Mobile Malvertising Detection Using Finite State Machines, Dynamic Analysis, and Static Analysis Techniques

by

Sean Patrick Sanders

Dissertation Proposal submitted to the

faculty of the Graduate School of

the University at Buffalo, The State University of New York in partial fulfillment of the requirements for the

degree of

Doctorate

in the Department of Computer Science

Copyright by

Sean Patrick Sanders All Rights Reserved

**Abstract**

Over 3,739 apps are published daily on the Google Play store [(Sharma, 2019)](https://www.zotero.org/google-docs/?2UdIHS). A handful of the applications contain advertisement malware, referred to as malvertising. As a result, Android advertisement malware has been a growing multi-billion-dollar problem. It constantly assaults many major advertising libraries (e.g., Google, Facebook, and Amazon), but we believe we have found an effective strategy for addressing malware incursions.

Practitioners and researchers have been concerned with the malicious intent and activities of the various families of advertising malware. These activities include click fraud creating an impression without user interaction. Another type of fraudulent activity occurs when a phony advertisement is displayed without the permission of the application developer and the mobile ad library developers. Another common and destructive problem occurs when the malware hides the ad, and the advertiser still has to pay for the ad to be displayed. In effect, the ad is invisible, and the user is unaware of the ad's presence, but there is still a monetary transaction.

Numerous approaches and static and dynamic analysis techniques have been used to combat advertisement malware. These include numerous machine learning techniques, static and dynamic analysis tools, and many database detection schemes, such as MalNet [(Freitas et al., 2021)](https://www.zotero.org/google-docs/?hcJZty).

We assert that advertising malware can be effectively detected through dynamic and static analysis techniques using the Soot compiler framework and finite state machine analysis (our framework is called MADScanner). Our research aims to detect the various types of malicious malware using the Soot compiler framework and finite state machines. This is a holistic approach to combating malvertising because the proposed framework combines finite-state machines, compiler injection, and networking analysis. Thus, this is a state-of-the-art approach to combating advertising malware.

**Contents**

[**Introduction 7**](#_uj5zjs3cx1j4)

[1. Research Questions 12](#_9cpm9z25jfxx)

[**Literature Review and Related Work 16**](#_vpfsq8qm5tng)

[Finite State Machines 16](#_bkaw91y7isq5)

[Forensic Tools 18](#_if9pbzfthwmf)

[Compilers 19](#_yc1bozf65vkn)

[Control Flow Analysis 20](#_3yeikwlm8vb3)

[Android Compiler Injection 20](#_dik96046lvfp)

[Security of Android Applications and the Soot Framework 21](#_pcdi3immbg2f)

[Malware Storage/Detection 22](#_1wnla4guqxwu)

[Machine Learning 25](#_q55r2euwiwz2)

[Advertisement Types 29](#_30z9iawuovhh)

[Ethereum Framework 30](#_wjmelq2ywjy4)

[Ethereum Infrastructure 31](#_ox4lwp4yj054)

[Android Smart Contract Communication 34](#_6phr2ha4o2eg)

[Sensitive Data Tracking 34](#_fd4bmtagqspm)

[**Implementation Framework 35**](#_u1f6b0lt28vn)

[Soot Framework 35](#_z1l6nlhanwuo)

[Soot Transformation 37](#_keb59iun2erc)

[Direct Translation With Stack Interpretation 37](#_4n4ew92i891r)

[Direct Translation With Stack Height 38](#_5wluw6ds4dvf)

[Split Locals 39](#_u1dh5i7uwaop)

[Type Locals 39](#_k2f7fcl9dds)

[Cleanup 40](#_11sulcmc1ww2)

[Obfuscation 40](#_fdwsgcsuq6vk)

[Hypotheses 41](#_hm3gwos01iln)

[**Work Accomplished Related to the Research Questions and Implementation Details 44**](#_ekukoy65j9jx)

[Research on Compilers and Blockchain 45](#_211a8shwvtpd)

[Research on Finite State Machines 48](#_g20ddxxzysk)

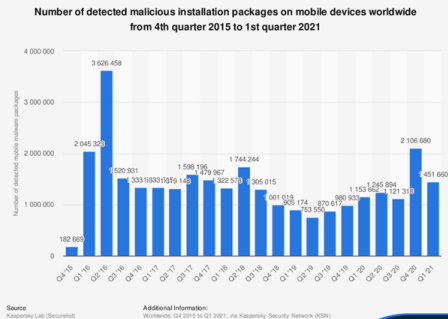
[Testing the Model 48](#_jxbikh6a7ebo)

[Additional Details (Manual Inspection of Android Applications) 49](#_t4z9emnbcrnd)

[**Methodology 50**](#_joibfcehdzgi)

[**Bibliography 50**](#_6brvqw49o5qe)

[**Appendices 52**](#_aqh44ohol4eh)

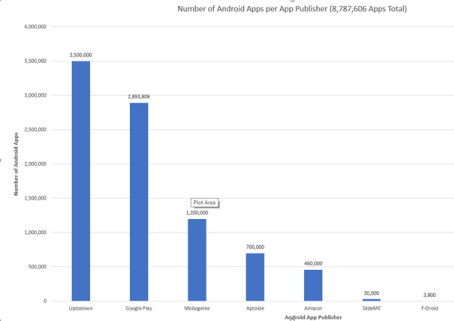


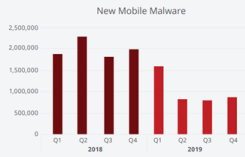
**Figure 1.** Number of detected malicious installation packages on mobile devices worldwide, Q4 2015 to Q1 2021.

Source: Kaspersky Security Network [(Lab, 2021)](https://www.zotero.org/google-docs/?kPTAvs)

# Introduction

Over the past several years, Google Play has released around 100,000 Android apps per month [(Sharma, 2019)](https://www.zotero.org/google-docs/?WdoVaR). Mobile malware has been a global problem, demonstrated through Kaspersky’s Statista graphic ([Figure 1](#tirxj8abw4k)). According to McAfee, mobile malware development is on the rise [(Samani, 2020)](https://www.zotero.org/google-docs/?s5aGfz). Based on their Quarter 1 report in 2019, they demonstrate that 35 million mobile malware programs have been produced. They also mention that advertisement click fraud is still a prevalent threat to the mobile advertisement market.

**Figure 2.** Number of applications per application publisher



**F****igure 3.** Number of new mobile malware by McAfee [(Samani, 2020)](https://www.zotero.org/google-docs/?XI2SY4)



**Fig****ure 4.** Total mobile malware by McAfee [(Samani, 2020)](https://www.zotero.org/google-docs/?SSNEH1)

Mobile malware has always been a problem, and it is not going away, as illustrated in [Figure 3](#58t25y1i3ihx) and [Figure 4](#lhp5geo48ei3).

App publishers need tools to counter the identified and not-yet-identified developers coding applications that will engage in click fraud, collect private information, and establish ransomware beachheads. The toolsets being developed will be cutting-edge automated tools (refer to the terminology table in the Appendix for all definitions of terms used). They will decompile the Android app, search for suspicious code using finite state analysis, inject blockchain calls to track suspicious apps, recompile the app, and use an auditing system to produce an auditing report for the publishers and the developers.

Specialized malware forensic analysis tools could be developed for:

• Publishers need a massive screening tool for a quick review of any potential threats;

• Publishers who are suspicious of the Android code and want to insert auditing code to track app state behavior;

• Businesses and educational institutions where users request app installation.

Our forensic analysis system aims to solve the problem associated with advertising click-fraud malware. Google could use such a system because of the growing issue of malware plaguing the Google Play store [(Osborne, 2021; Seals, 2021)](https://www.zotero.org/google-docs/?3ASUUI). Also, our tool could shorten the time it takes to publish an application through the Google Play store. Currently, this process of verification and publication can take up to seven days [(*HOW TO DEPLOY ANDROID APP IN GOOGLE PLAY CONSOLE IN 2021*, n.d.; *Publish Your App - Play Console Help*, n.d.)](https://www.zotero.org/google-docs/?tOGoqy).

Microsoft could also use such a forensic system. The company recently announced that Android apps would easily run on Windows 11 [(Kourafalos, 2021)](https://www.zotero.org/google-docs/?8ySMh5). However, this still introduces the same problem of having malware inside Android applications.

Mobile malware, in general, can occur in many forms [(*What Is Mobile Malware?*, 2018)](https://www.zotero.org/google-docs/?FlWvyA). Most malware attempts to steal sensitive data (passwords and credentials). However, other types of malware try to add other malware to a system. For example, some malware will display advertisements or fake advertisements during the execution of an Android application.

Android advertising malware (malvertising) involves fraudulent behavior related to advertising libraries. Two primary strategies are employed by advertising malware. The first consists of the displaying of fake advertising or hiding ads. This involves generating phony ad impressions. There are numerous related malware families and malware versions that utilize these strategies. The top ten malware variants produced 77% of malware activity in January 2021 [(cisecurity, 2021)](https://www.zotero.org/google-docs/?MSolGI). One of the most popular variants is Shlayer, a malvertising variant that accounts for 96% of total malware activity. Shlayer targets Apple devices and behaves by injecting advertisements into search results and applications [(Newman, 2020)](https://www.zotero.org/google-docs/?uRCz2M). It is interesting to note that Apple had notarized Shlayer as not being malicious.

Android application security has been a growing concern since the initial launch of the Android operating system. Much effort has been put into detecting security threats in Android applications, but some malvertising slips through the cracks. Researchers and security specialists use machine learning, artificial intelligence, threat analysis, and compiler injection to combat security threats for Android apps in the wild. Even though there are numerous approaches to detection and prevention, security specialists still need to detect certain types of sophisticated malware. These failures often occur at the machine-learning level [(cisecurity, 2021; Firdausi et al., 2010; Newman, 2020)](https://www.zotero.org/google-docs/?Sgmb5b). Machine learning sometimes fails because of classification failure due to inaccurate labeling of malware families.

Our research employs static and dynamic analysis of Android applications using compiler-based technology and finite-state machines. This is a more robust approach to detecting malvertising than static or dynamic analysis. Compiler injection has introduced new possibilities for detecting malware fraud. It has gained popularity in cybersecurity because it can be used to analyze and inspect applications. The ability to decompile and compile applications allows individuals to gain deeper insight into application behavior. This approach will enable us to create a cybersecurity analysis framework that professionals and researchers can use to identify advertisement fraud. In the following paragraphs, we discuss the major research questions addressed in this study, offer a literature review, define the hypotheses, outline the methods used to test the hypotheses and present the results and a discussion of the findings.

## Research Questions

One of the main goals of this study is to support security experts and advertisers in properly allocating funds and enhancing the user experience with Android applications. The user experience is improved by reducing the chances of ad fraud and concealed ads and decreasing the time users are exposed to counterfeit advertisements.

This research aims to discourage malicious actors from developing malware. Furthermore, we seek to establish a decision-making process based on heuristics informed by our findings to eliminate bias and deepen our understanding of strategies to prevent malvertising from infiltrating Android and Google’s advertising ecosystems.

**Question 1**: Can we develop a mechanism to identify individual Android applications that are part of the Zen and Joker malware families that contain malvertising click fraud?

The keyword “mechanism” refers to a component such as an online tool like VirusTotal combined with the Soot compiler framework that is used to help identify such malware families. Correct or incorrect behavior would have to be modeled to identify either malware family.

The next important step in our research agenda is to create an effective process for detecting advertising malware fraud. The finite state machine model can assist with detecting advertising malware. Specifically, we want to focus on the Zen and Joker malware. The Joker malware family has been known to generate fake advertisements. The Zen malware family has been found to create fake ad impressions using a command-and-control server or injecting code that performs clicks on advertisements. This leads us to the next central question:

**Question 2**: Based on our observations of the Zen and Joker malware families, can we model the behavior of these Android applications using a finite state machine model?

The behavior here refers to modeling and detecting malicious malvertising click fraud in both the Zen and Joker malware families. Identifying click fraud requires detecting correct behavior and what is expected using a finite-state machine model. The behavior of the Android application states would be added to the database during the dynamic analysis phase. Comparing the state data from the database would help with detecting whether or not any violations occurred.

A couple of complexities are involved when dealing with the Zen and Joker malware families. The first complex task involves identifying command and control servers that download malicious files and communicate with the Android application. The command-and-control servers can be discovered using string analysis and tools like VirusTotal. VirusTotal is a forensic tool that provides insight into whether or not the Android application has the potential to be malicious (see <https://www.virustotal.com/gui/home/upload>).

Another complexity related to malware detection is the use of third-party libraries by the malvertiser to show fake advertisements. We use the Soot compiler to conduct forensic analysis using finite state analysis and state machines. Using a tool like the Soot framework is helpful because it allows for incorporating forward-flow analysis. Forward-flow analysis can help discover the flow paths through the application, which helps uncover third-party libraries with potentially malicious intents. An additional discussion of the Soot framework is presented below.

Using finite state machines facilitates discovering the Zen and Joker malware families. It provides a more fine-grained approach to understanding when a malicious action is performed. Also, finite state machines can help create a decision process of what to do if malware incursions are discovered. For example, if an application performs three malicious actions in the finite state, it can be identified or marked as a dangerous application.

Based on the research questions discussed above, the significant contributions of this study include:

• Provides a decision process to detect advertisement malware.

• Uses a finite state machine model to detect malware fraud with high-level precision and accuracy;

• Provides insights into how advertisement fraud can be detected using both static and dynamic analysis;

• Develops a toolset that security professionals can use to detect advertisement fraud dynamically;

• Introduces a powerful network analysis approach that resides on top of the static and dynamic analysis functions

A detailed literature review follows that establishes a foundation for the research approach and the ensuing hypotheses.

# Literature Review and Related Work

This section focuses on the key literature related to finite state machines, forensic tools, Soot compilers and flow analysis, security of Android applications, machine learning, and other topics related to the research.

## Finite State Machines

Another growing topic is the study of finite-state machine models. Numerous articles explore the behavior of applications. For example, Jevitha et al. discuss a tool that they developed that uses finite-state machines to understand the Java application they analyze [(Jevitha K. P., 2020)](https://www.zotero.org/google-docs/?jsVKKR). Our research utilizes the same concept, except that we use it to help with the detection of advertising mobile malware.

Several research projects have used Finite-state machines to detect malware [(Beaucamps et al., 2010; Moser et al., 2007)](https://www.zotero.org/google-docs/?InudA9). They provide a concise way to determine whether or not fraudulent activity has occurred with advertisement malware. Using other graphical flow techniques would quickly lead to too many flows to follow, making it nearly impossible to create a generic flow chart to detect malvertising.

One interesting article [(Ziarek et al., 2016)](https://www.zotero.org/google-docs/?p3Ud6e) describes a tool called Jive, which is used for runtime visualization and verification of Java and real-time Java programs. It provides a way to verify application behavior. The tool uses diagrams, temporal query-based analysis over program schedules, executions and traces, finite state automata based on key object attributes of interest to the user, and verification of program execution correctness concerning design-time specifications. This article is significant because it uses finite-state machines and a visualization mechanism. The two concepts are very complementary to our approach to detecting advertisement fraud using dynamic analysis techniques.

Finite state analysis is a very powerful tool for monitoring software behavior [(Baranov, 2018)](https://www.zotero.org/google-docs/?PWAY7O). The benefit of finite state machines is that they enable better tracking during the dynamic analysis phase. For example, it is easier to tell during the runtime whether the advertisement options were set before or after the ad's display to the user.

Finite-state machines also enable the natural tracking of other options before an advertisement is displayed. Using finite-state machines also allows a more structured way of approaching advertisement fraud. Security analysts can create a decision process more easily because the states allow forensic analysts to understand what is expected versus what is out of the ordinary. Thus, if something unusual occurs, a decision can be made about what to do. In most cases, it would be wise to kill the application or inject Android code that would stop the application from continuing to run.

## Forensic Tools

Being able to classify advertisement placement fraud is a new and interesting topic. Liu et al. describe how they created a tool (DECAF) that can classify advertisement fraud placement [(Liu et al., 2014)](https://www.zotero.org/google-docs/?FSWGrE). The author’s implementation is focused on the Windows mobile platform. The tool utilizes user interface extraction and machine learning to help detect whether or not there was advertisement fraud placement. Also, the tool uses finite states to help with the detection process.

Graph structure analysis is another approach to detect malware in Android applications. Y. Du et al. discuss a tool they created that uses malicious subgraph mining to identify Android malware [(Du et al., 2021)](https://www.zotero.org/google-docs/?IX7wR3). The approach was designed to improve upon older techniques of large-scale graph structure analysis. Also, the authors use machine learning classification methods to help aid in the detection of malware, and they compared their approach against another popular tool, Androguard.

Web-based application analysis has been a recent topic of interest. Romano et al. discuss research where they look at the code’s assembly language to detect the behavior of the applications [(Romano & Wang, 2020)](https://www.zotero.org/google-docs/?aQRdxA). The tool provides a visualization of the behavior, function calls, and the interaction between functions. This is useful because they could use this to detect advertisement fraud in web-based applications.

## Compilers

Modern compilers can compile, decompile, inject code, and recompile to the original source. The Soot decompiler, in particular, has become very popular, and it can decompile and recompile Java and Android applications. Having the ability to deconstruct and reconstruct Android applications makes it a perfect compiler to analyze Android applications for malware.

Compilers are used to transform a high-level language into a low-level language. Byte code and machine code are examples of low-level languages. Decompilation takes a low-level language and transforms it into a high-level language. Soot decompiles the Java program by first executing the main method in the main class. Then the resolver is called to fetch a reference to a class source (referred to as a ClassSource). A ClassSource is an interface between the file containing the Java byte code and Soot. When the resolver has a reference to a ClassSource, it attempts to resolve it. The resolver is used to create the Soot class from the Java byte code class. Finally, the Soot class methods are set to an object. The Soot object is used to assist with the creation of the Jimple representation of the method.

When performing our analysis, we only deal with the intermediate representation of the code. In particular, we use the Soot compiler to decompile the Android application packages (APK) into Jimple intermediate representation and to leverage the framework to inject our code. Then the Soot framework will recompile the intermediate presentation back to its original APK file format.

### Control Flow Analysis

Control flow analysis is important to analyzing malvertising because it is a static analysis technique that allows researchers to see the control flow of a program. In compiler technology, the control flow is known as a control flow graph. A control flow graph is the graphical representation of control flow during the execution of programs. Control flow analysis is important because it helps with determining what information about a program at compile time is useful [(Shivers, 1988)](https://www.zotero.org/google-docs/?Ls9dF3).

The most influential compiler article related to our research discusses the role of control flow analysis [(Shivers, 1988)](https://www.zotero.org/google-docs/?TRf5QZ). Control flow analysis is essential because the Soot framework incorporates control flows to analyze Android applications. Control flow analysis displays the flow of data through the application. Without the knowledge gleaned from control flow analysis, it is not possible to incorporate more sophisticated analyses. Soot has a default control flow analysis, but it prevents the user from analyzing more complex applications and reduces the chances of successfully injecting any type of API calls; it also seriously limits blockchain calls at specific/critical Android app locations [(Sanders & Ziarek, 2021)](https://www.zotero.org/google-docs/?6nssLE).

### Android Compiler Injection

Android compiler injection is the process of injecting code into an application using a compiler. The Soot framework is unique because it is both a compiler and a decompiler [(Vallée-Rai et al., 2010)](https://www.zotero.org/google-docs/?qwASQ7). Also, the framework has the ability to read both APK and Java files. Soot can decompile the byte code of an Android application, inject code into the app, and repackage it back to its original form.

Compiler injection has been used to combat malware [(Enck et al., 2014; Freitas et al., 2021; Fuchs et al., 2009)](https://www.zotero.org/google-docs/?0E8gzA) and runtime visualization of code [(Ziarek et al., 2016)](https://www.zotero.org/google-docs/?zlSDfO). Most of the related literature will try to find either some anomalies or abnormal behavior in the malware. Machine learning and the injection of code into malware injection has been prevalent over the years. However, the injection of code for analyzing malvertising is a new concept that has not been explored yet.

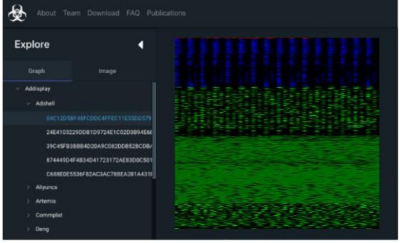
## Security of Android Applications and the Soot Framework

Android application security is important and can be divided into two categories, the Soot compiler framework and malware storage/detection. Another compiler research topic, relevant to our research, discusses the analysis of Dalvik byte code using the Soot tool [(Bartel et al., 2012)](https://www.zotero.org/google-docs/?UqKqzY). This brief article was a seminal contribution to compiler research. It illustrates how Soot can take the Dalvik byte code and decompile it to a Jimple representation. This is important because Soot could not analyze Dalvik byte code before the publication of the Bartell et al. article [(Bartel et al., 2012)](https://www.zotero.org/google-docs/?KzCo6H). Also, Dalvik byte code includes information about the Android application, such as the move instructions and other assembly-related operations. The authors also discuss the limitations of Soot in analyzing Dalvik byte code.

The article “Soot: A Java Byte Code Optimization Framework” on the Soot compiler discusses the optimization of byte code and the integration of intra-procedural and whole-program optimization. The paper laid the groundwork for the Soot framework tool we are using [27]. Soot was implemented to assist in the optimization of Java byte code. The initial intention of this research was to focus on the instrumentation of Java code. Instrumenting or instrumentation refers to the modification and analysis of a programming language through compiler technology. The study’s original experiment consisted of evaluating the Soot framework using 12 large benchmarks, including eight SPECjvm98 benchmarks running on JDK 1.2 for GNU/Linux. Soot provided an improvement of 8% when running the interpreter. It provided a 21% improvement when using the just-in-time compiler.

### Malware Storage/Detection

Freitas et al. [2] created MalNet, which is the largest publicly available cybersecurity image database. The most exciting aspect of MalNet is that it contains over 1.2 million software images across a hierarchy of 47 types and 696 families, with the aid of the AndroZoo dataset [31]. The authors used the Euphony classification structure to build the database. Euphony [32] generates the labels based on the VirusTotal report. It attempts to give each malware sample a unified “family” and “type” tag by learning the vendors’ patterns, structure, and lexicon. MalNet also allows researchers to improve on Euphony’s classification issues. Those issues include problems with naming disagreements and a lack of adopted naming standards across vendors. The most intriguing part of their findings is that users can see a graphical representation of the data (Figure 5: MalNet Explorer).



**Figure 5**: MalNet Explorer

Many articles have been written about databases that have been developed for storing applications that contain malware. For example, “AndroZoo is a malware database that comprises over 5 million Android applications and contains metadata” [33]. The authors collected the applications from multiple app stores (Google Play, Anzhi, AppChina, and FDroid). VirusTotal reports were also collected to be incorporated in the metadata. AndroZoo helped researchers conduct studies on malware samples that were collected. Furthermore, the authors created an analysis tool that helps with automatically reasoning about the security of Android applications [29]. The tool performs incremental checking of the application and extracts the Android application’s manifest. An exploration panel on the left allows users to select from the available image types and families. Users can then visually explore each image on the right.

Data flow analysis for tracking malware is another common technique employed in the wild. In one study, the authors created an analysis tool that helps with automatically reasoning about the security of Android applications [29]. It also performs incremental checking of the application. The tool extracts the Android application’s manifest file, checks the security specifications, and ensures compliance based on the data flows.

Beaucamps et al. wrote an article related to data flow analysis. The authors developed an alternative approach to detecting malware via the abstraction of application behaviors [18]. The behaviors were abstracted by dynamically abstracting and looking at the program traces. Suspicious behaviors are detected by comparing trace abstractions to reference malicious behaviors. The authors opted to have the execution traces represented as an automaton (referred to as a trace automaton). Then the traces were reconstructed, which gave a representation independent of the program.

Moser et al. discuss the limits of static analysis [34]. The authors explore a binary obfuscation scheme and attempt to obscure the flows of the application. They also attempt to obscure variables and other aspects of what the application is doing. Roundy et al. discuss the model they developed, which is an analyze-then-execute model for detecting Android malware [35]. The pre-execution analysis phase requires static and dynamic analysis techniques to detect malware in Android applications. They also use control-flow and data-flow analysis techniques. The combination of dynamic and static analysis techniques led to the claim that this process reduced the number of instrumentation locations by 100 times that of existing implementations.

Taint tracking is another approach that has been applied to tracking malware, and it is described in the research by Yin et al [36]. Yin et al. created a system called Panorama, which detects and analyzes malicious malware in applications [36]. Panorama was created to help code analysts and malware researchers study existing malware. The authors developed a three-step process: test, monitor, and analyze. Initially, they loaded the malware into the analysis environment and ran a series of automated tests on the malware. During the tests, they monitored the behavior and sensitive data that were accessed. The authors then used an approach of whole-system, fine-grained taint tracking. This approach marked the sensitive information introduced in the tests as tainted. The next step was to monitor the taint propagation over the entire system. The monitoring at this step was performed at the hardware level. They gathered the operating-system-level information using taint graphs. Taint graphs are “a representation of information flow that shows the processes that access tainted data, how the data propagates through the system, and finally, to which file or network connection this data is written to” [36]. The graphs support the policy-creation phase to help detect malicious software.

## Machine Learning

Machine learning research has been used extensively to discover and classify malware in Android applications [32, 37-39]. Fung et al. created a tool called RevMatch, a decision model [37]. The collaborative malware decisions were made based on querying the labeled malware detection history. If limited information was returned, then the system relied on partial matches. The framework used machine learning to help with the detection of malware. The authors examined multiple machine learning techniques to compare the false positives and errors that resulted from each technique.

Hurrier et al. discussed a tool, Euphony, which is an approach to classifying malware labels [32]. The labels are applied to each application using machine learning classification, and the tool is used to classify Android malware samples into malware families. The approach requires no prior knowledge of malware families for the classification process.

Shijo et al. introduced a method that uses static and dynamic analysis techniques to detect malware in Windows applications [38]. The static analysis method required that they extract the printable string information and use the strings to help catch the malware. The authors used machine learning vectors to help with the detection using a test and training dataset [38]. A vector in machine learning is a tuple of one or more values called scalars. The experimental results showed an accuracy of 95.8% using static analysis, 97.1% using dynamic analysis, and 98.7% using the integrated method.

Vecchio et al. wrote another machine-learning solution for classifying malware and introduced an approach to classifying malware in Android applications using graph structures of created strings [39]. Their system has a three-step process for detecting malware. The first step involves a static analysis that extracts the strings. In this step, they also extract the computations that use the strings. In the next step, they use a feature space generator to extract the computations. Finally, they use k-fold cross-validation and multiple machine learning algorithms. The novel approach creates a recall rate of 97% for classifying an application as having malware or not. Other authors created generic malware detection tools to combat malware using many techniques (dynamic and static analysis) and approaches that do not relate to machine learning [19, 40-42]. Tang et al. developed one such technique [42].

Specifically, Tang et al. developed Dual-Force, a dynamic analysis technique for detecting malware [42]. The technique simultaneously forces both Java and JavaScript code of WebView applications to execute along various paths requiring no environment setup and without providing any inputs manually. WebViews that the malware uses to execute the malicious code execution are then exposed. The technique correctly exposed malicious payloads in 119 out of 150 WebView malware instances.

Suarez-Tangil et al. created DroidSieve, a static analysis Android malware classifier [41]. The tool classifies the malware into a similar malware family by exploiting obfuscation-invariant features and artifacts introduced by obfuscation mechanisms. The authors achieved 99.82% accuracy with zero false positives for malware detection and 99.26% accuracy for family identification of obfuscated malware.

Maggi et al. created AndroTotal, a malware repository and malware dynamic analysis framework for Android applications [40]. They developed AndroTotal to allow researchers to scan Android applications against arbitrary malware detectors automatically. The tool was released to the public in 2013 and was composed of 18,758 submitted malware samples. There were at least a thousand distinct accounts that took interest, including several research groups.

Moser et al. claim that dynamic analysis alone is not enough to detect Android malware [19]. Instead, they propose analyzing multiple execution paths of the Android application to detect the malware and get a better overall picture of the app’s inner workings. The authors aimed to resolve test coverage and proposed creating a system that automatically detects malware in Android apps and provides a better report on the malware that is detected.

There are two recent empirical studies [43, 44] of malware behavior. Chen et al. explored state-of-the-art tools and found that they have limited capability to identify banking apps’ data-related security weaknesses [43]. Also, they presented Ausera, a static analysis system that detects banking application weaknesses. They found 2,157 flaws in 693 banking apps that were collected from 83 countries. Twenty-one banks confirmed the weaknesses that were discovered.

In another recent empirical study, Hammad et al. examined the effects of code obfuscation on Android apps and anti-malware products [44]. They conducted an analysis on obfuscated 3,000 benign and malicious apps. They generated 73,362 obfuscated apps using 29 obfuscation strategies from seven open-source, academic, and commercial obfuscation tools. Their findings suggest that 1) code obfuscation significantly affects Android anti-malware products, 2) even trivial obfuscations severely affected most anti-malware products, and 3) combined obfuscation strategies do not successfully evade anti-malware products more than individual strategies. They also discovered that the detection of anti-malware products depends not only on the applied obfuscation strategy but also on the leveraged obfuscation tool. The study concludes that anti-malware products are slow to adopt signatures of malicious apps, and code obfuscation often results in changes to an application’s semantic behaviors.

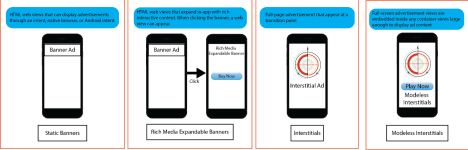
There are several articles that attempt to detect such changes to a system [45-47]. Shan et al. discuss self-hiding behavior (SHB) in Android applications, how to detect such devious behaviors, and the tools they developed for catching SHB-based Android applications [47]. Their study examines the characteristics of SHB and how to detect such behaviors using static analysis techniques. They show how SHB will try to hide the behavior and how some apps try to hide their presence.

An article by Hammad et al. discusses a tool that looks for self-hiding behaviors [46]. The authors developed SALMA, an Android software system that monitors itself and attempts to adapt its behavior at runtime to prevent a wide range of security risks. SALMA incrementally and efficiently analyzes the Android system when detecting incremental changes to the system. When a system change occurs, the system determines the affected portion and the subset of the security analyses that need to be performed. The combination of these analysis techniques dramatically improves the performance of the approach.

Demissie et al. created an approach for detecting vulnerable Android applications using a multi-level approach to identify app vulnerability discovery [45]. The first step in the process involves looking at the vulnerable applications and organizing the apps into specific clusters based on their description. Then, they generate the permission re-delegation model for each cluster. This process characterizes the permission re-delegation behaviors of the apps in the group. The approach checks whether the re-delegation behaviors are consistent with the model for the specific category to which each tested app belongs. If the results are not consistent, a test is performed to check for vulnerabilities in the app. The results of this approach show an 81.8% recall rate and 100% precision.

## Advertisement Types

Mobile ads can contain numerous types of advertising. These include static banners, rich media expandable banners, interstitials, and modeless interstitials [48]. Static banners are HTML web views that can display advertisements through an intent, a native browser, or an Android intent. An intent is a messaging object that can request an action from another app component [48]. Meanwhile, “rich media expandable banners are HTML web views that expand in-app with rich interactive content” [48]. Interstitials are full-page advertisements that appear at a transition point in the Android application. Modeless interstitials are full-screen advertisement views that are embedded inside any container views and large enough to display ad contents. Both interstitials and modeless interstitials are loaded in the background before it is time to display them. We have condensed this classification into a graphic that illustrates the various ad types, as shown in Figure 6: Mobile advertisement types.



**Figure 6**: Mobile advertisement types

## Ethereum Framework

We conducted extensive research on using Ethereum’s private blockchain environment for the tracking of mobile advertisements that spanned the course of a year. Then we explored and tried to implement using the new interplanetary file system (IPFS) to track advertisements. Our findings suggest that the blockchain is not suitable yet because of its complexities and limitations. Such limitations include requiring specific Android application API levels to allow it to use the APIs that enable the ommunication between the private blockchain and the app. Also, IPFS is not a suitable solution because of the lack of functionality that it provides. This is because IPFS only stores information and provides no functionality to perform any analysis of the information stored on IPFS.

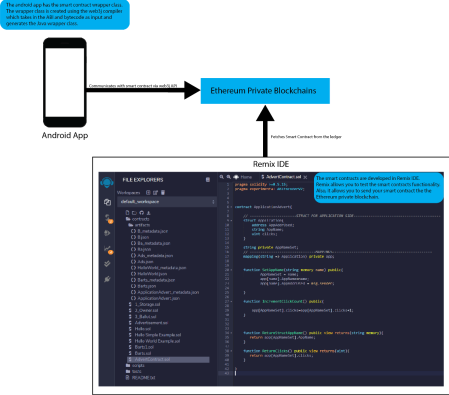
The blockchain is enticing and shows promise because of its immutability and smart contract infrastructure. We believe numerous papers could be written about how to use blockchain and compilers to track malvertising. For example, we could discuss how finite state machines could be incorporated in a smart contract to track the states that an application is in and determine if malvertising occurred.

### Ethereum Infrastructure

The Ethereum infrastructure is complex because of all the components it uses to allow for the creation of the public- and private-based blockchain environments, plus the use of smart contracts. The complexity is also increased because of the immutability of the data that are stored on the blockchain. A private blockchain is stored locally on a server. This is created using the geth console. The geth console is Ethereum’s proprietary tool that allows for creating private blockchains, creating Ethereum accounts and hosting the blockchain on a PC or server. A public blockchain environment is hosted on the web through multiple servers and computers.

A smart contract is a set of code that is written in the solidity programming language. It is a non-legally binding document that allows individuals to post information on the blockchain ledger. Smart contracts are stored in the ledger. However, currently the blockchain is not yet a suitable option. Zhou et al. discuss how the blockchain is underperforming under the excessive loads of transactions and why it is not suitable for blockchain applications [49]. Also, a solution has been proposed for using a serverless architecture to reduce such downfalls of traditional blockchain architectures [50]. Let us begin by discussing the complexities of the Ethereum private blockchain technology. All smart contracts are written in the solidity programming language. The solidity language is an object-oriented, high-level language for implementing smart contracts [51].

To get the Ethereum smart contracts on the blockchain, individuals must use the Remix integrated development environment and deploy the smart contract on the blockchain (Figure 7: Ethereum overview). Once deployed, Remix can fetch the smart contract from the blockchain. Let us now discuss the complexities of developing an Android application that can communicate with the deployed smart contract.



**Figure 7**: Ethereum overview

### Android Smart Contract Communication

To get the Android application to communicate with the developer’s deployed smart contract on the private blockchain, developers must extract both the Application Binary Interface (ABI) and byte code into the appropriate ABI and the Ethereum Virtual Machine byte code files. The ABI code represents the methods and structures of the smart contract code [52]. It is used to point to specific function calls. The byte code is the intermediate representation of the smart contract code.

Once these are on the system, it is necessary to use the web3j compiler to generate the Java wrapper class. This class represents the appropriate code that contains all the smart contract classes, the inputs, and the return types for each class. Even though we are not focusing on the blockchain for this dissertation, it is still important to discuss the implications and complexities of using the blockchain for our future work.

## Sensitive Data Tracking

Another related topic is sensitive data tracking, which allows for the analysis of possible attack vectors on mobile phones. TaintDroid is a tool that tracks possible attack vectors on a mobile device using control flow analysis [28]. The tool incorporates a system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sensitive data sources in smartphones. The authors developed a tool to allow users to view in real-time how applications use their sensitive data.

# Implementation Framework

This section introduces the implication framework for this research study, including the Soot framework, the details related to Soot transformation, and code obfuscation.

## Soot Framework

The Soot framework reads in Java files and Android APK files (Figure 8: Soot analysis overview) and then produces the intermediate representation. When Soot reads in these files, it examines the intermediate representation of the main class and then builds an object that references all the class’s main methods. The Soot object constructs the Jimple representation and then looks for any code instructions that tell Soot where to inject additional code. The compiler attempts to inject the specified code into the intermediate Jimple code and repackages it in its original form (Android APK file or Java).



**Figure 8**: Soot analysis overview

The Soot framework contains complex and important concepts that require further discussion. For example, a scene is a critical concept in Soot that manages the Soot classes for the application being analyzed. The scene also holds all the Android application classes associated with the APK that is being analyzed.

Another important concept is the SootClass, which represents the current class that the Soot tool is analyzing. The Soot class can contain many methods, and each method is referred to as a SootMethod. Another standard output format that the Soot tool uses is Jimple. Jimple is the simplified Java code format that the Soot framework uses to construct and deconstruct Android applications. Each Android application method contains a JimpleBody, or a body of the code enclosed in the current SootMethod, represented in Jimple form. Each SootMethod has many units. Units in this context are code fragments. Figure 9: Soot application overview presents an example of how Soot represents an Android application in Jimple form.



**Figure 9**: Soot application overview

As illustrated in Figure 9: Soot application overview, it is easy to manipulate Android APK files using the Jimple data structures that Soot provides. The Soot framework is a mature and powerful tool, carefully crafted by scholars at McGill University. The Soot tool allows the efficient and quick injection of blockchain calls into Android APKs. Other compiler-based tools, such as the Apktool, are not as easy or as efficient to use for blockchain code injection.

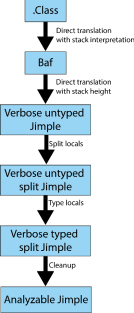
Another valuable set of features that the Soot framework offers is the numerous analysis functions. These analysis functions include forward and backward flow analysis, flow-through, points-to analysis, template-driven intra-procedural data flow, and directed call graphs [53]. The most important and commonly used analysis tools are forward and backward flow analysis. Forward flow analysis provides informati n about the future, or new code, and the path of execution [53]. Backward flow analysis provides information about the code regarding what variables will be used and not used [53].

## Soot Transformation

This section discusses the five steps necessary to convert byte code to analyzable Jimple code.

### Direct Translation With Stack Interpretation

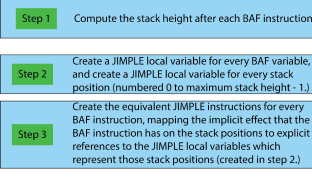
The Soot transformation is what allows Soot to convert Java to the Jimple intermediate representation [27]. Soot takes in class files, converts the code to Jimple, and then converts it to byte code (Figure 10: Byte code to Jimple code steps). The conversion uses two alternatives, generating GRIMP code or BAF code. GRIMP code is a tree-like code that Soot traverses. BAF code is stack code that is optimized by Soot.



**Figure 10**: Byte-code to Jimple code steps

### Direct Translation With Stack Height

Next, each Baf instruction is converted to an equivalent Jimple instruction sequence. This is done in three steps, as shown in Figure 11: Direct translation steps. The first step is accomplished by performing a depth-first traversal of the Baf instructions. Step 2 involves creating a Jimple local variable for every Baf variable and creating a Jimple local variable for every stack position. For Step 3, we create the equivalent Jimple instructions for every Baf instruction. This involves mapping the implicit effect that the Baf instruction has on the stack positions to explicit references to the Jimple local variables representing the stack positions.



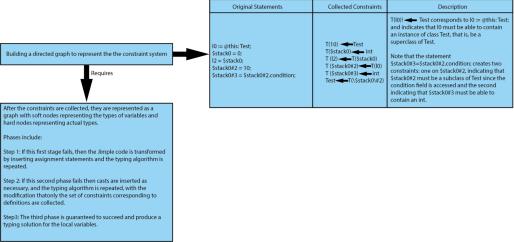
**Figure 11**: Direct translation steps

### Split Locals

To prepare for typing, the local variables must be split so that each local variable corresponds to one use-def/def-use web. This is because the Jimple code generated by the previous section may be untypable. Splitting the locals is then accomplished by computing the webs by traversing the use-def and def-use chains and associating one local variable with each produced web [27]. A web is a subset of all the uses and definitions of a particular local variable that is self-contained. This subset can be renamed without affecting the behavior of the code.

### Type Locals

Next, each local variable is given a primitive, class, or interface type. An algorithm developed by Etienne Gagnon et al. [27] must be utilized to solve this NP-hard problem: “The typing algorithm in Soot is an efficient multistage typing algorithm based on solving a type of constraint system; each stage is attempted in turn to provide a solution, and each is progressively more complex.” The phases and steps for this process are shown in Figure 12.



**Figure 12**: Typed local phases

### Cleanup

The final step is the cleanup. This involves performing some compaction to

eliminate the redundant copy statements present in the code due to the direct translation from the byte code.

## Obfuscation

Code obfuscation predates back to the early-to-mid-20th century when the Enigma machine was used in WWII to send secure messages. The Enigma cipher device encoded messages using cryptography techniques. The concept of hiding information was adapted by malicious actors (i.e., hackers) adapted the concept of hiding information. Code obfuscation is one way that hackers attempt to hide their malicious intents. Obfuscation attempts to make the code difficult or impossible for humans to read. For example, a hacker might attempt to remove lines between statements and crunch them together to make the code difficult to read. Another tactic imposed might be to create short variable names or ones that are unrelated to the code (referred to as entity renaming). A more advanced obfuscation tactic is to change the structure of the source code, for example, changing the code structure to a more complicated but semantically identical syntax. Another tactic is to hide strings from decompilers. For example, this might involve the hacker using the server to replace a plain-text string IP address value with a public key, which then gets decrypted at runtime by a server. The hackers could also replace source code variable names with cryptic names that have no meaning. The goal of the hackers using obfuscation is to confuse forensic malware analysts.

## Hypotheses

We have developed several hypotheses based on the research questions and the literature discussed in the preceding section of the paper. In particular, we plan to examine the application of compilers, static and dynamic analysis tools, and flow analysis to develop a decision process for detecting advertising malware fraud.

Finite-state machines have been used to detect malware [18, 19]. Using finite-state machines is crucial because it allows for a more concise way to determine whether or not fraudulent advertisement malware activity occurred. However, using flow charts would quickly lead to too many flows to follow, and it would be nearly impossible to make a generic flow chart to detect malvertising.

We are developing a finite state machine model as a mechanism to detect fraudulent activities. We have analyzed several common fraudulent Android application malware scenarios to ensure that our model can detect the malvertising fraud. We thoroughly reviewed the Google, Amazon, and Facebook library documentation to understand how the libraries function inside Android applications.

To create a generic finite machine model, we need to be familiar with the documentation from several sources. This is necessary to analyze similarities in API calls and to determine when API libraries change. It also ensures that our finite state model will not fail when advertising libraries change. This leads to the following hypothesis:

**Hypothesis 1**: Compilers and finite state machines can detect Android malware (phony advertisements, fake impressions, and the hiding of ads).

Using a combination of static and dynamic analysis is critical. Based on prior research [34, 38], it has been proven that using either approach alone limits and hinders the ability to gain better insight into program flows. For example, malvertising applications will often use a command-and-control server to send instructions to the application [54]. The orders might tell the application to hide or show ads. Thus, if we were only to use static analysis, we might not detect the fraudulent activity in a mobile app that was running. Similarly, researchers and practitioners have discovered that only using dynamic analysis has serious limitations [38]. Specifically, the major hurdle is not knowing exactly where to place finite state calls because the flow of the application and API calls is not clear. In essence, it is very difficult to understand the inner workings of Android applications without applying both static and dynamic analysis, which leads to the next hypothesis:

**Hypothesis 2**: Static and dynamic analysis techniques help determine where to place finite state machine calls in Android applications to detect advertisement malware.

Using flow analysis is essential because it allows us to gain a better understanding of the flow paths [35]. Data flow analysis will enable forensic practitioners to understand what data, definitions, and data flows are used in a program. Data flows are important because they allow us to understand whether or not fraudulent behavior will occur in the application [55]. This is critical because flow analysis and finite state automata will determine if a fraudulent violation occurred based on the finite state machine. Flow analysis is necessary to reduce the chances of missing the vital behavior details in an Android app.

**Hypothesis 3**: Flow analysis helps determine at which points in the application a finite-state machine can detect fraudulent behavior in an Android application.

A decision process that creates a scientific approach to determine if one or more instances of malware advertising exist is necessary to build a foundation for a formal model [37]. Ignoring the decision process might lead to an inability to detect several occurrences of fraudulent behavior. Also, it will guide forensic malware analysts when there is ssuspected malvertising but the tools do not detect the fraudulent activity. For example, the decision process will help decide if a fake advertisement is shown, even when the tool shows that it is a legitimate advertisement. Without an agile or robust process, analysts might not understand what course of action to take if several violations occurred versus a single fraudulent activity. This could also lead to the development of a severity level metric based on the number of fraudulent violations and the threat level of each violation. This leads to the next hypothesis:

**Hypothesis 4**: A decision process can be used to detect whether or not advertisement malware exists in Android applications.

In the next section, we provide details on the work accomplished to date on this research project.

# Work Accomplished Related to the Research Questions and Implementation Details

The major work activity on this research project at this point includes the following activities:

* Conducted research and wrote a paper about comparing the use of Soot and the APK tool to inject code into Android applications. The paper was presented at HICSS 2021 (Sanders and Ziarek, “A Comparison and Contrast of Apktool and Soot for Injecting Blockchain Calls Into Android Applications”).
* Conducted research and wrote a paper about developing a blockchain compiler framework to detect an audit Zen click fraud using smart contracts to track advertisements (Sanders and Ziarek, “Developing a Zen Click Fraud Detection Framework Using Smart Contracts,” submitted to HICSS, 2022).
* Analyzed 40 Android applications that contained malware. Each application took approximately 30 hours to analyze manually. This set the groundwork for a dynamic and static analysis infrastructure for analyzing Android application advertisements and Android malware.

The next subsections detail the activities related to the research project.

### Research on Compilers and Blockchain

We have conducted a study comparing the use of Soot and the APK tool to inject code into Android applications. The paper, “A Comparison and Contrast of Apktool and Soot for Injecting Blockchain Calls Into Android Applications,” was presented as a poster at the UB CSE poster session before the winter break of 2020. The paper was also presented at the 2021 Hawaii International Conference on System Sciences in January 2021 [26]. This research investigated whether Soot or the Apktool was better for injecting the blockchain calls into Android applications. An overview of the findings is displayed in Table 1. This paper was the foundation for the dissertation.

| **Features** | **Soot** | **Apktool** |
| --- | --- | --- |
| Automated analysis and injection of blockchain calls | Yes | No |
| Language  output | Jimple, Shimple,  and BAF | Smali |
| Can define main class for Android APK | Yes | No |
| Uses assembly-like  language | No | Yes |
| Can generate APK as output | Yes | Yes |
| Has poor  documentation | No | Yes |
| Better suited for  blockchain injection | Yes | No |

**Table 1**: Soot and Apktool comparison

The follow-up research paper on developing a Zen advertisement fraud-detection framework using smart contracts investigated the viability of using blockchain to track advertisements in Android applications. We initially considered injecting blockchain calls into the Android applications to track the advertisement fraud using finite-state machines. The analysis led us to the conclusion that the blockchain was not a viable solution for tracking malvertising. The Ethereum framework is very fragile and too slow for a viable implementation for tracking advertising fraud at this point. The limitations, complexity, and efficiency of the Ethereum infrastructure led us to conclude that the blockchain was not a viable solution for storing data on fraudulent activities.

For example, using the Ethereum blockchain to track our states limited developers to using a very early version of the Android API level (28). The web3j API library requires specific functions and features that only Android API level 28 and higher have. Also, it takes a tremendous amount of time to complete a blockchain transaction. This is compounded further when transactions pile up in the processing queue. Some would argue that using a private blockchain is more effective because individuals do not have to wait as long for a transaction to complete. In addition, they do not have to pay for the transactions to clear when using a private blockchain. While this is true, there is still a significant cost. The long-term cost is the amount of time and effort it takes to create the private blockchain and enable the Android devices to communicate correctly with the private blockchain. A decentralized network is not necessary for this application because we only have a few publishers, and they would probably use this just for their ad libraries. For example, Google would not participate in such an activity due to confidentiality concerns. In addition, current editions of the Ethereum blockchain network are not reliable. The SQLite database is a more robust and reliable compared to the Ethereum network [26]. There has also been discussion about the fragility of the Bitcoin network [56].

### Research on Finite State Machines

We developed a finite state machine model to analyze malvertising behavior, as shown in Figure 23. The model targets the clicking, showing, loading, and displaying of mobile ads, which, according to numerous research articles, are the most common states that contribute to advertisement fraud. A flaw of the current model is that it does not account for the use of command-and-control servers sending commands to the Android application and instructing the application to hide, show, or display a different mobile ad. We plan to address the command-and-control server issue using finite state analysis.

### Testing the Model

We performed numerous tests detecting malvertising using our framework. We injected log messages into the Android applications and thus were able to log when the application started and completed. Our framework was able to inject and detect when an ad was clicked, loaded, and shown. We tested the functionality and success of our tool by interacting with a set of applications that we knew had advertisement malware. Then we compared the results with what the finite states should be, based on our example scenarios. Many of the test apps we developed from scratch attempted to mimic the advertisement malware behavior. This is important because we wanted to isolate the behavior in a simpler application before moving on to a more complex application that is deployed to the Google Play store.

### Additional Details (Manual Inspection of Android Applications)

In addition to writing the papers and implementation details, we analyzed 40 Android applications that contained malware. This involved decompiling Android applications, analyzing the Jimple code, and manually mapping out code behavior, recompiling it, and then tracking the behavior. Based on the behavioral analysis, we were able to determine what malicious calls were executed, what third-party libraries communicated with the app, and the degree of advertisement fraud in each application. Each app took about 30 consecutive hours to analyze. The total time to analyze the apps was at least 1,200 hours. This entailed mapping out all the classes and all the calls to the other classes in each function. Also, I studied the Google, Amazon, and Facebook advertisement library calls in each Android application.

These results have interesting implications. The Google Play store receives about 100,000 apps per month. Thus, it would take about 4,000,000 hours to manually analyze all applications that are submitted to the store. An automated process for detecting advertisement fraud is required because publishers do not have the resources to conduct such manual inspections. As illustrated below, automating the process could cut down the analysis time to about ten minutes per app.

The take-aways from the manual inspection of Android applications are:

* Malvertising developers used third-party libraries to hide malicious intentions related to click fraud.
* Malvertisers use command-and-control servers to feed into the app instructions and download additional malicious files.
* Malvertisers use third-party advertisement libraries to hide the malicious intentions. The malicious intentions of the malvertisers were that they pretended to use legitimate advertisement libraries but, rather, used a custom-made library.
* We used the manual inspection to help determine the application start state. • We used the manual inspection process to help with developing heuristics of where to place or inject code into Android applications for detecting fraudulent ad clicks

# Methodology

This section focuses on the key details related to finite state machines, dataset, and details related to the MADScanner framework.

# Bibliography

[Baranov, S. (2018). *Finite State Machines and Algorithmic State Machines: Fast and Simple Design of Complex Finite State Machines* (1st ed., Vol. 1). ISBN Canada. https://www.amazon.com/Finite-State-Machines-Algorithmic-Complex/dp/1775091724](https://www.zotero.org/google-docs/?SkXLYw)

[Bartel, A., Klein, J., Le Traon, Y., & Monperrus, M. (2012). Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, 27–38. https://doi.org/10.1145/2259051.2259056](https://www.zotero.org/google-docs/?SkXLYw)

[Beaucamps, P., Gnaedig, I., & Marion, J.-Y. (2010). Behavior Abstraction in Malware Analysis. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, & N. Tillmann (Eds.), *Runtime Verification* (pp. 168–182). Springer. https://doi.org/10.1007/978-3-642-16612-9\_14](https://www.zotero.org/google-docs/?SkXLYw)

[cisecurity. (2021). *Blog \textbackslashtextbar Top 10 Malware January 2021*. https://www.cisecurity.org/blog/top-10-malware-january-2021/](https://www.zotero.org/google-docs/?SkXLYw)

[Du, Y., Cui, M., & Cheng, X. (2021). A Mobile Malware Detection Method Based on Malicious Subgraphs Mining. *Security and Communication Networks*, *2021*, e5593178. https://doi.org/10.1155/2021/5593178](https://www.zotero.org/google-docs/?SkXLYw)

[Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2014). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, *32*(2), 5:1-5:29. https://doi.org/10.1145/2619091](https://www.zotero.org/google-docs/?SkXLYw)

[Firdausi, I., lim, C., Erwin, A., & Nugroho, A. S. (2010). Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection. *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, 201–203. https://doi.org/10.1109/ACT.2010.33](https://www.zotero.org/google-docs/?SkXLYw)

[Freitas, S., Duggal, R., & Chau, D. H. (2021). MalNet: A Large-Scale Cybersecurity Image Database of Malicious Software. *arXiv:2102.01072 [Cs]*. http://arxiv.org/abs/2102.01072](https://www.zotero.org/google-docs/?SkXLYw)

[Fuchs, A. P., Chaudhuri, A., & Foster, J. S. (2009). SCanDroid: Automated Security Certification of Android. *ACM Conference on Security & Privacy in Wireless and Mobile Networks*. https://drum.lib.umd.edu/handle/1903/11847](https://www.zotero.org/google-docs/?SkXLYw)

[*HOW TO DEPLOY ANDROID APP IN GOOGLE PLAY CONSOLE IN 2021* (Vol. 2021). (n.d.). techliance. https://blog.techliance.com/deploy-android-app-in-google-play-console/](https://www.zotero.org/google-docs/?SkXLYw)

[Jevitha K. P., S. M., Swaminathan Jayaraman, Bharat Jayaraman. (2020). Finite-state model extraction and visualization from Java program execution. In *Software: Practice and Experience* (Vol. 51, Issue 2, pp. 409–437).](https://www.zotero.org/google-docs/?SkXLYw)

[Kourafalos, O. (2021, July 18). *What Does Windows 11’s Support for Sideloading Apps Mean?* MUO. https://www.makeuseof.com/windows-11-sideloading-apps-support/](https://www.zotero.org/google-docs/?SkXLYw)

[Lab, K. (2021). *Number of detected malicious installation packages on mobile devices worldwide from 4th quarter 2015 to 1st quarter 2021*. statista. https://www.statista.com/statistics/653680/volume-of-detected-mobile-malware-packages/](https://www.zotero.org/google-docs/?SkXLYw)

[Liu, B., Nath, S., Govindan, R., & Liu, J. (2014). {DECAF}: Detecting and Characterizing Ad Fraud in Mobile Apps. *DECAF*, 57–70. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu\_bin](https://www.zotero.org/google-docs/?SkXLYw)

[Moser, A., Kruegel, C., & Kirda, E. (2007). Exploring Multiple Execution Paths for Malware Analysis. *2007 IEEE Symposium on Security and Privacy (SP ’07)*, 231–245. https://doi.org/10.1109/SP.2007.17](https://www.zotero.org/google-docs/?SkXLYw)

[Newman, L. H. (2020). Apple Accidentally Approved Malware to Run on MacOS. *Wired*. https://www.wired.com/story/apple-approved-malware-macos-notarization-shlayer/](https://www.zotero.org/google-docs/?SkXLYw)

[Osborne, C. (2021, March 9). *Malicious apps on Google Play dropped banking Trojans on user devices*. ZDNet. https://www.zdnet.com/article/malicious-apps-on-google-play-dropped-banking-trojans-on-user-devices/](https://www.zotero.org/google-docs/?SkXLYw)

[*Publish your app—Play Console Help*. (n.d.). Retrieved October 5, 2021, from https://support.google.com/googleplay/android-developer/answer/9859751?hl=en&utm\_source=pocket\_mylist](https://www.zotero.org/google-docs/?SkXLYw)

[Romano, A., & Wang, W. (2020). WasmView: Visual Testing for WebAssembly Applications. *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 13–16.](https://www.zotero.org/google-docs/?SkXLYw)

[Samani, R. (2020). *McAfee Mobile Threat Report* [Report]. https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf](https://www.zotero.org/google-docs/?SkXLYw)

[Sanders, S., & Ziarek, L. (2021). *A comparison and contrast of APKTool and Soot for injecting blockchain calls into Android applications*. https://doi.org/10.24251/HICSS.2021.820](https://www.zotero.org/google-docs/?SkXLYw)

[Seals, T. (2021, March 9). *Google Play Harbors Malware-Laced Apps Delivering Spy Trojans*. https://threatpost.com/google-play-malware-spy-trojans/164601/](https://www.zotero.org/google-docs/?SkXLYw)

[Sharma, A. (2019). Google Play Store Stats and Facts You Should Know in 2021. In *Appinventiv*. https://appinventiv.com/blog/google-play-store-statistics/](https://www.zotero.org/google-docs/?SkXLYw)

[Shivers, O. (1988). Control flow analysis in scheme. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 164–174. https://doi.org/10.1145/53990.54007](https://www.zotero.org/google-docs/?SkXLYw)

[Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (2010). Soot: A Java bytecode optimization framework. *CASCON First Decade High Impact Papers*, 214–224. https://doi.org/10.1145/1925805.1925818](https://www.zotero.org/google-docs/?SkXLYw)

[*What is Mobile Malware?* (2018, December 4). Forcepoint. https://www.forcepoint.com/cyber-edu/mobile-malware?utm\_source=PANTHEON\_STRIPPED](https://www.zotero.org/google-docs/?SkXLYw)

[Ziarek, L., Jayaraman, B., Lessa, D., & Swaminathan, J. (2016). Runtime Visualization and Verification in JIVE. In Y. Falcone & C. S�nchez (Eds.), *Runtime Verification* (pp. 493–497). Springer International Publishing. https://doi.org/10.1007/978-3-319-46982-9\_33](https://www.zotero.org/google-docs/?SkXLYw)

# Appendices

Terminology

| **Term** | **Definition**  A program that translates statements written in a source programming language and into machine language, object code or assembly. |
| --- | --- |
| compiler | A program that translates machine language, object code or assembly into a high level language such Java. |
| decompiler | A low-level representation of program code that has been compiled. It can closely resemble assembly language. |
| bytecode | The Android Package Kit is used to distribute and for the subsequent execution of an Android application. It is similar to the exe format in Microsoft |
| APK  code injection | Windows.  The process of injecting statements into an application at a specific location without disturbing the flow of the application code. |
|  | A compiler framework that is able to decompile and compile Java code with the capability of analysing and instrumenting Java code. |
| soot | Refers to the modification and analysis of a programming language through the use of compiler technology. |
| instrumentation | An intermediate representation of Java code that Soot generates as output. |
| jimple | A compiler framework that is able to simply decompile and compile Java code. |
| APKTool | A peer-to-peer network that allows for the sharing of data among a vast number of peers [ All data stored on the blockchain is |
| blockchain  Ethereum blockchain | **mearian\_faq\_2017**].  immutable.  A blockchain environment that allows the use of smart contracts. |
| smart contract | A contract with written rules and terms allowing for controlling the storage, sharing, and modification of data. |
|  |  |
| Backward  flow analysis | Provides information about the future code along the path of execution. |
| Forward  flow analysis | Provides information about past code along the path of execution. |