Android Malware Static Analysis Techniques

Suzanna Schmeelk, Junfeng Yang and Alfred Aho
Columbia University
New York, New York
{schmeelk, junfeng, aho}@cs.columbia.edu

ABSTRACT

During 2014, Business Insider announced that there are over a billion users of Android worldwide. Government officials are also trending towards acquiring Android mobile devices. Google's application architecture is already ubiquitous and will keep expanding. The beauty of an application-based architecture is the flexibility, interoperability and customizability it provides users. This same flexibility, however, also allows and attracts malware development.

This paper provides a horizontal research analysis of techniques used for Android application malware analysis. The paper explores techniques used by Android malware static analysis methodologies. It examines the key analysis efforts used by examining applications for permission leakage and privacy concerns. The paper concludes with a discussion of some gaps of current malware static analysis research.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.2.1 [**Software Engineering**]: Software Architectures—*Patterns*; H.4 [**Communications Applications**]: Information browsers

Keywords

Cyber Security, Android Application Security, Static Analysis, Malware Analysis, Java

1. INTRODUCTION

Mobile computing is steadily transforming our computing sphere. In 2008 Google introduced Android with the capability of installing and executing third-party applications [14]. The beauty of a mobile-phone application-based architecture is the flexibility, interoperability and customibility it provides users. This same flexibility, however, also allows and attracts malware.

In July 2014, Business Insider [34] reported that 85% of global phone shipments–totaling over one-billion users [9]–ran the Android Operating System. The ubiquitous use of Android is therefore increasing the need for malware-detection mechanisms. Aware of the trend towards security-related concerns, a 2014 article published in the Wall Street Journal [7] announced that Google will embed Samsung's security platform, called Knox, into any device that ships the latest version of Android. Google's announcement came as Google was kicking off the annual I/O Developers' Conference in San Francisco. "The move [for Samsung] to work closely with Google comes shortly after Samsung said five of its Galaxy-branded smartphones and tablets loaded with Knox received approval from the U.S. Defense Information Systems Agency, which allowed them to be listed as an option for Pentagon officials [7]." Google, and the mobile-computing market at large, are clearly finding it necessary to boost security and, more specifically, boost their malware prevention mechanisms.

There are multiple techniques used in preventing, thwarting and detecting malware. Typically they can be summarized as *dynamic analysis* or *static analysis*. *Dynamic analysis* techniques are used when executing the code. *Static analysis*, used traditionally during standard compiling, examines source (or intermediate, binary) code for patterns. Results from both techniques are informed by the analysis precision. The focus of this paper is in understanding the current state-of-the-art static analysis research techniques used in the analysis of malware. Static analysis techniques can be used to address many software questions raised during different software lifecycles stages. This paper found four high-level archetypal motivations for using malware-specific static analysis techniques across the development and maintenance of applications as follows: development, sandboxing, cyber improvement labs and search-and-find. We discuss each category in Section 4.

2. RELEVANT ANDROID ARCHITECTURE

Google's Android operating system is a Linux-based variant. The operating system acts as middleware between phone hardware-firmware and third-party applications. Apps [16], in this model, are run within the operating system in a sandboxed environment, either the Dalvik Virtual Machine (DVM) or the Android Runtime (ART), where each app is given its own unique Linux system-level *uid* [17]. The DVM is slightly different than Oracle's Java Virtual Machine (JVM) [30]. First, the DVM supports 218 opcode instructions [16, 26, 10] versus JVM's 256 instructions [29]. Second, Dalvik uses a register-based architecture as opposed to Oracle's stack-based architecture [10]. Third, Dalvik's

CISR '15, April 07 - 09, 2015, Oak Ridge, TN, USA

http://dx.doi.org/10.1145/2746266.2746271

Software Development Kit (SDK) library is quite different than Oracle's SDK library. Fourth, DVM interacts with the underlying phone system through a mediated IPC system Binder [17] and the user via call-backs [16]. Fifth, an important difference is that the application structures of Google's Application Package Code, file name extensions, inflation, permissions and user interface screens are unique to code running inside a DVM.

All the building blocks of an Android Application Package can be useful to inform a robust static analysis system. An Android application is comprised of many meta-elements and its internal code is somewhat complex [14]. An Android Application APK file consists of a manifest file, strings file, main file, R file, resource file, program code and more.

The more robust the analysis of the static code, the more accurate the overall results.

3. ANDROID MALWARE CONCERNS

Android is well-known to be dominating the mobile computing market. Desirable characteristics for a secure system typically fall into three well-known categories: confidentiality, integrity and availability (CIA). The CIA triangle has been the industry standard for information security since the beginning of computer security. In the last few years, the triangle has been extended into a polygon model to include privacy, identification, authentication, authorization and accountability. Threats to security are traditionally manifested in a threat to one or more corners of the polygon.

Malware is a general term for describing any form of unwanted software that threatens the CIA model. It is usually defined as software designed to do damage or to do other unwanted actions on a computer system. We can classify systems for malware discovering capabilities as either being preventative or detective. Preventative information security analysis techniques are geared towards protecting software and systems before they are publically released. This category includes developer tools geared at analyzing code for weaknesses. Information security threat-detecting mechanisms are geared towards locating "in-the-wild" malware which is infecting or will potentially infect benign systems. In the case of Android, preventative measures include analyzing Android applications before release; whereas, detective measures include locating publically available malicious applications that are free available for download from the internet.

3.1 Confidentiality and Integrity Threats

Confidentiality in computer systems pertains to concealment of information and resources [6]. Integrity of computer systems pertains to the origin of data and how well the data is protected on the current machine [6]. In the Android application domain, many threats fall into both categories.

Privilege Escalation [44] is the ability of a program to escalate its privileges without root approval, or mobile-phone user approval in the case of Android. In Android there is one predominate way to accomplish a privilege escalation attack–through interfacing, via intents, with other benign applications and within its own app components with higher privileges [8, 42].

Spyware [44] is software that is installed surreptitiously, collects user system information and unknowingly transmits the information off the system [27]. In the case of Android, many free applications generate revenue on their use of Adware libraries. Sometimes adware libraries can either knowingly or unknowingly be nefariously disclosing user information and activities to malicious third parties [11]. High financial charges from the clandestine use of SMS messages to premium numbers [44] have been frequently noted in literature. Additionally, unknowingly routing traffic through third-party locations intended to snoop [6] is another concern of spyware. Specifically, use of third-party applications, opens up this possibility.

Backdoors [40] are Trojan software that is installed into a system to allow an unidentified attacker access to the system. In Android applications, this means allowing an attacker get into the mobile device or remotely control it [44] via commands.

3.2 Availability Threats

Availability of computer systems makes it easy for users to access their systems and their information without interference [38]. Availability can be affected via botnets [40], usurpation [6] and purposeful delay [6].

3.3 Other Android Polyhedral Threats

Whitman and Mattord [38] describe five extensions to the CIA model: privacy, identification, authentication, authorization and accountability. Using the definitions given by Whitman and Mattord, we briefly describe the new categories.

The privacy category is directed towards an organization that collects the information and uses it in ways known only to the provider. In Android, this would mean that data collected from applications by an organization from which it originated would be used only in ways approved by the user. Adware routinely falls into this category as a threat.

The identification category is a characteristic of a computer system that recognizes individual users, typically by means of a user name or other ID. Usernames for the Android and applications fall into this category.

Authentication "occurs when a control proves that a user possesses the identity that he or she claims [38]." Threats to this model are malware that steal user credentials.

Authorization occurs after authentication, when the user is granted permissions based on their level of trust. Threats to this security capability include escalation of privileges.

Finally, accountability is about associating all activities and processes with a particular entity. Threats to this model in Android are usurpation malware.

4. ARCHETYPAL TECHNIQUES

Static analysis techniques are archetypal for object-oriented programming but must be enhanced to fit Android. An example archetypal static analysis is type checking, used ubiquitously in standard compilers, which dates back to IBMs mid-19th century Fortran and perhaps earlier. In this section, we examine many archetypal static analysis techniques used in the prevention and discovery of malware that started as early as the 1990s. Of course, due to the uniqueness of the Android environment, most techniques have had to be re-customized to accurately analyze the new system semantics. One of the most interesting findings of the paper is the number of standard static analysis technque variations being used for malware analysis.

4.1 Reachability Analysis

Nikolic and Spoto, [28], define reachability from a program variable "v to a program variable w states that from v, it is possible to follow a path of memory locations that leads to the object bound to w [28]." Reachability is an important concept within static analysis to perform abstract interpretations precisely on a static representation of the code under investigation. There are multiple analysis types that rely on the accuracy of reachability (or unreachability) analysis.

Side-effect analysis tracks "which parameters p of a method might be affected by its execution in the sense that the method might update a field of an object reachable from p [28]." As an example, an arbitrary assignment only affects p if the *lvalue* variable used in the assignment is reachable from p. During static analysis, if there is knowledge that an arbitrary *lvalue* variable is not reachable from p, then there is knowledge that that particular *lvalue* will not affect p.

Field initialization analysis is a traditional analysis statically tracking variable initialization, or lack thereof, within a program. One of the primary analyses used during this static analysis is *nullness*. Nullness analysis determines at an arbitrary object reference either *lvalue* or *rvalue*, if the object can be uninitialized. Field initialization analysis can also be used for determining possible aliasing [28].

Cyclicity analysis is an analysis technique that examines code of cyclical data structures. Cyclical data structures can occur at an arbitrary assignment if the *lvalue* variable used in the assignment is reachable from the *rvalue*. The contrapositive also holds true, based on the rules of logic.

Path-length analysis counts the maximum number of pointer dereferences that can be followed from a program variable.

Program slicing is closely related to reachability analysis as it determines the set of programs statements, the program slice, which may affect the values at some point of interest.

Taint Analysis is the analysis of a code base for variable influence. There are two forms of taint analysis—*explicit* or *implicit*. *Explicit* terminology is used when taint can be directly passed during assignments. *Implicit* terminology is used when taint is indirectly passed during control flow structures. *Implicit* passes can happen in a few ways, for example Inter Component Communication (ICC) message, Broadcast messages and Media interaction.

Pointer Analysis is the analysis of memory references for particular traits. *Alias analysis* tracks which pointers access the same heap memory.

Sharing analysis tracks potential variables that might be bound to overlapping data structures. *Points-to* analysis computes the objects that a pointer variable might refer to at runtime. *Escape analysis* tracks where in a program a pointer can be accessed. *Shape analysis* âĂŞ determines how many dereferences can occur at different program stages. It discovers and verifies properties of linked, dynamically allocated data structures. Shape analysis captures aliasing, points-to and acyclicity information.

4.2 Data Flow Analysis

There are multiple static ways to accurately and precisely capture data flow within a program. Currently, many of the techniques are prone to time and space constraints. *Intraprocedural* analysis tracks data flow strictly within a procedure. *Interprocedural* data flow analysis analyses values between procedures usually through a technique referred to as a *call graph*. *Context-sensitive* data flow analysis is an interprocedural analysis that considers the calling context when analyzing the target of a function call. *Path-sensitive* data flow analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions. *Flow-sensitive* data flow analysis takes into account the order of statements in a program.

4.3 Call Graphs

Call graphs are static graphs of function calls throughout a program. In pure Java programs, call graphs of functions are used for analysis of applications and libraries. Android can be statically analyzed with multiple "types" of call graphs. An *Activities Call Graph (ACG)* can be constructed to precisely analyze activity data flow. A *Component Call Graph (CCG)* can be statically constructed to precisely analyze explicit component communications. Wei, et al., [37] use a *Data Dependency Graph (DDG)* to analyze data flow of android application data. Wei, et al., also introduce a data flow graph for *Inter-component data flow (IDFG)* to more precisely analyze implicit data flow among components.

4.4 Entry-point Analysis

Entry-point analysis is especially important in mobile computing applications mainly due to the callback framework. This analysis determines where a program may begin execution. This is particularly challenging with Android because of the architecture. The application architecture is structured around the use of callbacks and activities specified by user demand; there is not a single main method where execution begins.

4.5 Intermediate Representations

Intermediate representation is the internal representation of code traditionally found during mid-compiler stages.

4.5.1 SMALI Code

Smali is sponsored by Google and, curiously, Google chose Icelandic terms for a few of their naming conventions. *Smali* is the Icelandic term for *assembler*. Smali is therefore an Android assembly language. *Baksmali* is the Iceland term for *disassembler*. And, *Dalvik* is the Icelandic term for *fishing village*. Depending on the direction of code transformations, either Smali or Baksmali will be used to transform up or down to the Dalvik dex format.

4.5.2 Watson Libraries for Analysis (WALA)

IBM's T.J. Watson Research Center was the original developer of the *Watson Libraries for Analysis (WALA)*, which is a Java program analysis library. The typical *WALA* client will use the libraries to perform interprocedural analysis. The code is traditionally analyzed by building a class hierarchy, building a call graph and building a graph of control flow nodes. *WALA* IR is an immutable control-flow graph of instructions in *Static Single Assignment (SSA)* form.

4.5.3 Soot

Soot, developed by the Sable Research Group of McGill University, was first introduced in a 1999 CASCON paper [35] entitiled, "Soot - a Java Bytecode Optimization Framework." *Soot* is designed for performing optimizations in Java. *Soot* provides four intermediate representations for analyzing and transforming Java bytecode: *Baf*, *Jimple*, *Shimple* and *Grimp*. "*Baf* is a streamlined representation of bytecode

which is simple to manipulate. *Jimple* is a typed 3-address intermediate representation suitable for optimization. *Shimple* is an SSA variation of *Jimple*. *Grimp* is an aggregated version of *Jimple* suitable for decompilation and code inspection. *Soot* can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode. [39]"

In 2012, Bartel, et al., [4] introduced *Dexpler* as a tool for converting Android Dalvik to *Jimple* using *Soot*. The research of Bartel, et al., provides a comprehensive table that maps Dalvik byte code instructions to *Jimple* statements.

5. MALWARE ANALYSIS RESEARCH

Static analysis is a technique to analyze code without executing the code. Classically, static analysis is performed by a compiler at compile time. The ultimate goal of static analysis informs the methodologies used during the analysis. Our research of the top ACM listed papers found four main goals for performing malware specific static analysis. We describe each category in the following subsections.

5.1 Developer Tools Research

The set of static analysis tools that are used by developers to guard against future malware application usurpation fall into this category of developer tools research.

In 2012, Lu, et al., [25] published CHEX. CHEX is a tool for statically vetting Android applications for component hijacking vulnerabilities. CHEX static analysis relies on the Dalysis framework. Dalysis uses *DexLib* and IBM's *WALA* to build an intermediate representation of the application. Their analysis uses flow and context-sensitive data flow analysis. The authors introduce the idea of *app splitting* to generate and permute data-flow summaries at each split–fragment of code reachable from a single entry point. The authors claim that the tool is low overhead. The technique is useful for collecting information about permission leakage, unauthorized data access, intent spoofing and hijacking vulnerabilities in general.

5.2 Mobile Sandboxing Research

Many researcher groups are establishing mobile sandboxes for determining application safety. These sandboxes are designed to detect application malware using static analysis and dynamic analysis combinations.

5.2.1 SmartDroid

In 2012, Zheng, et al., [41] introduced a novel technique to automatically and systematically reveal user-interface-based discrepancies between strings given within Android interface and permission files and the application behavior. Their technique first used static analysis to analyze strings and to attempt (un)successfully to correlate with the application behavior in terms of switch paths. Then, dynamic analysis is used to verify the static results. After disassembling the Dalvik byte code into Smali, two call graphs are created–a function call graph (FCG) and an activity call graph (ACG). The FGC tool can determine all call paths to sensitive APIs. The AGC maps a sensitive source function to a sensitive sink Activity. The information gathered is passed to the dynamic analysis machine to match the user interface element with the corresponding user interface event functions.

5.2.2 Detecting Control Flow in Smartphones

Graa, et al., [18] published a tool that combines static and dynamic analysis to detect control flow in mobile phones. Their technique enhances dynamic taint analysis from simply explicit taint analysis used in TaintDroid, to include implicit analysis as used in Trishul. Trishul statically builds control flow graphs to create a summarization bitmap for the applications dataflow. The bitmap is used along with a runtime enforcement technique to improve the dynamic taint flow problem.

5.2.3 Mobile-Sandbox

In 2013, Spreitzenbarth, et al., [33] developed a prototype Mobile-Sandbox to both statically and dynamically study Android applications for malware. Their static analysis relies on multiple techniques. First, the static analysis tool quickly extracts a hash value of the application, which is checked against the widely-used VirusTotal database, and the resulting match from Kaspersky is used for reporting classification of the application into existing malware families. Second, the analysis extracts the permissions, SDK version, the intents and the services using the *aapt* tool, which is delivered with the Android SDK. Third, the sandbox extracts the dex files, converts them into an intermediate representation and searches for potentially dangerous functions and methods. Examples of functions that the sandbox looks for are *sendTextMessage()*, which sends SMS messages; *getPackageInfo()*, which searches for installed AV products; *getSimCountryIso()*, which determines the country where the user resides; *java/lang/Runtime exec()*, which executes a specific command in a separate process; and any calls to available encryption libraries. At the end of the static analysis, an XML file of relevant information is created and passed for dynamic analysis.

5.2.4 AsDroid

In 2014, Huang, et al., [20] published AsDroid. The tool matches application behavior to permission requests specified in the Android manifest. The tool uses partial call graphs, points-to analysis, reachability analysis, interprocedural data flow analysis, intraprocedural data flow analysis and text analysis. Their technique first used static analysis to analyze strings and to attempt to either successfully or unsuccessfully (inverse) correlate with the application behavior in terms of switch paths. AsDroid relies on IBM's *WALA* library, which builds an intermediate representation using static single assignment form; it, then, uses reachability analysis to connect suspicious function calls with top-level user interface actions.

5.2.5 Exposing Application Behavior Risks

Johnson, et al., [22] published a novel static and dynamic analysis technique to expose software security and availability risks for commercial mobile devices. Their static analysis relies on a quick first-pass and a subsequent in-depth analysis. The first quick pass extracts hardcoded strings in an application and analyses them for well-known URLs or suspicious strings, such as *nc*, *su*, etc. The in-depth analysis attempts to understand the behavior of applications by comparing the applications' requested permissions (given in the manifest file) with the corresponding functionality based on the API calls. It does this by using the *apktool* to convert the byte code into Smali code. The final output of the static analysis is both a report of discrepancies between

permissions requests and actually API calls as well as a list of API calls and their controlling flow for invocation during dynamic analysis. The groups of API calls of interest are commands executed, execution of binaries, Java reflection, loading of libraries, network events, files accessed and dynamic class-loading. The dynamic analysis tries to get a complete coverage of application execution using information gathered about control-flow from the static analysis.

5.2.6 A5

In 2014, Vidas, et al., [36] published the Automated Analysis of Adversarial Android Applications (A5) mobile sandbox tool which is currently housed at CMU. The overarching goal of the tool is to invoke and fingerprint malware. A client browses to the A5 website and uploads an application. The sandbox analyzes the application and returns analysis results to the client. Techniques used in the mobile sandbox are both static and dynamic. A5 relies on Androguard to extract information from the Manifest file. The sandbox then decompiles the application into Jimple using Soot. Jimple builds IR of components control flow graph which is further flush-out into abstract syntax trees to examine each statement element. The tool is not flow-sensitive. This process informs the sandbox of a set of receivable intents to invoke during dynamic analysis.

5.3 Cyber Improvement Research

The cyber improvement research category, primarily used by research institutions, is generalized research methods that encompass statically analyzing an application, possibly improving it and, then, releasing it back into the wild.

5.3.1 Using Static Analysis for Automatic Assessment

In 2011, Batyuk, et al., [5] introduced the static analysis tool located on a server in Germany. The tool was one of the first to introduce a remote repacking technique where a security conscious user would visit their DAI-Labor website, upload their Android application of choice, have it statically analyzed on the DAI-Labor server, receive a report and then have DAI-Labor clean-up the application and repackage/resign and return to the user for use. The repackaging does, of course, use different jar signers. The analysis server relies on *readelf* to extract function calls. The static analysis detectors on the analysis server are written in Python and use regular expression pattern-matching analysis techniques. The analysis reports are stored in a central *CouchDB* non-relational database. HTML reports are generated dynamically for the users to better understand their application.

5.3.2 API-level Access Control Tool

Hao, et al., [19] evaluated the effectiveness of API-level access control mechanisms to secure Android libraries by bytecode rewriting. In general bytecode rewriting security mechanisms statically analyze an application to identify security-sensitive API method use. It, then, overwrites the methods of interest to access the secure wrapper code rather than the unsafe original code. When the application is run, the new framework checks the calls with new privileged operations. Hao, et al., found in their research a number of potential attacks geared towards bypassing or deterring the additional overwritten safety-checks.

5.3.3 MobSafe

In 2013, Xu, et al., [40] published a prototype tool called MobSafe, which is a cloud computing-based forensic analysis for massive mobile applications that relies on data mining to determine if an application contains malware. The tool relies on both static and dynamic analysis. MobSafe uses Static Android Analysis Framework (SAAF), which is sponsored by Google for a good part of the data flow analysis. SAFF extracts the contents of applications, applies program slicing to the Smali code to analyze a program for permissions and heuristic patterns and focuses on functions of interest. MobSafe also relies on *readelf*, *ded*, *apktool*, *Androguard* and *Soot* to analyze other components of the applications. MobSafe then uses Android Security Environment Framework (ASEF), Stace and Randoop for the dynamic analysis. ASEF starts *ABD* logging, invokes the application, sends random input and compares the log to the CVE library. Stace watches system calls in the Linux kernel. Randoop stimulates the Android application with random inputs and watches for output.

5.3.4 Chiromancer

In 2014, Anwer, et al., [1] published Chiromancer. The tool is featured off a web-server at the Indraprastha Institute of Information Technology, Delhi. A security-concerned Android user can browse to the server and upload the application of interest. The Chiromancer sandbox analyzes the application by using *Soot* to decompile the code and to perform data flow analysis. After analysis, the server returns configuration options to the user. The user specifies the enhancements, if any, to be made and responds. The sandbox performs the changes, re-signs the application with new keys and returns the application to the user. Example optimizations include skipping adware entirely, removing SMS to premium numbers, removing GPS update frequency and removing latent wake locks.

5.4 Search-and-Find Research

Search-and-find research encompasses independent general static analysis techniques that have been discovered to locate specific malware manifestations and vulnerabilities.

5.4.1 Analysis of Android Applications' Permissions

The Johnson, et al., [21] study statically analyzed the Android API classes to map which API calls required permissions and the permission type. Their study extended research conducted in 2011 by Felt, et al., [12] at the University of California at Berkeley. Felt, et al., compiled an extensive list of Android API calls, intents and content providers that required permissions in Android 2.2.1. Johnson, et al., identified two additional strands of API calls that required permissions. First, they found an additional 146 API calls in the *com.android.server* package that required permissions. These additional calls happen when using Java reflection. The second group of functions they identified were an additional 217 calls in the source code and used only by system processes. After identifying these functions, they studied 141,372 applications and compared the function calls with permissions listed in the manifest.

5.4.2 AndroidLeaks

Gibler, et al., [15] introduced a prototype tool, AndroidLeaks, to automatically detect potential privacy leaks using static analysis. They applied their approach on 24,000

Android applications from several Android markets and verified, within approximately thirty hours, that over 2,000 applications leaked data. The paper considers private data to be phone information, GPS location, wifi data and audio recordings using the microphone. AndroidLeaks internal representation is built first by converting the Dalvik code to Java using *ded* or *dex2jar* and, then, using IBMs *WALA* framework to build the internal representation. The analysis consists of a context-sensitive system dependence graph (SDG) based in part on call graphs. The analysis is designed with a taint-aware slicing, intraprocedural and interprocedural analysis. Reachability analysis is used to determine if sensitive information may be sent over the network. They used their technique for identifying taint sources in callbacks, used for leaking of private data.

5.4.3 DroidAlarm

The DroidAlarm tool, published in 2013 by Zhongyang, et al., [43] finds privilege escalation threats and malware exploits by accessing private data through applications with higher privilege. The tool performs an interprocedural call flow graph. It then inspects the code for data flow to public interfaces that may be suspicious.

5.4.4 Anadroid

In 2013, Liang, et al., [24] introduced the static malware analysis framework, Anadroid. The framework takes advantage of two techniques to soundly raise analysis precision. First, Anadroid uses a pushdown system to model dynamically dispatched interprocedural and exception-driven control flow. Second, the tool uses *Entry-Point Saturation* to soundly approximate all possible interleaving of asynchronous entry points of analysis applications. The researchers used the DARPA Automated Program Analysis for Cybersecurity (APAC) test suite of 52 applications and found the push-down method saved execution time.

5.4.5 AppProfiler

In 2013, Rosen, et al., [32] introduced the AppProfiler tool. AppProfiler uses a data flow analysis technique for gathering information about the possible set of values calculated at various points in a program. The researchers derived behavior rules from examining a few applications. They then used *dex* to decompile the APK. Finally, they checked their rules using Fortify's Static Code Analyzer (SCA) to perform analysis of the rules over all applications. In total, the researchers analyzed over 33,000 applications for the paper. The aim of their research is to produce a static analysis tool that works on a large scale.

5.4.6 FlowTwist

In 2014, Lerch, et al., [23] introduced FlowTwist to efficiently use context-sensitive and inside-out taint analysis for wide-breath-analysis Android applications. FlowTwist is a novel taint-analysis approach that works inside-out (i.e., tracks data flows from potentially vulnerable calls to the outer level of the API, which the attacker might control). The tool builds an interprocedural control flow graph (ICFG), followed by an IFDS (Interprocedural Finite Distributive Subset) to compute data-flow facts. IFDS is performed twice: once for integrity-related problems and once for confidentiality-related problems. Empirical results for components of FlowTwist were presented in the research; the tool was not formally prototyped.

5.4.7 Apposcopy

Feng, et al., [13] developed the Apposcopy taint analysis tool. The application matches signatures based on control-flow (e.g., launches a broadcast) and data-flow (e.g., reads private data and sends it through a designated sink). The tool produces an Inter-Component Call Graph with specific control and dataflow properties. A benefit to their method is that tthe tool works well with malware obfuscation.

5.4.8 FlowDroid

FlowDroid is an open source tool introduced in 2014 by Arzt, et al., [2, 3]. Analysis, based on *Soot* library, uses *Jimple* and produces an accurate call-graph analysis framework. They used dataflow and pointer analysis for a more precise taint analysis. The dataflow analysis techniques they used were context, flow, field and object-sensitive to reduce the number of false alarms. They also used both forward analysis and on-demand backward analysis for their taint analysis. Finally, they compared FlowDroid with Fortify's Static Code Analyzer and AppScan and found FlowDroid was more effective. The researchers also introduced the *DroidBench* benchmark to compare analysis techniques.

5.4.9 Amandroid

In 2014, Wei., et al., [37] introduced the Amandroid tool. The tool uses flow-sensitive, context-sensitive data flow analysis to calculate object points-to information by building inter-procedural control flow graphs (ICFG). The tool uses ICC points-to facts based on explicit and implicit data flow analysis. Amandroid uses *dex2ir*, which is a modification of Android SDK's *dexdump* that ships with Android.

5.4.10 DroidRay

In 2014, Zheng, et al., [42] published DroidRay. The researchers analyzed 250 Android firmwares and 24,000 pre-installed applications. They found that pre-installed applications have more permissions than applications a user downloads. The tool inspects, in part, the shared UID from preinstalled applications specified in the *AndroidManifest.xml* file. An interesting find was that many pre-installed applications of the same application across firmware share the same UID. The research was driven to analyze control flow for privilege escalation looking for malicious applications.

6. RESULTS: RESEARCH GAPS

Figure 1 shows the distribution of CIA research motivations for twenty-one of the top Android research papers examined for this study. We see, 50% of the papers are geared towards using static analysis techniques for finding potential permission and data leakage. Four papers examined application behavior, as informed by system calls, as compared to the applications declared intentions. Four papers examined potential privilege escalation exploitations. One paper examined receivable intents which can be viewed as botnet activity. Finally, two papers examined security techniques in general.

We noticed a few research gaps. First, broadcast receiving functionality is hinted at in some of the literature, but the analysis is not fully described. It could be that applications could invoke suspicious activities upon receiving notification.

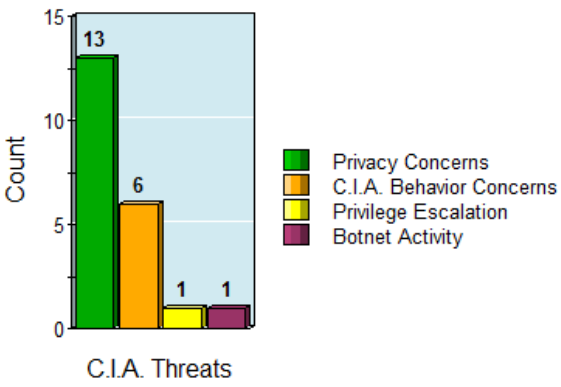

Figure 1: Malware security research distributions.

Second, many papers explained that covert channels were outside the scope of their work. We find this a particularly interesting finding as it shows openings in cyber security research. Third, there are few, if any, papers that explore the malware that exploits availability measures. We could think of a few: triggering the mobile device to vibrate until battery completion and triggering an application to dereference a null pointer and terminate. Fourth, similar to a *Cron*, Android's built-in alarm manager sends intents even when an application is paused. Exploring the manager breadth would be a worthwhile investigation. Finally, Petsas, et al., [31] found that malware can determine that it is running in a sandbox and behave differently based on certain sandbox virtualization parameters. Static analysis can inform the sandbox that these particular values are being checked and, perhaps, even convert them to no-ops for subsequent sandbox dynamic analysis.

7. CONCLUSION

This paper examined twenty-one of the most relevant papers listed by the ACM for static analysis techniques being used for Android malware research. This paper was written in motivation to understand in detail what static analysis techniques were being employed for malware analysis. Our findings were three-fold. First, we were impressed by the variations of standard static analysis techniques being used "in-the-wild." Second, we found that literature is geared more towards confidentiality threats (data leakage and permission abuse), leaving other CIA threats un-researched. Third, we found that research in this area is very fragmented. We plan, in the future, on developing more general static analysis techniques that applies to multiple threats.

8. REFERENCES

[1] S. Anwer, A. Aggarwal, R. Purandare, and V. Naik. Chiromancer: A tool for boosting android application performance. In *Proc. of the 1st International Conf. on Mobile Software Engineering and Systems*, MOBILESoft 2014, NY, NY, USA, 2014. ACM.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6), June 2014.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, NY, NY, USA, 2014. ACM.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, NY, NY, USA, 2012. ACM.

[5] L. Batyuk, M. Herpich, S. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conf.*, Oct 2011.

[6] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[7] J. Cheng. Samsung security platform to be part of next android version. *Wall Street Journal.*, June 2014.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proc. of the 9th International Conf. on Mobile Systems, Applications, and Services*, MobiSys '11, NY, NY, USA, 2011. ACM.

[9] J. Edwards. Proof that android really is for the poor. *Business Insider.*, June 2014.

[10] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. of the 20th USENIX Conf. on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[11] E. Erturk. A case study in open source software security and privacy: Android adware. In *Internet Security (WorldCIS), 2012 World Congress*, June 2012.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the 18th ACM Conf. on Computer and Communications Security*, CCS '11, NY, NY, USA, 2011. ACM.

[13] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proc. of the 22Nd ACM SIGSOFT International Symp. on Foundations of Software Engineering*, FSE 2014, NY, NY, USA, 2014. ACM.

[14] M. Gargenta and M. Nakamura. *Learning Android: Develop Mobile Apps Using Java and Eclipse*. O'Reilly Media, Inc., 2nd edition, 2014.

[15] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th International Conf. on Trust and Trustworthy Computing*, TRUST'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] Google. Android online developer portal. http://developer.android.com/. Acc. Dec. 1, 2014.

[17] Google. Android open source project. https://source.android.com/. Acc. Dec. 1, 2014.

[18] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and

A. Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In *Proc. of the 4th International Conf. on Cyberspace Safety and Security*, CSS'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] H. Hao, V. Singh, and W. Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proc. of the 8th ACM SIGSAC Symp. on Information, Computer and Communications Security*, ASIA CCS '13, NY, NY, USA, 2013. ACM.

[20] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proc. of the 36th International Conf. on Software Engineering*, ICSE 2014, NY, NY, USA, 2014. ACM.

[21] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of android applications' permissions. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conf. on*, June 2012.

[22] R. Johnson, Z. Wang, A. Stavrou, and J. Voas. Exposing software security and availability risks for commercial mobile devices. In *Reliability and Maintainability Symp. (RAMS), 2013 Proc. - Annual*, Jan 2013.

[23] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proc. of the 22Nd ACM SIGSOFT International Symp. on Foundations of Software Engineering*, FSE 2014, NY, NY, USA, 2014. ACM.

[24] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proc. of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '13, NY, NY, USA, 2013. ACM.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proc. of the 2012 ACM Conf. on Computer and Communications Security*, CCS '12, NY, NY, USA, 2012. ACM.

[26] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proc. of the 27th Annual ACM Symp. on Applied Computing*, SAC '12, NY, NY, USA, 2012. ACM.

[27] Microsoft. Malware protection center glossary. http://www.microsoft.com/. Acc. Dec. 1, 2014.

[28] u. Nikolić and F. Spoto. Reachability analysis of program variables. *ACM Trans. Program. Lang. Syst.*, 35(4), Jan. 2014.

[29] Oracle. Ch. 7. opcode mnemonics by opcode. https://docs.oracle.com/javase/specs/jvms-7.html. Acc. Dec. 1, 2014.

[30] Oracle. Java virtual machine technology. http://docs.oracle.com/. Acc. Dec. 1, 2014.

[31] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proc. of the Seventh European Workshop on System Security*, EuroSec '14, NY, NY, USA, 2014. ACM.

[32] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proc. of the Third ACM Conf. on Data and Application Security and Privacy*, CODASPY '13, NY, NY, USA, 2013. ACM.

[33] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proc. of the 28th Annual ACM Symp. on Applied Computing*, SAC '13, NY, NY, USA, 2013. ACM.

[34] S. Tweedie. Android is crushing the smartphone market, and it is not even close anymore. *Business Insider.*, July 2014.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative Research*, CASCON '99. IBM Press, 1999.

[36] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proc. of the 4th ACM Workshop on Security and Privacy in Smartphones*, SPSM '14, NY, NY, USA, 2014. ACM.

[37] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security*, CCS '14, NY, NY, USA, 2014. ACM.

[38] M. E. Whitman and H. J. Mattord. *Roadmap to Information Security: For IT and Infosec Managers.* Delmar Learning, 1st edition, 2011.

[39] S. Wiki. Android reversing analysis. http://wiki.secmobi.com/tools. Acc. Jan. 1, 2015.

[40] J. Xu, Y. Yu, Z. Chen, B. Cao, W. Dong, Y. Guo, and J. Cao. Mobsafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology*, 18(4), Aug. 2013.

[41] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proc. of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, NY, NY, USA, 2012. ACM.

[42] M. Zheng, M. Sun, and J. C. Lui. Droidray: A security evaluation system for customized android firmwares. In *Proc. of the 9th ACM Symp. on Information, Computer and Communications Security*, ASIA CCS '14, NY, NY, USA, 2014. ACM.

[43] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Droidalarm: An all-sided static analysis tool for android privilege-escalation malware. In *Proc. of the 8th ACM SIGSAC Symp. on Information, Computer and Communications Security*, ASIA CCS '13, NY, NY, USA, 2013. ACM.

[44] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 2012 IEEE Symp. on Security and Privacy*, SP '12, Washington, DC, USA, 2012. IEEE Computer Society.