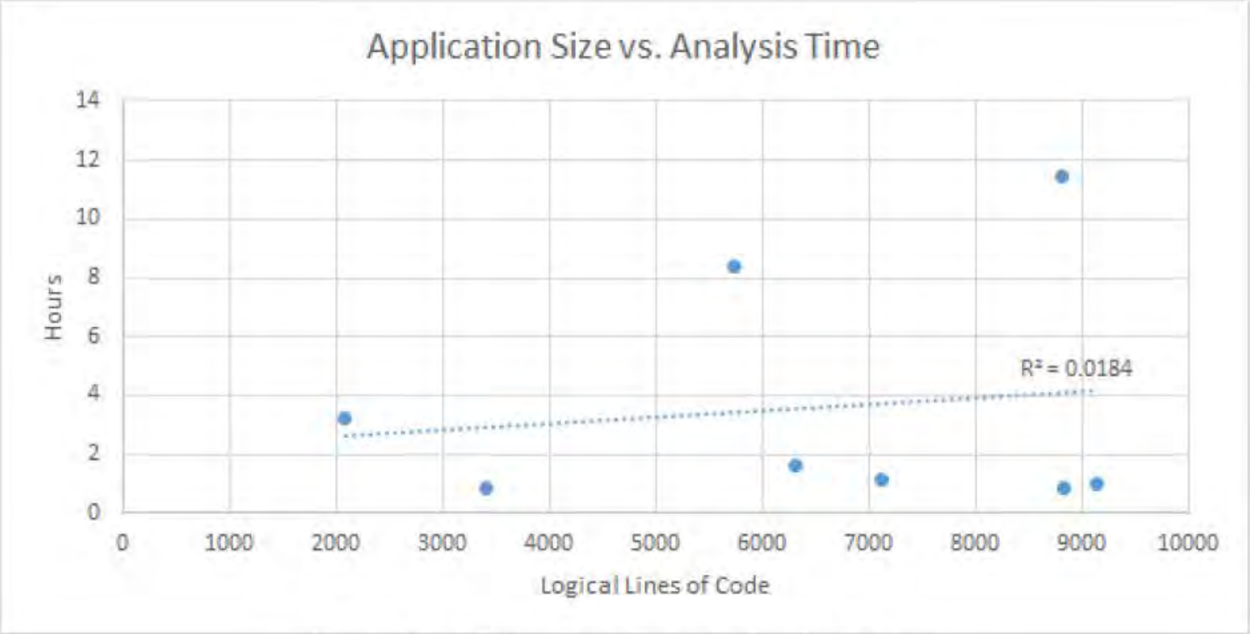


Figure 6 - Application Size vs. Analysis Time

4.5 Engagement Observations

In engagements 1A the Red team created new small applications, less than 500 lines of code (LOC), and utilized existing open source applications ranging from 2K LOC to 60K LOC. The malice focused on breaking automation techniques. The general consensus was that the smaller applications were small enough to manually read to discover malware.

After engagement 1C all applications had a minimum of 1000 LOC. As we expected, our performance with respect to analysis time and correct detection of the intended malware was not correlated with the size of the application (see Figure 5 and Figure 6). This confirms our belief that the difficulty in finding the malware has less to do with the size of the code and more to do with the characteristics of the malware itself. For instance a malware that connects to the Internet and uses a class loader to execute code that it downloads when the application starts stands out like a sore thumb in an application that primarily works offline regardless of the size of the application. If the malware is blended with legitimate functionality and uses existing control flow and data flows for malicious purposes the task of detection becomes much more difficult even for applications of a few hundred lines of code. Engagements in Phase II tended to increase the difficulty by creating subtle domain specific malware that could not be modeled *a priori* and thus coming up with a good malware hypothesis became the bottleneck.

While the applications in 1A through 1C contained several modified open source applications, starting with engagement 2A the applications were developed from scratch. This simulated an adversary that was intimately familiar with the environment the source code. As an experiment the Red team hired an intern who produced large, complex, but poorly written application to confuse the human auditor. In other cases large sections of dead, broken, or unfinished code were intentionally added to increase the difficulty of discovering malicious behaviors. In such cases, the difficulty increased primarily due to poor code quality and not the size of the code.

The general complexity of the base applications also increased in later engagements. While applications in early engagements included apps such as a countdown timer, bible, screensaver, battery monitor, calculator, and backup utilities, later engagements included apps such as an AI Short Message Service (SMS) bot, a network scanner with custom programming language and interpreter shell, device administration utilities, and a device tracking system with a signature based firewall to prevent abuse. These applications took more human effort to understand and hypothesis potentially malicious behaviors.

Early engagements were almost entirely confidentiality based, which made it easy to hypothesize malware and analyze code for its presence. The types of malware that we failed to detect in Phase I were because of the difficult of generating malware hypothesis as such as malware that wasted battery by using an inefficient algorithm or an infinite loop in a background thread or issues that could have been detected with functional testing such a backup application that used a regular expression to incorrectly backup a Chinese phone number (flipping the 5 and 6 digits).

Near the end of Phase I we saw applications that maliciously collaborated with other applications. We also observed the difficulty of dynamic analysis increasing, presumably to thwart teams utilizing dynamic analysis approaches. This increased difficulty made the “trigger analysis” much more difficult because the malware would occur in a set of very precise conditions such as the 99th time an application was launched or after a menu 3 levels deep was selected on one screen and another menu was selected 3 levels deep on another screen. If one part of the malware was discovered, it was still difficult to figure out how that part could be triggered by another action or state in some other part of the application. In many of the later challenge applications, the malware itself was external to the application and what was left to discover in the application was more of a vulnerability that could be leveraged by a malicious interaction with the application. To make it more difficult these interactions tended to happen over custom protocols for which we only had the client libraries or interactions with a web server that was not provided during the audit, making it much more difficult to realize the big picture without additional time investments during the audit.

We saw our analysis time increase from an average of approximately 1 hour per application in Phase I to approximately 9 hours per application in Phase II. This became an issue during onsite engagements 2B and 3B where our maximum analysis time was capped at 4 and 5 hours respectively, resulting in poor performance. All Blue teams together could only discover two malwares in Engagement 3B.

Especially in engagement 4A, “decoy malwares” were placed near the official malware. These decoy malware looked and felt malicious to a human, but were somehow subtly broken or deactivated (but not always) so as to misdirect the human analyst’s suspicion down the wrong trail. These applications, while challenging, were not realistic in our opinion because malware authors tend not to intentionally draw suspicion to themselves.

5. Tool Releases

The Atlas platform enables the creation of much more powerful analysis tools than would be possible starting from scratch and represents years of research and development in practical, highly scalable techniques for software analysis.

The Security Toolbox and the Flow Analysis Toolbox are furnished to the Government with unlimited rights. At the end of the project EnSoft will furnish the Government with a perpetual organization-wide license of the version of Atlas necessary to run the final Security

Toolbox and Flow Analysis Toolbox for the sole purpose of using the Security Toolbox and the Flow Analysis Toolbox at no additional cost. In addition, at the end of the project, EnSoft will furnish the Government one year of complimentary maintenance, which includes EnSoft's world-class support and all updates to Atlas during the maintenance period.

Any and all modifications to the Atlas platform remain the sole property of EnSoft. All licenses of Atlas that will be furnished are for binaries of Atlas. EnSoft will provide binaries to non-Government parties including ISU and performers in other technical areas as required. EnSoft will provide source and binaries to the Government. This source code is licensed to the government for the sole purpose of building Atlas for use with the Security Toolbox and the Flow Analysis Toolbox. This source code may not be disclosed to non-Government parties and remains the sole property of EnSoft.

Information about the commercial off the shelf (COTS) version of Atlas is available on the EnSoft webpage [5]. Information about the XCSG schema and tutorials for learning Atlas are available online [6]. Several components of the Security Toolbox have been extracted into smaller general-purpose toolbox plugins and released under the MIT License [7].

6. Concluding Remarks

The DARPA APAC program gave us an opportunity to make three important technological advances:

1. A graph database program analysis platform and a graph schema for representing program semantics that together facilitate both automation and human comprehension.
2. Malware analysis techniques and their incorporation in a security toolbox to provide man-machine analysis system to detect novel, sophisticated Android malware.
3. An innovative technique to summarize large software libraries and its incorporation in the FlowMiner tool that mines expressive, compact information flow summaries from a library for accurate and scalable partial program analysis.

These technological advances have enabled us to perform well in challenge engagements. In almost every engagement our team’s analysis times were lower than those of all other performers. We attribute the speed and accuracy of our analysis to the maturity of our tools and their capabilities to quickly traverse and answer complex queries about very large code. We believe that this is a highly relevant capability of any malware detection tool. We were continually complemented on the maturity and usability of our tool by the Red teams. We believe our tool capabilities matured significantly as a result of the engagements, but since we cannot quantitatively define the difficulty of malware it is difficult to prove.

The APAC Board Area Announcement (BAA) listed developing practical program analysis tools to keep malware out of Department of Defense (DoD) app stores as a primary goal for the program. To determine tool relevance the challenge app reports requested written explanations of how the tooling was relevant to solve the challenge, but in many cases a tool's relevancy would be better demonstrated through a video or live demo at program meetings. Our team made strong efforts to provide demonstrations of our tool at PI meetings and to interested parties, but several teams did not.

The Red team was able to craft very domain specific malicious vulnerabilities, such as a custom protocol that when combined with another instance of itself causes a broadcast storm of messages (SMSBot from engagement 2B). The crucial difficulty lies in generating a good malware hypothesis. Coming up with the right theorem is itself the difficult part!

In the beginning our team was the only team that proposed a human-in-the-loop process, but it seems the teams that proposed fully automated processes all underestimated the importance of the human role and adopted similar strategies to our team's proposal as the program progressed. In our opinion, significant new research is warranted to enable the human analyst generate domain-specific malware hypotheses.

7. Publications

1. Security Toolbox for Detecting Novel and Sophisticated Android Malware. Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, Nikhil Ranade. *International Conference on Software Engineering*, 2015.

2. A "Human-in-the-loop" Approach for Resolving Complex Software Anomalies. Suresh Kothari, Akshay Deepak, Ahmed Tamrawi, Benjamin Holland, Sandeep Krishnan. I*EEE International Conference on Systems, Man, and Cybernetics*, 2014.

3. Atlas: A New Way to Explore Software, Build Analysis Tools. Tom Deering, Suresh Kothari, Jeremias Sauceda, Jon Mathews. *International Conference on Software Engineering*, 2014.

4. Multi-faceted Practical Modeling Education for Software Engineering. Suresh Kothari, Jeremias Sauceda. ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, 2013.

8. References

1. DARPA APAC Program. http://www.darpa.mil/Our_Work/I2O/Programs/Automated_Program_Analysis_for_Cybersecurity_(APAC).aspx, 2015.

2. BAA 11-63. *Automated Program Analysis for Cybersecurity*. DARPA Information Technology Office, 2011.

3. BAA 99-08. *Software-Enabled Control.* DARPA Information Technology Office, 1999.

4. "Common Attack Pattern Enumeration and Classification." *CAPEC*. MITRE, Web. 20 Feb. 2015. <https://capec.mitre.org>.

5. "Atlas." *EnSoft*. EnSoft Corp., Web. 20 Feb. 2015. <http://www.ensoftcorp.com/atlas>.

6. "Main Page." *AtlasWiki*. EnSoft Corp., Web. 20 Feb. 2015. <http://ensoftatlas.com>.

7. "EnSoft Corp." *GitHub*. Web. 20 Feb. 2015. <https://github.com/EnSoftCorp>.

9. List of Acronyms

AFRL	Air Force Research Laboratory
APAC	Automated Program Analysis for Cybersecurity
API	Application Programming Interface
AST	Abstract Syntax Tree
BAA	Broad Area Announcement
CIA	Confidentiality, Integrity, and Availability
COTS	Commercial Off The Shelf
CPA	Comprehension-Driven Program Analysis
DARPA	Defense Advanced Research Projects Agency
DoD	Department of Defense
GUI	Graphic User Interface
ISU	Iowa State University
LoC	Lines of Code
QMR	Query-Model-Refine
SEC	Software Enabled Control
SMS	Short Message Service
XCIL	eXtensible Common Intermediate Language
XCSG	eXtensible Common Software Graph
XML	eXtensible Markup Language