*droid: Assessment and Evaluation of Android Application Analysis Tools

BRADLEY REAVES and JASMINE BOWERS, University of Florida
SIGMUND ALBERT GORSKI III, North Carolina State University
OLABODE ANISE, RAHUL BOBHATE, RAYMOND CHO, HIRANAVA DAS, SHARIQUE HUSSAIN, HAMZA KARACHIWALA, NOLEN SCAIFE, BYRON WRIGHT, and KEVIN BUTLER, University of Florida
WILLIAM ENCK, North Carolina State University
PATRICK TRAYNOR, University of Florida

The security research community has invested significant effort in improving the security of Android applications over the past half decade. This effort has addressed a wide range of problems and resulted in the creation of many tools for application analysis. In this article, we perform the first systematization of Android security research that analyzes applications, characterizing the work published in more than 17 top venues since 2010. We categorize each paper by the types of problems they solve, highlight areas that have received the most attention, and note whether tools were ever publicly released for each effort. Of the released tools, we then evaluate a representative sample to determine how well application developers can apply the results of our community's efforts to improve their products. We find not only that significant work remains to be done in terms of research coverage but also that the tools suffer from significant issues ranging from lack of maintenance to the inability to produce functional output for applications with known vulnerabilities. We close by offering suggestions on how the community can more successfully move forward.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Automated static analysis**; **Dynamic analysis;**

Additional Key Words and Phrases: Android, application security, program analysis

ACM Reference Format:
Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *droid: Assessment and evaluation of Android application analysis tools. ACM Comput. Surv. 49, 3, Article 55 (October 2016), 30 pages.
DOI: http://dx.doi.org/10.1145/2996358

This work was supported in part by the U.S. National Science Foundation under grants CNS-1222680, CNS-1253346, CNS-1464087, CNS-1464088, CNS-1513690, CNS-1526718, and CNS-1540217. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Authors' addresses: B. Reaves, J. Bowers, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, and P. Traynor, Department of Computer & Information Science & Engineering, E301 CSE Building, PO Box 116120, Gainesville, FL 32611; emails: {reaves, jdbowers, ooa0002, rabo, raymond.cho, hiranava, sharique88, hskarachiwala, scaife, bwright611, butler}@ufl.edu, traynor@cise.ufl.edu; S. A. Gorski III and W. Enck, Department of Computer Science, 890 Oval Drive, Campus Box 8206, Raleigh, NC 27695-8206; emails: {sagorski, whenck}@ncsu.edu.

DOI: http://dx.doi.org/10.1145/2996358

1. INTRODUCTION

Android has become the most widely used mobile operating system in the world [Northcraft 2014]. Run on some of the most and least expensive smart phones, this platform is supported by a series of markets that boast over 1 million applications providing functionality ranging from social networking to travel and finance [Statistia 2015]. As such, this platform now has access to some of its users' most sensitive operations and data.

Android has accordingly become a popular target for analysis by the security research community, but not simply for the preceding reasons. code. This information allows program analysis tools to more easily produce accurate results. When combined with the massive number of applications available for analysis, researchers have had significant opportunities to produce valuable real-world results. Unsurprisingly, the past half decade has seen the publication of hundreds of papers on the topic of Android application security.

Given this sheer volume of work, the systems security research community is at an important crossroads. If we have addressed the major intellectual problems facing this platform and produced artifacts capable of helping others improve the security of their applications, our work is largely finished. Alternatively, if we have neglected areas or our tools are insufficient, the work must continue. This issue simply cannot be resolved without a meaningful systematization of the techniques, frameworks, and artifacts generated by this extended community.

In this article, we attempt to answer two critical questions for the systems security community interested in program analysis of Android applications. First, which areas have been addressed by prior security research and which still require attention? We attempt to answer this question through a comprehensive classification of security research in this space. Second, because we as a community value the analysis tools and artifacts provided by many of these efforts, which tools are available and can be readily used by the application development community to improve the security of their outputs? We attempt to answer this question by performing extensive testing on a representative set of popular Android application analysis tools. In doing so, this work makes the following contributions:

—*Systematize Android-specific analysis challenges*: The tight integration of Android applications with the system runtime causes even sound program analysis algorithms to become unsound without careful modeling and abstraction. Furthermore, the platform introduces new abstractions that change the threats that must be considered by program analysis tools seeking to identify vulnerabilities and dangerous behavior. Accordingly, we provide readers with the context to understand how program analysis in Android is different from analysis in other domains.

—*Characterize application security analysis efforts*: With more than half a decade of research on this topic, our community has tackled a wide array of problems as they relate to Android program analysis. However, reasoning about which problems are being addressed and the areas that lack attention is difficult. This thrust of our work offers a comprehensive classification of Android application security analysis efforts published at 17 different venues since 2010.

—*Evaluate the state of Android tools*: The systems community often builds on the tangible artifacts of our peers' research, making the release of operational tools critical to our success as a discipline. Such tools are also critical to ensuring independent validation of published results. As such, we discuss our experience working with seven popular tools, measure their ability to analyze a range of applications, and then determine their ability to be used by application developers and auditors to improve the security of their products. In general, we find that the tools are challenging to use, and when they do work, they often produce output that is difficult to interpret.

It is critical to note that we do not analyze every Android-related paper ever written; rather, we attempt to systematize only those papers that look to improve the security strictly of Android applications and were published at a major venue. We believe that it would be worth systematizing these other efforts to make an Android operating system, market ecosystem, and so forth, but these will require separate dedicated efforts from this significant undertaking. Indeed, prior efforts have focused on developing a taxonomy of Android security research [Sufatrio et al. 2015], examining the research devoted to appification challenges [Acar et al. 2016], and investigating the state of the art of input generation for dynamic testing [Choudhary et al. 2015].

The remainder of this article is organized as follows. Section 2 provides a background into Android-specific challenges for analysis. Section 3 briefly describes the methods used to develop our survey. Section 4 conducts an in-depth analysis of the state of the art in Android program analysis tools and the classes of problems they claim to address. Section 5 presents the methodology we used to select and evaluate our candidate research artifacts. Section 6 presents the results of our analysis and offers takeaways for other researchers based on our observations. Section 7 offers lessons learned about the state of available analysis tools for Android and suggests ways to address open challenges. Section 8 provides concluding remarks.

2. ANDROID-SPECIFIC ANALYSIS CHALLENGES

We begin our analysis by highlighting the unique challenges in analyzing Android applications. The first challenge deals with how applications are distributed. Although Android applications are written in Java, the application package binary does not contain Java bytecode. The Android SDK includes an additional step that transforms Java bytecode into DEX bytecode, which is designed to run in the Android-specific Dalvik VM. Given the abundance of program analysis tools and frameworks designed to operate on Java bytecode, the research community has created retargeting tools such as ded [Enck et al. 2011], Dare [Octeau et al. 2012], and dex2jar [dex2jar 2015] to transform DEX bytecode into Java bytecode.

Note that retargeting DEX bytecode to Java bytecode is significantly different from decompiling DEX bytecode to Java source code. Directly decompiling DEX bytecode often results in hard-to-read source code with infinite while loops and break statements. Therefore, ded and Dare use Soot [Vallée-Rai et al. 1999; Bartel et al. 2012a] to optimize the Java bytecode before decompiling to source code. Although the Soot optimization produces much more readable Java source code, it significantly increases the decompilation time. Despite the ability to disable Soot optimization in ded and Dare, slow performance of the default configuration has steered researchers toward other tools.

With tools such as Dare and the recent inclusion of a DEX front end for the Soot analysis framework [Lam et al. 2011; Bartel et al. 2012a], one might wrongly assume that Android's program analysis challenges have been solved. This assumption is far from true. The Android runtime environment presents many practical challenges for both static and dynamic program analysis tools. The challenges are akin to the soundiness argument [Livshits et al. 2015], which highlights the practical limitations of static program analysis on Java code (among other languages). Android's runtime environment makes some of these aspects even harder through pervasive use of difficult-to-analyze language features, as well as new abstractions that pose similar practical challenges. The remainder of this section is devoted to describing these practical challenges that we colloquially call *Androidisms*.

2.1. Android Abstractions

A key difference between Android applications and Java applications is the lack of a *main()* function that begins execution. Instead, the Android runtime automatically invokes parts of the application code based on system and user events. Program

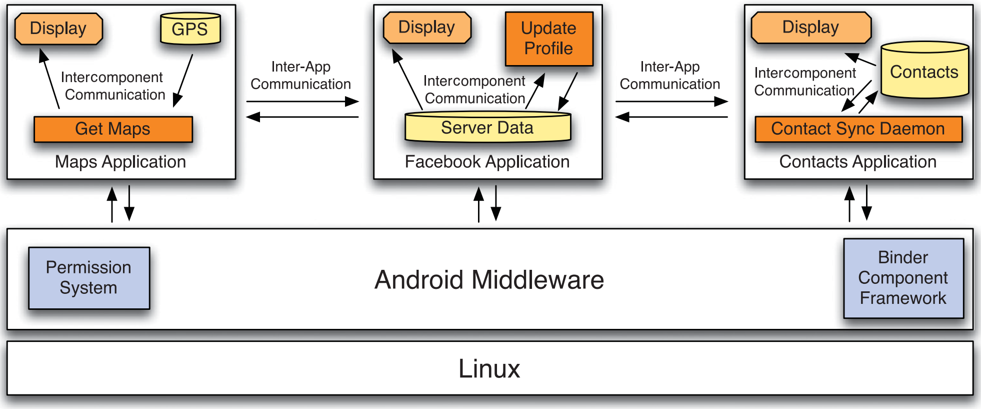

Fig. 1. How Android application components interact within the same app, with other apps, and with the Android framework.

analysis tools tracking control flow and dataflow must be sensitive to Android's runtime abstractions. Specific details of these challenges can be found in the papers on FlowDroid [Fritz et al. 2014], Amandroid [Wei et al. 2014], and DroidSafe [Gordon et al. 2015]. These three tools address many, but not all, of the challenges.

2.1.1. Application Lifecycle. The Android SDK requires developers to partition application functionality into four conceptual component abstractions: activity, service, broadcast receiver, and content provider. Prior work has described these abstractions in detail [Enck et al. 2009]. For the purposes of this article, it is important to observe that each component type is a Java class that is extended by application code. The runtime invokes specific methods (e.g., *onCreate()*) in response to system events. Activity components have the most complex lifecycle. Activity components define interfaces to create, start, pause, resume, stop, and destroy the component instance. Each of these interfaces in each component is an entry point that must be considered during program analysis. Fortunately, nearly all application components must be statically defined in the package manifest file. The only exception is dynamically created broadcast receiver components, which are often overlooked by program analysis tools.

2.1.2. Inter-Component Communication. The Android runtime provides abstractions for components to interact with one another, whether or not they are in the same or a different application. An illustration of this procedure is provided in Figure 1. Intent messages are the primary abstraction used by third-party applications. An intent message can be explicitly addressed to a target component. However, intent messages can also be implicitly addressed using action strings. Developers use the application manifest to define intent filters for components, which indicate the set of action strings the component can handle. The Android runtime automatically resolves the target component based on the set of currently installed applications and a complex set of resolution rules. Resolution sometimes involves the end user by prompting for a preferred application. The rules also include other string values, such as a category and MIME type. Further complicating analysis, third-party developers can add to the list of predefined action strings. In all cases, these strings (action string, component name, category, MIME type) may be defined as constants in the code or be assigned based on input from other intercomponent communication (ICC), the network, or files. These strings may also be encrypted or obfuscated in some way to deter reverse engineering. Correctly matching the intent message addressing values to intent filters is a nontrivial task that must incorporate system state.

In addition to resolving ICC control flow during program analysis, tools must resolve dataflow. Intent messages can transfer sensitive information not only through action strings but also through a field called *extras*. This field contains an instance of a *Bundle* class, which is simply a key-value storage structure. Tools that calculate dataflow dependencies (e.g., taint tracking) must track the key associated with each value in the Bundle. Furthermore, since Bundles may be passed between applications, the program analysis tool must re-create the app and system state when analyzing the called component.

2.1.3. System Callbacks. Applications frequently interact with the Android runtime. There are two types of runtime functionality: (1) functionality that executes in the same process as the application and (2) functionality that executes in system processes. Both types present challenges for program analysis. Android heavily uses threads, which is a known challenge for Java program analysis [Rinard 2001]. Android exacerbates the challenge by providing new APIs for thread-like functionality. For example, the *android.os.Handler* class allows message objects in one thread to be passed to a message queue in another thread. Because the Handler abstraction uses a generic *Message* type, dataflow analysis must track additional context sensitive information.

Tracing functionality through system processes is equally, if not more, challenging. All Android interprocess communication (including ICC) is built on top of the Binder IPC mechanism. Binder allows applications to pass object references that are used for callbacks from other processes. For example, to get geographic location, an application registers a *LocationListener* object with the system's Location Manager Service. Based on the request parameters, the Location Manager Service will automatically call the *onLocationChanged()* method of the listener object on specified intervals. Tracing such control flows requires understanding the behavior of the underlying framework. Because this framework contains thousands of methods, most static analysis tools that need to analyze flow through the framework use manually described models of the behavior described as stub functions [Gordon et al. 2015; Fritz et al. 2014; Wei et al. 2014; Kim et al. 2012; Feng et al. 2014; Lu et al. 2012].

2.1.4. UI Callbacks. Most Android applications are designed around a graphical user interface. In addition to the activity component lifecycle challenges discussed earlier, program analysis tools must account for control flow and dataflow originating from UI widgets. When an application wishes to receive click events for a widget, it registers an *OnClickListener* object for that widget with the Android runtime. When the widget is pressed, the *onClick()* method of the listener object is called. These click-based callbacks and the other types of UI callbacks provided by the Android API are all entry points that must be considered in the analysis. Furthermore, it is possible for an application developer to create a custom widget that specifies its own set of custom events and UI callbacks in addition to those provided by the Android API. As these custom callbacks are also entry points that must be considered in the analysis of applications, the analysis of Android applications becomes even more complex. Additionally, the majority of applications specify user interfaces and their callbacks using an XML resource file, from which the interface is generated at compile time. Several works, including UIPicker [Nan et al. 2015] and SUPOR [Huang et al. 2015b], have used this resource file as a starting point.

2.1.5. Persistent Storage. Android is designed to operate on a range of devices with different hardware capabilities, with a special focus on devices with limited memory. Android components (building blocks of apps) can be interchanged at any moment, like when a user switches between applications. As a result of this capability, any component can be suspended at any moment. In addition, applications frequently save values into persistent storage. Program analysis seeking to track dataflows must trace values into and out of persistent storage. Android persistent storage includes files, shared preferences, SQLite databases, and content provider components. Shared preferences are a key-value storage that is shared among all components in an application. Content provider components present a relational database interface for sharing information between applications. They are typically backed by SQLite database files.

2.1.6. Rich API Features. The Android runtime simplifies application development by providing a rich set of APIs. In addition to tracking control- and dataflows through APIs that use native code, a program analysis tool might wish to designate certain features as sources or sinks for analysis. Achieving complete coverage of such APIs is practically challenging due to the sheer number of API calls.

2.2. Java Abstractions

Even without the analysis challenges created by Android's specific design nuances, the analysis of Java code is still challenging. Because Java uses several difficult-to-analyze language features, constructing an exact call graph of any Java program is a well-known hard problem that makes modeling exact program behavior difficult. Furthermore, no existing algorithms can construct a precise call graph for any Java program while remaining sound. Further still, increasingly precise but still unsound call graphs often require large amounts of memory and computing power to construct, which is detrimental to performance. Sound but imprecise call graphs are easy to construct but include edges to methods that are technically unreachable, which can increase the call graph's size dramatically and affect analysis performance as well. Thus, constructing a call graph for analysis purposes, and therefore the analysis itself, is ultimately a trade-off between precision, soundness, and performance. Ultimately, the difficulty in Java code analysis boils down to four major Java language features.

2.2.1. Inheritance and Polymorphism. Java code relies heavily on a class hierarchy structure to reduce code repetition and enable more robust code reuse than non–object-oriented languages (e.g., C). In other words, every class in Java ultimately has a parent class for which it derives some of its methods and fields. These methods and fields can in turn be overridden by new methods and fields of the same name in the child class. Moreover, a developer can specify that a class has certain method signatures by writing an interface, leaving the actual method body code up to the developer of a class that implements that interface. Further still, in another attempt to encourage code reuse, Java allows for polymorphic types (i.e., generics) in classes and methods, thus allowing a class or method to be reused with many different types of objects.

As a result of all various language features of Java that tend to hide an object's underlining class, static resolution of method invocation and field access statements to proper method/field targets are difficult. When a method is invoked on an object, the object could be declared as a type that is a superclass of the object's actual underlining class or as an interface type. Both situations make it difficult to determine an actual invoke resolution target, as the actual underlining class of an object is determined at runtime and hidden by the type abstractions of the code. Thus, making security inferences about app behavior is subsequently more difficult because the actual code executed at runtime may vary.

To approximate the method invocation targets, many different techniques have been proposed. Some such as class hierarchy analysis (CHA) [Dean et al. 1995] are sound but imprecise, which often detrimentally effects performance. Others are precise but not necessarily sound, such as the many precision-increasing variations on points-to sets [Lhoták and Hendren 2003; Smaragdakis et al. 2011, 2014]. Moreover, as precision increases, so does the amount of memory and processing power required. This ultimately leads to tools like FlowDroid [Fritz et al. 2014], Amandroid [Wei et al. 2014], and DroidSafe [Gordon et al. 2015], which are precise but not necessarily scalable to large, complex applications or will not run on standard desktop hardware.

2.2.2. Reflection. Reflection allows application developers to dynamically inspect classes, interfaces, methods, and fields, as well as create objects, invoke methods, and access object fields without the class, method, or field name being known at compile time. This is generally performed by providing the class, method, or field name as a string to the related reflection method. Unfortunately, as the string values are decided at runtime and may be set based on input, producing an exact call graph of a program using reflection is a well-known hard problem [Livshits et al. 2005].

Many Android apps and Java programs often rely on reflection for tasks such as API compatibility checking and code obfuscation. Indeed, it is recommended on the Android Developers Blog for application developers to perform backward compatibility checks using reflection [Android Developers Blog 2009]. Although it is a fairly common practice for malicious code to hide its behavior from analysis tools through the use of reflection [Rasthofer et al. 2016], reflection is used by nonmalicious application developers for obfuscation purposes as well. Unfortunately, soundly analyzing reflected code is such a troublesome task that most tools neglect reflected code altogether, which in many cases can affect the soundness of the analysis.

2.2.3. Dynamic Loading. Like other languages, in Java it is possible to dynamically load code. This is generally performed in Java by dynamically loading classes that are usually identified using a string representation of the class name. As a developer can write his or her own class loader, the string used to identify and load classes can in fact be anything [Rasthofer et al. 2016; Livshits et al. 2005]. Additionally, Android allows applications to dynamically load DEX bytecode using the `dalvik.system.DexClassLoader` object [Android Developers Blog 2011]. This interface is required because DEX files are limited to 65,000 methods [Android Developer Documentation 2015], and some applications (e.g., games) include many Java library dependencies. However, this feature has also been used suspiciously by ad libraries [Grace et al. 2012].

The challenges of dynamic loading are well-known hard problems for Java program analysis. Similar to reflection, dynamically loaded code is often neglected in the analysis of Java code, which can affect the soundness of the analysis.

2.2.4. Native Code. Although Java code is ultimately easier to analyze statically despite the difficulties mentioned previously, it has one major setback: the support for the execution of native code. Native code can be invoked in any number of ways from Java code, including the execution of a binary file from a command as a subprocess, a call to a native method through the Java Native Interface (JNI), and in Android even through callbacks to native activities. Moreover, native code can make callbacks into the Java code, a process typically conducted through the JNI. On top of being able to perform all operations available to normal non-Java programs, native code executed from Java code on Android runs within the same memory space as the Java code. Thus, native code has access to all Java code and data and can freely modify any Java-related information of an application, making the static analysis of just the Java code unsound. Even if the native code is not inherently malicious, execution paths can easily be hidden from the majority of Android and Java analysis tools by simply running the execution paths through native code, which is notoriously difficult to analyze and often ignored by most tools.

2.3. Dynamic Analysis Considerations

Many of the preceding challenges are specific to static program analysis. The challenges result because the static program analysis tools do not have knowledge of the system state. In contrast, dynamic program analysis tools sidestep many of these challenges because the analysis is performed in a running system. However, dynamic program analysis, in general, is inherently limited in code coverage.

The first critical decision when designing a dynamic analysis tool is how to monitor and collect data about an app's runtime behavior. Dynamic program analysis tools can use one of several analysis approaches. The first approach is generating execution traces for subsequent analysis. These traces can happen at any level in the software stack, including native processor instructions, virtual machine instructions, system calls, Android API calls, or even simply network traffic. These traces provide a full view of application behavior from the point of view of the tracing location, but all tracing locations present a challenge of associating low-level behavior (e.g., individual machine instructions) with high-level semantics of interest (e.g., sending a premium SMS). The second approach that tools interested in dataflow analysis can use is dynamic taint analysis. Taint analysis tracks the flow of data from any number of deferred sensitive sources to defined sensitive sinks. Although research has found this to be performant and effective, this analysis is ultimately limited to questions of dataflow. Third, statically or dynamically rewriting apps for analysis is a common approach in traditional binaries [Nethercote 2004] but is not one that has been applied to Android. App rewriting has been popular for implementing enhanced policy enforcement [Georgiev et al. 2014; Shekhar et al. 2012; Bhoraskar et al. 2014; Davis and Chen 2013], so this rewriting for dynamic analysis may be possible.

Regardless of analysis approach, dynamic program analysis requires effective test input generation. The Android SDK includes an input fuzzing tool called *Monkey*. However, Monkey can get stuck in activity components and not explore the rest of the application. Therefore, dynamic analysis tools using Monkey may need to run the same application multiple times to achieve good results. Android applications also commonly present input barriers to functionality. For example, some applications present a login screen before the application may be used. Running the application multiple times will not overcome such challenges. Fortunately, login screens often include a link or button to register for an account. Therefore, incorporating text analytics into test input generation can help produce better coverage but will still likely not produce full code coverage in many cases.

Certain analysis motivations (e.g., malware analysis) must also consider the impact of system events. For example, when a new SMS message arrives, the Android runtime will send intent messages to all broadcast receiver components that defined an intent filter for the SMS_RECEIVED action string. To exercise these code paths, the dynamic program analysis tool must simulate such system events. Somewhat related to simulating system events, some applications use antireversing measures such as not executing certain functionality if the application is run on an emulator. Such techniques are commonly used by malware but are also benign applications attempting to protect intellectual property. Emulator detection is not the only difficulty that apps can present to dynamic analysis. Runtime code integrity checks (potentially obfuscated) and root checks (to prevent debugging on rooted platforms) are additional challenges.

Finally, Android's runtime functionality is distributed among libraries in the application under test and system applications. As mentioned previously, control- and dataflow frequently cross application and system boundaries. Therefore, dynamic taint analysis tools must instrument not only the application under test but also the system applications.

3. SURVEY METHODOLOGY

During our preliminary research, we thought it only fitting to gather all related papers in the space. To ensure that we had a comprehensive repository of Android publications,

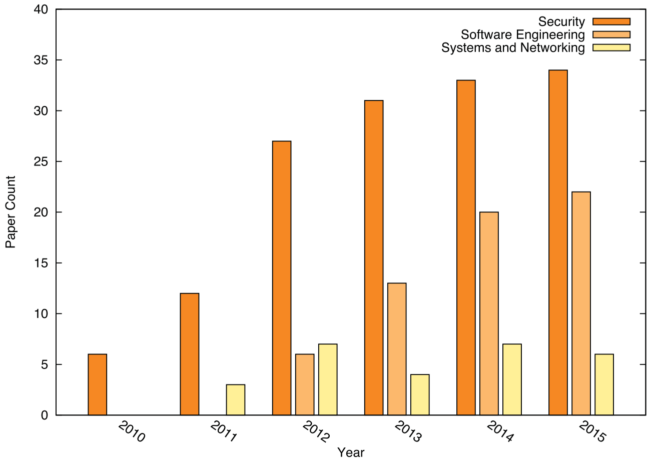

Fig. 2. Number of Android security papers presented in software engineering, security, systems. and networking venues has risen sharply over the past 6 years.

we gathered all of the approximately 300 Android-related papers from 17 security and software engineering conferences since 2010. The complete list of conferences is ACM CCS, ACSAC, ACM WiSec, USENIX OSDI, ACM MobiSys, ACM SPSM, IEEE MoST, ACM/IEEE ICSE, ACM PLDI, ACM CODASPY, ACM POPL, IEEE S&P, USENIX Security, ISOC NDSS, ACM MobiCom, ACM/IEEE ASE, and ACM FSE. Figure 2 shows the contribution of the security, software engineering and programming languages, and systems and networking research communities to this collection. Note the steady increase in published Android work each year. We chose to only focus on academic works because academic tools are generally easier to obtain and test, do not require expensive licensing or long and involved procurement processes, and most importantly have clearly stated goals and capabilities.

Having finished our collection of papers, we then filtered out papers unrelated to security or Android application analysis. Because Android has seen an extensive amount of research from the security community and beyond, a narrow focus was important to adequately treat the complex topics involved. We limited our scope to static analysis and dynamic analysis techniques. Some tools have aspects of both static and dynamic analysis. These "hybrid" tools tend to have a strong emphasis on one analysis class with additional techniques from the other class to improve analysis. We have classified these tools into the relevant static or dynamic analysis sections based on the predominant technique. Specifically, we selected papers if they could affirmatively answer the following question: Does this published research paper provide an artifact (e.g., tool) that automatically produces a result concerning the security of one or more Android applications? A corollary question was the following: Could this artifact be useful to the developer or auditor of a single app or a small set of apps? We note that some artifacts may be applicable only to developers or only to third-party analysts. We consider either use case equally valid, but we note that most analysis techniques will be applicable to either case.

Many papers, although beneficial to the Android community, discussed topics out of scope. For example, some papers [Chakradeo et al. 2013] produce a relative ordering of applications in terms of security behaviors but cannot produce a semantically meaningful analysis about a single app. These tools are useful in the context of a whole market but less useful to developers. Other papers use static analysis techniques to rewrite apps to improve security aspects of application libraries [Georgiev et al. 2014; Shekhar et al. 2012] or to enhance the expressiveness of a permission model [Bhoraskar et al. 2014] or user-specified policy [Davis and Chen 2013], or to repair already discovered vulnerabilities [Zhang and Yin 2014]. One paper statically analyzes high-risk methods in the Android operating system to identify potential system vulnerabilities but does not analyze apps [Huang et al. 2015a].

The end result of this paper analysis was a focused study in Android application security spanning the past 6 years of research.

4. STATE OF THE ART

We are aware of the many challenges that Android apps and Java programs in general pose to static analysis tools; however, we do not have a clear view of the challenges that have been addressed by prior security research and the challenges that may still need attending. To answer these questions, we performed a comprehensive review of Android static or dynamic analysis techniques and characterized them further in Tables I and II, respectively. The following section discusses the findings of this review process.

4.1. Static Analysis

Static analysis approaches comprise the majority of application security research—36 papers in our survey—outnumbering dynamic approaches by more than a factor of 4.

Table I shows our characterization of the static analysis space. Static analysis is a common approach for several reasons. First, Java is a high-level language and is easier to analyze statically than other languages. Additionally, Dalvik VM code (DEX) is also a relatively high level instruction set that retains many of the semantics of Java. The easy analysis of DEX has been a boon to researchers because it means that tools can be developed and tested on a large corpus of real-world applications. Examining more than 100,000 applications is not uncommon in many of these papers.

Whereas Java and DEX are easier to parse and analyze than other environments, the Android application architecture still presents several challenges that must be addressed for effective static analysis. These include classic static analysis challenges and Android-specific issues.

4.1.1. Common Java Static Analysis Challenges. The first general static analysis challenge—faced by all analyses of Java-based code—is how to properly reason about reflection. This is a difficult problem, and most tools do not address it. Five tools are able to handle reflection involving string constants: DroidSafe [Gordon et al. 2015], Stowaway [Felt et al. 2011], FlowDroid [Arzt et al. 2014], DroidSift [Zhang et al. 2014a], and Scandal [Kim et al. 2012]. DroidSafe and Stowaway, however, take it a step further and attempt to handle reflection involving strings that are constructed programmatically. DroidSafe for instance, attempts to account for reflection in application code using a method similar to those found in two previous works [Smaragdakis et al. 2015; Livshits et al. 2005]. Smaragdaki developed an approach to analyze reflection with high empirical soundness, and Livshits developed an algorithm that approximates the targets of reflective calls by using points-to information. Alas, these works make several assumptions, such as the assumption of well-behaved class loaders, which cannot be guaranteed when analyzing malicious code. Moreover, no static analysis tools in our survey attempt to handle reflection involving encrypted/obfuscated strings, a technique commonly used (especially by malware) to obfuscate and protect code against static analysis attempts. However, Harvester [Rasthofer et al. 2016] is specifically designed to handle the analysis of malicious code that uses reflection. Unfortunately, as Harvester relies on dynamic analysis to resolve reflection code, its analysis may not always be sound. Another classic static analysis challenge is the dynamic loading of code at runtime. In Android, this takes the form of dynamic DEX

Table I. Static Analysis

		FlowDroid [Arzt et al. 2014]	DroidSafe [Gordon et al. 2015]	Amandroid [Wei et al. 2014]	MalloDroid [Fahl et al. 2012]	DroidSift [Zhang et al. 2014a]	Epicc [Octeau et al. 2013]	AdRisk [Grace et al. 2012]	UI Deception [Bianchi et al. 2015]	CryptoLint [Egele et al. 2013]	CLAPP [Fratantonio et al. 2015]	CHEX [Lu et al. 2012]	Blue Seal [Shen et al. 2014]	MassVet [Chen et al. 2015]	Stowaway [Felt et al. 2011]	PScout [Au et al. 2012]	Drebin [Arp et al. 2014]	DIDFail [Klieber et al. 2014]	Anadroid [Liang et al. 2013]	AppProfiler [Rosen et al. 2013]	AsDroid [Huang et al. 2014]	COPES [Bartel et al. 2012b]	SEFA [Wu et al. 2013]	AAPL [Lu et al. 2015]	Intent Input Validation Vulnerability Detection [Gallingani et al. 2015]	DroidEagle [Sun et al. 2015]	User-Aware Privacy Control Analysis [Xiao et al. 2012]	User-Centric Dependence Analysis [Elish et al. 2012]	ViewDroid [Zhang et al. 2014b]	DroidJust [Chen and Zhu 2015]	SCANDAL [Kim et al. 2012]	WeChecker [Cui et al. 2015]	AppIntent [Yang et al. 2013]	WiFiLeaks [Achara et al. 2014]	Comdroid [Chin et al. 2011]	Apposcopy [Feng et al. 2014]
Reflection	String constants	●	●	○	○	●	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○
	Constructed strings	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Obfuscation/encryption handling	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Misc	Dynamic DEX file loading	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Multiapp support	○	○	○	○	○	●	○	●	○	○	○	◐	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
	Native code	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
	Extensible	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○
	Publicly available	●	●	●	●	○	●	○	○	○		○	●	○	○	●	○	●	●	●	○	○	○	○	○		●	○	○	○	○	●	○	○	○	○
	Benign only/malicious/dual use	D	D	D	D	M	D	B	D	B	D	B	B	M	B	B	M	D	M	D	M	B	B	D	B	M	M	M	M	D	D	B	B	M	B	M
Vulnerabilities Detected	Detect sensitive information leaks	●	●	○	○	○	○	●	○	○	○	●	●	○	○	○	○	●	●	○	○	○	●	●	○	○	●	●	○	●	●	●	●	●	○	●
	Authentication	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Cryptography	○	○	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Data validation	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Intent spoofing	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○
	Unauthorized intent receipt	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	◐	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○
	Configuration and deployment	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Permission misuse	○	○	○	○	○	○	○	○	○	○	○	●	○	●	●	◐	○	●	○	○	●	●	○	○	○	○	○	○	○	○	●	○	○	○	○
	Plagiarism detection	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○
	Other	○	○	○	○	•	○	○	•	○	•	○	○	○	○	○	○	○	○	•	•	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Technique	Taint analysis	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	●	●	○	●	○	○	●	○	●	●	○	○	●
	Reachability analysis	○	○	○	○	○	○	●	○	○	○	●	●	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○
	Symbolic execution	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	●	○	○	○
	APK rewriting	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Intercomponent flow graph	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Intercomponent flow analysis	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Heuristics	○	○	○	●	○	○	○	●	●	○	○	○	○	●	●	○	○	○	○	○	○	●	○	○	○	○	●	○	○	○	○	○	●	○	●
	View comparison	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○
	Other	○	○	○	○	•	○	○	•	•	•	○	○	○	○	○	•	○	•	•	•	•	○	•	○	○	○	○	○	○	•	○	○	○	○	○
Frameworks	Soot	●	●	○	○	●	●	○	○	○	○	○	●	○	○	●	○	●	○	○	○	●	○	○	●	○	○	●	○	●	○	●	●	○	○	●
	WALA	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Androguard	○	○	○	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
	Baksmali	○	○	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○
	Other	○	○	•	○	○	○	○	○	○	○	•	○	○	•	○	○	•	○	•	○	○	○	•	○	○	•	○	○	○	○	○	○	○	•	○

loading and is often used to hide code from static analysis attempts. Unfortunately, none of the tools surveyed addressed this problem. Similarly, Android provides support for interacting with native code through several interfaces. However, all tools surveyed fail to properly address flows through native code as well.

4.1.2. Modularity Challenges. Although issues of runtime code loading and class resolution are common across static analysis, Android's unique emphasis on interapplication communication means that some meaningful security analyses can only be performed in the context of multiple applications. Furthermore, because of the modularity of Android, apps communicate with the system through these same communication channels to perform a significant portion of their functionality. Essentially, code in the application package does not comprise the entirety of all code executed by the app. Unfortunately, most analysis tools still treat apps as individual (isolated) entities, except for system interaction. They do not model or account for interapplication communication, so they often do not get full insight into the full range of possible app behaviors even if they do model ICC. In particular, only four have support for interapplication analysis: Epicc [Octeau et al. 2013], UI Deception [Bianchi et al. 2015], DidFail [Klieber et al. 2014], and DroidJust [Chen and Zhu 2015]. A fifth tool, Blue Seal [Shen et al. 2014], requires dynamic analysis for interapp control.

4.1.3. Analysis Goals. The most popular topic for tools to address is sensitive information leakage, with 17 tools addressing this issue. Although Android is certainly not the only platform with concerns about information leakage, the high concentration of sensitive information available on mobile platforms makes this problem a high priority. Despite significant investments into tools that address this issue, the Android-specific and general Java-related analysis challenges discussed in Section 2 still cause researchers significant hardships in developing tools that produce sound and precise results in a practical amount of time. Epicc [Octeau et al. 2013] for example, although highly advanced in its handling of ICC, is still unable to properly reason about ICC through content providers. Moreover, Epicc cannot analyze code containing reflection, dynamic DEX loading, or native code. The second most popular topic for tools to address is the open problem of permission misuse (e.g., the overpermissioning of Android apps), which is covered by eight tools. In a way, this problem is linked to the issue of sensitive information leakage (with several tools handling both), as overpermissioned applications have access to more sensitive information. Other popular topics related to the issue of sensitive information leakage are those of intent spoofing and unauthorized intent receipt, each covered by four papers, as they are often issues exploited by malicious apps.

Several other reoccurring topics for tools include those of cryptography misuse and plagiarism detection (i.e., the identification of modified and/or repackaged legitimate apps, a common technique used by many malicious apps). For the former, Fahl et al. [2012] and Egele et al. [2013] design specialized tools to statically detect improper uses of cryptography. For plagiarism detection, MassVet [Chen et al. 2015], DroidEagle [Sun et al. 2015], and ViewDroid [Zhang et al. 2014b] attempt to address the issue in an effort to stem the rise of mobile malware.

Finally, other tools have looked into less studied issues. Several look at issues related to whether sensitive functionality is warranted by the current app context to detect stealth behaviors indicative of malicious intent (AsDroid [Huang et al. 2014] and AAPL [Lu et al. 2015]). Clapp [Fratantonio et al. 2015], on the other hand, provides a loop analysis framework that is designed to detect inputs that could lead to an app denial of service attack.

Note that unlike the dynamic analysis tools that we surveyed, the majority (27) of static analysis tools were meant to be applied to benign apps, whereas only 9 tools were applicable only to malicious apps. We note that 14 tools were "dual use," meaning that they provide insights into both malicious and benign applications. This is shown later in Figure 4.

4.1.4. Analysis Techniques. The ultimate goals of many of the tools are closely related; however, the techniques used to achieve these goals vary. For information leaks and permission misuse, reachability (6 tools) or taint analysis techniques (10 tools) are common. Indeed, almost every paper that covers these topics uses some variant of these techniques. AAPL [Lu et al. 2015], for example, although fundamentally a static taint tracking tool, expands on the basic taint tracking premise by building the interprocedural constant evaluation and concrete object type inference. Flow analysis (4 tools) and flow graph generation (1 tool) are also popular and sometimes supplementing techniques for these areas. Moreover, slicing-based dataflow analyses have been used in the 2 papers that share a common code base (CryptoLint [Egele et al. 2013] and UI Deception [Bianchi et al. 2015]). Further still, several papers have relied on simple call graph analysis and/or even simpler scanning techniques to search for the presence of calls to sensitive system APIs (Drebin [Arp et al. 2014], AppProfiler [Rosen et al. 2013], Copes [Bartel et al. 2012b]). Finally, 10 papers relied on some combination of lightweight analysis and simple heuristics to derive analysis results.

4.1.5. Frameworks. Because Android apps are based in Java, a common theme in Android static analysis is to adapt Java-based analysis frameworks for Android analysis. Twenty-seven tools used one or more public frameworks. Soot [Vallée-Rai et al. 1999; Bodden 2012; Padhye and Khedker 2013; Lam et al. 2011; Reps et al. 1995; Sagiv et al. 1995] was the most popular with 14 tools, followed by the Android lightweight static analysis tool Androguard [2012] with 5 tools, Baksmali [2009] with 5 tools, and Wala [2006] with 3 tools.

The reuse of these frameworks and creation of new frameworks is a boon for the community, as shared infrastructure makes collaboration and code reuse easier, saving precious development time in future research. Unfortunately, although many researchers build on the work of the Java community, only 3 tools were developed to be extensible—Amandroid [Wei et al. 2014], Anadroid [Liang et al. 2013], and Scandal [Kim et al. 2012]. Furthermore, only 12 of the 36 tools were publicly available, meaning that it is even more difficult to verify results or build on these tools.

4.2. Dynamic Analysis

We find that the state of the art in Android dynamic analysis differs starkly from the state of static analysis. Table II shows our characterization of the dynamic analysis space. We survey a total of nine tools.[1] Half of these are publicly available, although only one tool is designed as an extensible framework.

Immediately, one can see that dynamic analysis has simply not seen the attention or investment that has been lavished on static analysis approaches. We hypothesize that this is the result of the following reasons. The first reason is simply that static analysis techniques are studied well outside the field of computer security, with multiple fields having researchers interested in such topics—from programming language researchers to compiler developers to software engineers. Another reason that dynamic analysis is less popular is that researchers may have the perception that static approaches (although not perfect and often prone to false positives) will give better results owing to the fact that static analysis approaches can process all code

[1] LazyTainter [Wei and Lie 2014] is not shown in the table. This article optimizes memory performance of TaintDroid [Enck et al. 2014] while not changing its security-relevant functionality or analyses.

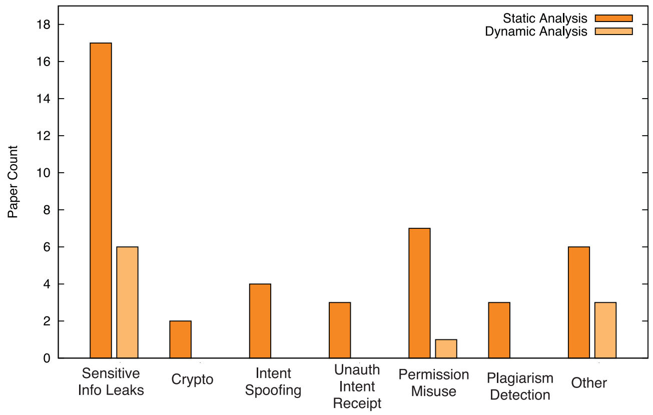

Fig. 3. Information leaks are by far the most common vulnerability class detected by the systems we survey.

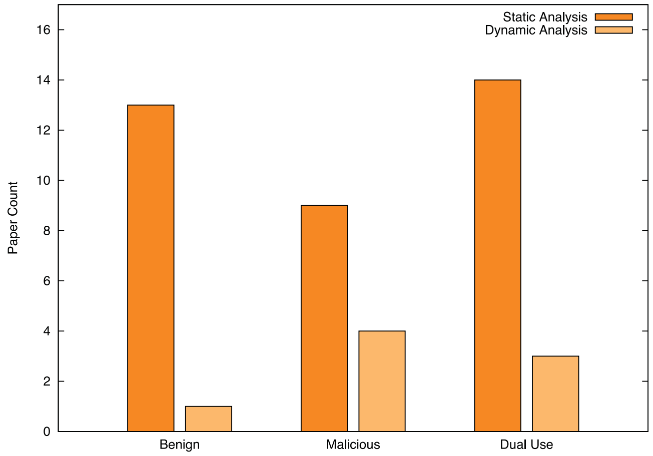

Fig. 4. Dynamic analysis approaches focus more on malicious applications, whereas static analysis tools focus more on benign and dual-use applications.

paths with a sufficiently sound technique (subject to the previously discussed caveats of reflection, mobile code, and native code).

4.2.1. Analysis Goals. We believe that dynamic analysis of Android applications is an area ripe for innovation. To date, out of all tools that we survey, six are designed to detect malicious activity, seven detect information leaks, and four detect both. This is shown in Figure 3. These tools also detect root compromise (PREC [Ho et al. 2014], AppsPlayground [Rastogi et al. 2013]), advertising fraud (MAdFraud [Crussell et al. 2014]), and permissions abuse (VetDroid [Zhang et al. 2013]) by applications. All but two tools can handle analyzing malicious applications, and five were meant only for detecting malicious applications (as shown in Figure 4). For an entire analysis strategy, we believe that this is a narrow view of the capabilities of dynamic analysis. Properly constructed dynamic analysis has the ability to supplement or overcome many of the limitations of static analysis. For example, the literature measuring cryptographic vulnerabilities (MalloDroid [Fahl et al. 2012] and CryptoLint [Egele et al. 2013]) or advertising behavior has relied on static analysis followed by a manual review of execution, and dynamic analysis could reduce reliance on manual review.

4.2.2. Analysis Techniques. Most of the tools that we survey perform either taint tracking (five tools) or use system calls (four tools) to perform their analysis. Only three systems are capable of tracing network data. Most system call approaches deal with Linux-level system calls (often captured using `strace`); however, two systems actually gather traces at the Android API level—giving a deeper insight into the high-level semantics of the system. CopperDroid [Tam et al. 2015], on the other hand, takes a different approach for portability reasons, demonstrating that it is possible to reconstruct events such as Binder IPC through behavioral analysis of the captured system calls. Additionally, several systems track dataflows in apps, and five tools actually track data through multiple applications. Further still, two tools can actually track data passing through native code—in sharp contrast to static approaches, which do not.

4.2.3. Code Coverage Challenge. Unlike static analysis, dynamic analysis requires specific inputs to drive an execution trace (and thus an analysis). Selection of inputs determines which branches of code will be executed and ultimately governs code execution. Three of our tools address input generation, whereas the other six consider it an orthogonal problem beyond the scope of their research efforts. Of the tools that do address user input generation, two tools go beyond random input generation (i.e., fuzzing). Although user inputs are necessary to model an app's true behavior in general, Android has a rich set of system events that should also be generated to have a high code coverage. An example of these system events include receiving a call or SMS, and these events trigger code execution in a large body of legitimate and malicious applications. Only two tools address this challenge at all, and only one (PREC [Ho et al. 2014]) goes beyond fuzzing coverage, using the actions that an app expects to respond to as a means of generating events. TriggerScope [Fratantonio et al. 2016] attempts to statically detect logic bomb triggers via trigger analysis; similar approaches may be useful in the future for aiding dynamic analysis.

The view that input generation is an orthogonal problem to analysis is a common one, and it is supported by several separate works beyond the scope of this article. Choudhary et al. [2015] published a survey last year solely on the issue of test generation approaches. We refer readers to that work for more information on Android test generation [Amalfitano et al. 2012; Aviv et al. 2012; Xu et al. 2012; Gomez et al. 2013; Machiry et al. 2013; Hao et al. 2014; Liang et al. 2014; Mahmood et al. 2014; Narain et al. 2014].

4.2.4. Resiliency. When we designed this study, we knew from past experience that dynamic analysis frequently focused on detecting malware. We also knew that due to the costs and operational complexity of doing bare-metal malware analysis at scale, most malware detection tools would be emulation based. We were not surprised by this point, but we were surprised at how few tools aimed to hide the fact that apps are executing in an emulator (only one), and no tool looked for emulation detection schemes in the apps under test. Recently, researchers have attempted to address this limitation. Vidas and Christin [2014] proposed a technique to identify Android runtime analysis systems based on behavior, software/hardware, and performance. Sand-Finger [Maier et al. 2014] computes fingerprints of Android systems to find distinct signatures of analysis environments. And Mutti et al. [2015] proposed the tool BareDroid, designed for practical Android application bare-metal analysis.

4.3. Conclusions

This section outlined how the current state of the art addresses several fundamental challenges. One of the most important unaddressed challenges is that static analysis tools lack support for native code. Without support for native code analysis, none of the static analysis tools studied in this section can guarantee that their analysis is sound

Table II. Dynamic Analysis

		AppsPlayground [Rastogi et al. 2013]	CooperDroid [Tam et al. 2015]	TaintDroid [Enck et al. 2014]	DroidScope [Yan and Yin 2012]	MAdFraud [Crussell et al. 2014]	VetDroid [Zhang et al. 2013]	PREC [Ho et al. 2014]	WifiLeaks [Achara et al. 2014]
Input Simulation	User input simulation	●	●	○	○	○	○	●	○
	User input simulation: fuzzing	●	○	○	○	○	○	●	○
	User input: intelligent input generation (e.g., logins, zip codes)	●	●	○	○	○	○	○	○
	Network access simulation	○	○	○	○	○	○	○	○
	System event simulation	●	●	○	○	○	○	○	○
	System event simulation: fuzzing	●	○	○	○	○	○	○	○
	System event simulation: intelligent input generation	○	●	○	○	○	○	○	○
Techniques	Taint tracking	●	○	●	●	○	●	○	○
	Syscall traces	○	●	○	●	○	●	●	○
	Android API layer traces	●	○	○	●	○	○	○	○
	Library call traces	○	○	○	○	○	○	○	○
	Network traces	○	○	○	●	●	○	○	●
	Native instruction traces	○	○	○	●	○	○	○	○
	Dalvik instruction traces	○	○	○	●	○	○	○	○
	Concolic execution	○	○	○	○	○	○	○	○
Control- and DataFlow Tracking	Multiple applications	●	●	●	●	○	○	○	○
	Multiple applications: system applications	●	●	●	●	○	○	○	○
	Multiple applications: user applications	●	●	●	●	○	○	○	○
	Native code	○	●	○	●	○	○	○	○
Resiliency	Emulator detection detection	○	○	N/A	○	○	○	○	○
	Emulator Obfuscation—mimicking realistic environment	●	○	N/A	○	○	○	○	○
	Detects logic bombs or context-sensitive behavior	○	○	○	●	○	○	●	○
Misc	Extensible	○	○	○	○	○	○	○	○
	Useful for any app	●	●	●	●	○	●	●	○
	Publicly available	●	◐	●	●	○	○	○	○
	Benign only/malicious/dual use	D	M	B	M	M	D	M	D
Vulnerabilities Detected	Malicious activity	●	●	○	●	●	●	●	○
	Detect sensitive information leaks	●	◐	●	●	○	●	○	●
	Authentication	○	○	○	○	○	○	○	○
	Cryptography	○	○	○	○	○	○	○	○
	Data validation	○	○	○	○	○	○	○	○
	Intent spoofing	○	○	○	○	○	○	○	○
	Unauthorized intent receipt	○	○	○	○	○	○	○	○
	Configuration and deployment management	○	○	○	○	○	○	○	○
	Permission misuse	○	○	○	○	○	●	○	○
	Plagiarism detection	○	○	○	○	○	○	○	○
	Other	•	○	○	○	•	○	•	○

when faced with an application containing native code. This is especially troubling for tools that are intended to work with malicious code, as malicious applications often hide their functionality in native code to avoid detection. Other examples of similar open problems in static analysis include the lack of support for dynamic DEX loading and the minimal support for reflection in static analysis tools. Both represent areas of unsoundness in these tools, which can also be detrimental when dealing with malicious applications. These issues are complex, as both reflection and dynamic DEX loading can be obfuscated by encryption, packers, or other techniques. For dynamic analysis, one of the most important open problems is that analysis frameworks are not resilient to tool detection and evasion. In summary, there is still much to do to improve program analysis for Android apps.

5. EXPERIMENTAL METHODOLOGY

The systems security community values tangible artifacts. Public release of tools allows independent verification of scientific results and provides a stepping stone for other researchers and industry seeking to transition research ideas to practice. Many of the tools described in Section 4 are not open source or publicly available in any form (e.g., via a Web site). Fortunately, some tools have been made available due to the encouragement of funding agencies and technical program committees, as well as the values of some authors.

As part of systematization of Android application analysis tools, we seek to characterize publicly available tools with respect to the following high-level criteria:

5.0.1. C1. How difficult is it for an individual with a computer science background to use the tool? Although releasing the source code for a tool is immensely valuable, there is usually a steep learning curve before the tool may be applied. Research papers rarely (if ever) claim to produce production quality code. The technical artifacts are tools for researchers, written by researchers. However, the "consumers" of these tools are often first-year graduate students exploring research areas or employees for a company seeking to transition research ideas into practice. Therefore, if program committee members, advisors, and project managers are going to state "just go use tool X," we need to characterize the difficulty of using tool X.

5.0.2. C2. How well does the tool work in practice? It is sometimes difficult to extrapolate from a paper how well a tool will work in practice. In some cases, the dataset selected by the authors might have been unknowingly favorable to the tool. In other cases, the authors might have missed samples with important characteristics. Alternatively, new platform features or development trends might have changed assumptions made during the tool's development. We do not seek to repeat all of the experiments made by the authors; however, we do seek to understand how well tools work in practice. Perhaps more importantly, the findings may help to identify gaps leading to new research challenges.

The ultimate goal of this research is to evaluate each tool on these criteria independently on its own merits. Because we select such a diverse set of tools and evaluate them on such high-level criteria, these experiments are not meant to provide a narrow comparison between tools but rather to serve as an indicator of reproducibility and effectiveness of Android software security research.

The remainder of this section describes the experimental methodology used to evaluate tools against these criteria.

5.1. Tool Auditor Selection

To assess the difficulty of using tools, we recruited eight computer science graduate students (six masters students and two PhD students) from the University of Florida to serve as auditors. These students were available for a period of 10 weeks, bounding the available analysis time for this project. Auditors were selected through an interview process that assessed their computer and network security knowledge, programming experience, and understanding of the Android domain. Each of the students selected had prior Java programming experience and previous experience with Android. The tools that were assigned to each auditor were determined based on the auditor's prior experiences (e.g., programming languages) to maximize the likelihood of success.

5.2. Audited Applications

Each of the Android security tools were assessed by performance examination during the analysis of three classes of Android applications: DroidBench applications (version 2.0), mobile money applications, and the top 10 most widely used finance applications available through Google Play. Finance apps were chosen based on the fact that most of the data transmitted via financial apps (e.g., banking, personal budgets) is sensitive and private.

The DroidBench suite of applications [Arzt et al. 2014] is a set of minimal Android apps designed to stress test program analysis tools. The suite was chosen because it serves as the benchmark by which many of the tools based their effectiveness. Accordingly, reuse of DroidBench allowed us to perform independent validation. The mobile money applications serve as a means of providing branchless banking in the developing world. These applications were the subject of an earlier work [Reaves et al. 2015] and exhibit a multitude of issues including, but not limited to, incorrect certificate validation, flawed cryptography, and information leakage. By including this class of applications, we make available several real applications with known vulnerabilities to contrast the artificial applications in DroidBench. The following mobile money applications were used in this study: Oxigen Wallet, Vivo Zuum, GCash, MCoin, My Airtel, and mPay. Finally, the top 10 free finance applications in Google Play were selected because these apps provide a rich number of features associated with transactions, registration, and authentication that can be tested by the Android security tools and have a large user base. Critically, these applications are produced by developers associated or employed by large corporations, which separate this class from the mobile money and DroidBench apps. These applications are Google Wallet, Chase Mobile, Credit Karma, Bank of America, Wells Fargo Mobile, PayPal, Capital One Mobile, Android Pay, Venmo, and GEICO Mobile. We used the latest version of these applications as of September 28, 2015.

5.3. Tool Audit Procedure

For each tool, an auditor was tasked with analyzing all applications in the three aforementioned classes. The auditors each received one tool at a time and were asked to maintain a detailed time log. By doing so, we were able to monitor the time needed to configure the requisite environment and successfully generate results using a given tool. General information on each tool, including the problem it seeks to address, the vulnerabilities it can detect, and the type of analysis the tool performs, was documented to provide a baseline for expectations when analyzing the applications in the test set.

The assessment of C1 evaluated the tool based on accessibility, availability of instructions, and assumed knowledge. The purpose of the assessment was to determine if an auditor can use the tool's documentation and source code comments to interpret analysis results.

The assessment of C2 evaluated the tool based on precision and performance. Auditors recorded details related to output and accuracy. These details aided in determining whether or not the tool's claimed accuracy was reproducible on a different test set. The auditors' details also provided information about what users can expect once the tool's analysis was done. For performance, the auditors recorded CPU time and memory usage while analyzing applications. These measurements determined the computing resources needed to analyze not only artificial applications but also real ones. Hardware limitations are discussed in the following section.

Each auditor initially ran his or her respective tool using default settings. If the results proved to be either inaccurate or the tool ran without terminating, the auditor chose new tool settings that we believed would produce the expected results. The auditor continued this process until either the tool produced accurate results or our analysis time expired.

5.4. Hardware Limitations

During the audit of FlowDroid, TaintDroid, and DroidSafe, the auditors' test machines proved to be insufficient when analyzing the real applications from both the mobile money and finance top 10 sets. As a result, these tools and the remaining applications were analyzed on a machine with 48GB of RAM and dual quad-core Xeon processors. In the case of FlowDroid and DroidSafe, this machine proved to be inadequate due to lack of sufficient memory (FlowDroid) and a suspected memory leak (DroidSafe). The auditor of FlowDroid then employed an Amazon EC-2 Ubuntu machine equipped with 64GB of RAM and 12 vCPUs. The result proved to be the same. The lack of sufficient computing resources proved to be problematic in the case of FlowDroid and DroidSafe. These issues are discussed in greater detail in Section 6.

5.5. Tool Maintenance

Less than half of the tools have been updated within the past 6 months. In addition, less than half have a dedicated community, although all but one tool provided an email address for developers to send questions and comments. One of the seven tools did not have a code repository at all.

6. TESTING RESULTS

We now discuss the results of our empirical evaluation of publicly available tools for Android application analysis. For each tool, we begin with a brief summary of its characteristics and purpose. We then report the auditors' experiences using the tools, as well as the precision and performance of analysis. We found that some tools failed to run all of the applications.

6.1. Amandroid

Amandroid [Wei et al. 2014] is an application analysis tool designed to detect data leaks, data injections, and API misuse. Its main motivation is to improve the current state of static analysis of Android applications. The tool is publicly available and runs solely on Linux. Amandroid was written in Scala and computes a points-to analysis of every object—a detailed state of all objects during execution. It then builds an interprocedural control flow graph that handles ICC by including flow- and context-sensitive edges.

6.1.1. Usability Experiences. The auditor spent 3 hours setting up the tool; however, most of the setup time was the result of misleading documentation, and it could have been set up in 1 hour. The primary source of confusion was that Eclipse was not actually required. Negligible time was required to configure the tool, and all configuration options were displayed when executing it via the terminal. The Amandroid output was human readable but complex, as it requires a thorough understanding of Amandroid terminology. Additionally, the output appeared to be machine parsable due to its clear structure. The output is a comprehensive text file that lists the components of the code base and a verdict on each.

6.1.2. Experimental Results. The auditor found that Amandroid only successfully ran on small applications. All of the DroidBench applications ran successfully, each requiring approximately 90 seconds. However, only four of the six mobile money applications were successfully analyzed, and Amandroid identified a confused deputy vulnerability in one of them. In addition, only 2 of the 10 Google Play Store finance applications were successfully analyzed. Finally, in most cases, we needed to increase the default memory configuration parameter from 2GB to 4GB.

6.2. AppAudit

AppAudit [Xia et al. 2015] is a Web-based hybrid (static/dynamic) taint analysis tool. The main motivation of the tool is the discovery of sensitive data leaks. AppAudit is only available via its Web interface, and no source code is publicly available. AppAudit covers two types of analyses, static and dynamic, by first analyzing the application statically, then further pruning false positives by performing dynamic analysis in a simulated execution.

6.2.1. Usability Experiences. Since AppAudit is Web based, there was no client-side setup. The Web interface was very primitive and only contained an interface to upload an APK file. The analysis output was also very minimal, simply listing detected vulnerabilities or detected malware. Although the output was also human readable, it did not appear to be easily machine parsable. No manual effort was required to interpret the results once they were disclosed.

6.2.2. Experimental Results. Only 80% of the DroidBench applications ran successfully. Of the DroidBench applications that completed, AppAudit detected a leak in only 39 apps, in contrast to the 100% accuracy reported in the paper [Xia et al. 2015] for a previous version of the DroidBench suite. Curiously, AppAudit reported several DroidBench applications as matching malware signatures (often several different signatures matched a single application). For the other application datasets, none of the mobile money applications and only 5 of the finance top 10 were successfully analyzed. Four of the failures were due to native code in the application under test. The AppAudit paper lists native code analysis as a limitation. Finally, AppAudit took an average of 3 seconds to successfully analyze an app.

6.3. DroidSafe

DroidSafe [Gordon et al. 2015] is a static application analysis tool designed to analyze malicious information flows in Android source code and APK files. The primary contribution of DroidSafe is to closely model the Android runtime environment.

6.3.1. Usability Experiences. DroidSafe took the auditor 15 hours to set up. A significant portion of that time was spent hunting down and installing tool dependencies. The configuration itself took 4 hours and consisted of copying APK files into directories and adding makefiles for each. The auditor found that DroidSafe did not run on recently updated applications that do not support Android SDK 19. The documentation was hard to follow and appeared incomplete and outdated. The analysis of several applications resulted in NullPointerExceptions in the tool, and the auditor had to modify the code for the analysis to complete.

6.3.2. Experimental Results. The auditor was able to run DroidSafe on more than half of the DroidBench applications. Each ran successfully, taking on average 5 minutes to complete. The auditor was also able to run the tool on half of the Google Play Store apps and all of the mobile money apps; however, all analyses failed. The average time before the program crashed due to a suspected memory leak was about 60 minutes. In an attempt to complete the analyses, the memory allocation was increased from 16GB to 48GB, yet the memory increase did not help.

6.4. Epicc

The main motivation of Epicc is to build a precise and scalable analysis tool that tracks flows through ICC. Epicc [Octeau et al. 2013] is a static Android application analysis tool designed to analyze retargeted `.apk` files (if used in conjunction with Dare [Octeau et al. 2012]). It is designed to detect seven classes of vulnerabilities: activity hijacking, broadcast theft (sniffing), malicious broadcast injection, malicious activity launch, protected system broadcast without action check, malicious service launch, and service hijacking. The source code is publicly available.

6.4.1. Usability Experiences. Epicc is a Java-based tool that is run on the command line. Running Epicc on APK files also required setting up Dare. The setup instructions for Epicc were easy to follow, but Dare was slightly challenging to use. The output was human readable but not easily machine parsable.

6.4.2. Experimental Results. All applications from all three classes completed successfully. Vulnerabilities were only found in the mobile money apps and DroidBench apps. The top 10 financial apps ran in an average of 18 minutes, the mobile money apps ran within 15 minutes, and the DroidBench apps ran within 1 minute.

6.5. FlowDroid

FlowDroid [Arzt et al. 2014] is a static taint analysis tool for Android applications designed to identify leakage of private information. The main motivation of FlowDroid is to analyze and determine connections from source to sink. Therefore, it can only detect one type of vulnerability: information leakage. FlowDroid covers several types of taint analyses, which include context, flow, field, object-sensitive, and lifecycle-aware taint analysis. The source code is publicly available.

6.5.1. Usability Experiences. The initial setup took 1 hour and 45 minutes, which consisted of downloading FlowDroid .jar files, the missing SDK files required to run DroidBench, and installing Android Studio. An additional 2 minutes were required for each application analyzed to change configuration parameters to accommodate the application size. Overall, the documentation was clear; however, the analysis memory requirements for real applications came as a surprise. The FlowDroid output appeared to be machine parsable. The output was also human readable but unintuitive at first glance.

6.5.2. Experimental Results. All of the DroidBench applications were successfully analyzed. Each DroidBench application took on average of 4.505 seconds to run, took 30 seconds to track methods, and required 172.9MB of memory. The auditor's results confirmed the published results [Arzt et al. 2014]. The auditor attempted to analyze the top financial apps, but even the smallest application, Venmo, could not be analyzed with the default settings. In addition, only one of the six mobile money apps ran successfully after using an Amazon EC2 machine with 64GB of RAM and 16 virtual CPUs.

6.6. MalloDroid

MalloDroid [Fahl et al. 2012] is a static Android application analysis tool designed to detect improper TLS certificate validation that may allow Man-in-the-Middle (MitM) attacks. The source code is publicly available.

6.6.1. Usability Experiences. The tool setup took the auditor 1 hour to complete, which consisted of downloading `mallodroid.py` as well as Androguard [2012]. The tool documentation was sufficiently detailed to run the tool but was vague with regard to installing the Androguard framework. Most of the setup time was spent setting up Androguard through trial and error. The output was human readable, as well as machine readable, as long as the XML output option was passed at the command line.

6.6.2. Experimental Results. MalloDroid does not detect information flow leaks, and therefore the DroidBench apps were not analyzed. The MalloDroid tool produced an alert if there were potential TLS issues or vulnerabilities. The existence of unsafe code does not necessarily mean that these applications are vulnerable to an MitM attack. MalloDroid successfully analyzed all top financial apps, with the analysis for each application completing in about 4 minutes. Additionally, all of the mobile money applications were analyzed successfully, also requiring 4 minutes on average. Of the 10 applications analyzed, 80% had potential vulnerabilities. In the MalloDroid paper [Fahl et al. 2012], only 8% of those 13,500 applications produced potential vulnerabilities. Since we do not have the full list of apps that were tested, we can only assume that out of the 13,500, some of those did not use TLS at all, which will cause a clean MalloDroid scan. In the paper, Fahl's team also performed a manual audit of 100 apps and found that 41% of those were actually vulnerable, which is more consistent with the results presented here.

6.7. TaintDroid

TaintDroid [Enck et al. 2010] is a dynamic taint analysis tool designed to analyze commonly used applications to identify leaks of privacy-sensitive information. It is designed to run on a real device. The source code is publicly available.

6.7.1. Usability Experiences. Out of the tools discussed in the section, TaintDroid was the most difficult to set up, in part because it required the user to download and build the Android AOSP project. The initial auditor for this experiment spent a significant time attempting to get TaintDroid set up. The auditor did not have access to a supported device and was unable to get TaintDroid to run as an emulator on a remote server (due to the high system requirements for building AOSP). A second auditor was then tasked to perform the experiments. The auditor had access to a Galaxy Nexus device (supported by TaintDroid) and successfully completed the setup and experiments. TaintDroid primarily reports output to `logcat` but includes a GUI application to notify the user. The logs are partially human readable. For example, the output specifies the process ID, a hex encoded taint tag vector, and the entire buffer sent to the network. Although the GUI application has code to parse the logs, parsing the network buffer is more difficult. Finally, since TaintDroid uses purely dynamic analysis, all applications must be run by hand.

6.7.2. Experimental Results. Of the 119 DroidBench applications, 86 were successfully analyzed. Of the 86 analyzed applications, leaks were detected in 55. All 6 of the mobile money apps ran successfully, and no privacy leaks were detected. However, only 3 of the 10 Google Play Store finance applications would run. This was likely due to their use of native code, which is not supported by TaintDroid. When the native library is loaded, TaintDroid intentionally crashes the application.

7. LESSONS LEARNED

Our analyses of both the literature and tools have provided us with several important insights, which we share next.

In our study, we surveyed several hundred Android-related papers and understand that no tool can address all major issues, even within a specific topic such as static analysis. We also noticed in our review that in many instances, researchers claim that challenging analysis problems are "out of scope" or "orthogonal." Unfortunately, this approach, although certainly understandable and often valid, leads to missed opportunities. These difficult out-of-scope challenges are often avoided entirely in the face of easier (although important) problems. As a result, it is unclear whether efforts to address these so-called orthogonal problems can be built in an efficient and effective manner. In essence, it is not clear that problems claimed as orthogonal can indeed be addressed by complementary work as a matter of engineering integration. For example, analysis of native code (discussed later in this section) requires different algorithms and analysis infrastructure as opposed to analysis of Java or Dalvik code. We do not believe that researchers should be expected to produce fully featured analysis products ready for the market; however, we do believe that further exploration of both more difficult problems and research on the integration of orthogonal concerns are fertile ground for future research.

7.1. Static Analysis

All of the papers that we examined sidestep native code analysis. This is reasonable from the academic perspective because Android analysis and ARM analysis are two separate problems. However, there remains much room for innovation in this space, including how one meaningfully unites control- and dataflow analyses from heterogeneous computing platforms. The use of a common intermediate representation (IR) may be one approach, but representing both high-level Android and low-level ARM semantics together would be nontrivial. The use of native code has historically been limited to a small portion of applications in the market (5% to 7% [Crussell et al. 2012]). However, in recent work, researchers found that 14% of applications with fewer than 50,000 downloads contained at least one native library, whereas 70% of applications with more than 50 million downloads contained at least one native library [Viennot et al. 2014]. The increased importance of native code means that the challenge of analyzing a complete application—both Dalvik and native portions—is more important than ever.

Virtually every paper that we examined analyzed DEX files from compiled APKs. Accordingly, the community has spent very little time on traditional source code analysis. Although Dalvik is a high-level VM, some portion of the original semantics are still lost during compilation, requiring some amount of guesswork on the part of the decompiler to lift back to Java. Furthermore, the extensive Androidisms discussed in this article still remain stumbling blocks to traditional Java analysis frameworks.

Finally, reflection- and obfuscation-aware static analysis tools are generally lacking. These techniques need not be applied only against malware; rather, advances against the basic protections offered by tools such as ProGuard [2002] would benefit the community greatly.

7.2. Dynamic Analysis

As we mentioned earlier, dynamic analysis research represents a far smaller proportion of the community's effort than work on static analysis. There are several reasons we believe this to be true. First, emulation (the primary method of executing apps) is inherently slow. Despite improvements to both the official Android emulator and the release of improved third-party emulators like Genymotion, analysis of large, complex applications is still resource intensive. Bare-metal analysis strategies like the one proposed by BareDroid [Mutti et al. 2015] will help to address the scalability challenge, although at a significant cost in hardware. Such costs may still be worthwhile to larger organizations, such as corporate customers looking to create "internal app markets" for employees that automatically ensure that apps do not leak sensitive information. Second, input generation and code coverage is a fundamentally hard problem for all dynamic analysis strategies. It is also diametrically opposed to scalability and efficiency, as greater code coverage requires greater resources. These challenges mean that there is currently no dynamic analysis analogue to scalable lightweight static analysis systems like CryptoLint [Egele et al. 2013] or MalloDroid [Fahl et al. 2012], which can analyze code at market scale.

7.3. Tool Availability, Testing, and Maintenance

There is no question that our community values research artifacts. The authors of this article have personally pushed to shepherd papers through the conference acceptance process to ensure that such tools are made available. We commend researchers for publicly releasing their tools and hope to encourage more to do so in the future. Unfortunately, most of the tools surveyed in Section 4 released no artifacts beyond an academic paper. We understand that research artifacts are not industrial tools with dedicated engineering teams, and accordingly these xtools can be expected to be "rough around the edges." However, only a small portion of the community seems to be releasing code at all.

This is problematic for a few reasons. First, as noted previously, it prevents the community from buildixng on one or a small number of analysis frameworks. This in turn causes researchers to spend significant time attempting to reimplement mechanisms that others have already built. Second, part of our motivation for testing tools in this work was to reinforce the traditional scientific practice of independent evaluation of results. Indeed, for the tools we tested that claimed to successfully evaluate the DroidBench apps, we were able to mostly confirm this assertion. The lone difference is AppAudit, which now only finds 47.5% of the applications that it can run in the DroidBench set as vulnerable. We note that DroidBench has expanded from 65 to 116 applications since AppAudit's publication. Independent verification is commonplace in other communities but rarely practiced in our community, partially for the reason of availability. It is our hope that this work is the first of many that encourage this practice to become more valued and widespread.

Most tools required a significant investment in terms of time to get running. Our team found several of these research artifacts frustrating to use despite the best intentions of the tool developers. A significant amount of our team spent time trying to resolve dependency issues with software libraries and operating system incompatibilities. Many of these issues could be prevented if tool developers distributed not only their research code but also a complete virtual machine[2] containing a working build of their tools. We have found this method to be effective both for internal and external distribution of research artifacts. Additionally, our auditors indicated that it would be very helpful if each tool came with a test application so that it could be verified that that tool was correctly configured before attempting to analyze more complex apps. Last, we appreciate that releasing and maintaining tools does not come at zero cost. We encourage researchers to take advantage of funding opportunities such as the recently changed National Science Foundation Transition to Practice perspective. For researchers based in the United States, this effort will allow them to compete for funding designed to "support the development, implementation, and deployment of applied security research into an operational environment."[3]

8. CONCLUSIONS

Applications for the Android platform run the gamut of functionality, providing its users with solutions in spaces as diverse as the control of Internet of Things devices to connecting with business opportunities across the world. Because these applications are built based on Java, the research community has had unprecedented access to measure and improve the security of more than a million applications spanning this spectrum. The research community has responded with enthusiasm, producing a range

[2] Other technical approaches, including Linux containers, may provide similar benefits, but the end result should be an easy to run and validate artifact free of dependence on external projects.
[3] http://www.nsf.gov/pubs/2015/nsf15575/nsf15575.htm.

of techniques and tools to identify and mitigate application security issues. This is the first work attempting to reason about the space of Android application security research and the tools that these efforts have proposed. We began by discussing Android-specific challenges to program analysis and followed with a comprehensive analysis of the published application analysis research performed over the past 5 years. We then evaluated the ability for applications developers to apply the tools created in the preceding research to a range of real applications. We found first that many areas still require attention by the community and that the tools deserve additional study and support before they are ready for a wider community to reap their rewards. Finally, we offered several suggestions on how to move forward as a community.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and Benjamin Mood for his assistance in preparing figures.

REFERENCES

Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. SoK: Lessons learned from Android security research for appified software platforms. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Jagdish Prasad Achara, Mathieu Cunche, Vincent Roca, and Aurelien Francillon. 2014. WifiLeaks: Underestimated privacy implications of the ACCESS_WIFI_STATE Android permission. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*.

D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. 258–261.

Androguard. 2012. Androguard Home Page. Retrieved September 24, 2016, from https://github.com/androguard/androguard.

Android Developer Documentation. 2015. Building Apps with over 65K Methods. Retrieved September 24, 2016, from http://developer.android.com/tools/building/multidex.html.

Android Developers Blog. 2009. Backward Compatibility for Android Applications. Retrieved September 24, 2016, from http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html.

Android Developers Blog. 2011. Custom Class Loading in Dalvik. Retrieved September 24, 2016, from http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'14)*.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, 259–269.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 217–228.

Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2012. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, New York, NY, 41–50.

Baksmali. 2009. Baksmali Home Page. Retrieved September 24, 2016, from https://github.com/JesusFreke/smali.

A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. 2012b. Automatically securing permission-based software by reducing the attack surface: An application to Android. In *Proceedings of the 2012 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. 274–277.

Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012a. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP'12)*. ACM, New York, NY, 27–38. DOI:http://dx.doi.org/10.1145/2259051.2259056

Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the USENIX Security Symposium*.

A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. 2015. What the app is that? Deception and countermeasures in the Android user interface. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 931–948.

Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and soot. In *Proceedings of the ACM International Workshop on State of the Art in Java Program Analysis (SOAP'12)*.

Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. 2013. MAST: Triage for market-scale mobile malware analysis. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'13)*.

Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. 659–674.

Xin Chen and Sencun Zhu. 2015. DroidJust: Automated functionality-aware privacy leakage analysis for Android applications. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'15)*. ACM, New York, NY, Article No. 5.

Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*.

Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet? In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. 429–440.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on Android markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'12)*.

Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. MAdFraud: Investigating ad fraud in Android applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, New York, NY, 123–134.

X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S. M. Yiu. 2015. WeChecker: Efficient and precise detection of privilege escalation vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'15)*. ACM, New York, 25:1–25:12.

Benjamin Davis and Hao Chen. 2013. RetroSkeleton: Retrofitting Android apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*.

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. 77–101.

dex2jar. 2015. pxb1988/dex2jar. Retrieved September 24, 2016, from https://github.com/pxb1988/dex2jar.

Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY, 73–84.

Karim O. Elish, Danfeng Yao, and Barbara G. Ryder. 2012. User-centric dependence analysis for identifying malicious mobile apps. In *Proceedings of the Workshop on Mobile Security Technologies*.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems* 32, 2, 5:1–5:29.

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of Android application security. In *Proceedings of the USENIX Security Symposium*.

William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding Android security. *IEEE Security and Privacy Magazine* 7, 1, 50–57.

Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, , 50–61.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'11)*.

Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.

Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards detecting logic bombs in Android apps. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*.

Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. CLAPP: Characterizing loops in Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 687–697.

Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

D. Gallingani, R. Gjomemo, V. N. Venkatakrishnan, and S. Zanero. 2015. Static detection and automatic exploitation of intent message vulnerabilities in Android applications. In *Proceedings of the 2015 Mobile Security Technologies Workshop*.

Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2014. Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'14)*.

L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 72–81.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information flow analysis of Android applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'15)*.

Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*.

Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, New York, NY, 204–217.

Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. 2014. PREC: Practical root exploit containment for Android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*.

Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015a. From system services freezing to system server shutdown in Android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, NY, 1236–1247.

Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015b. SUPOR: Precise and scalable sensitive user input detection for Android apps. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 977–992.

Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*.

Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. 2012. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Proceedings of the Workshop on Mobile Security Technologies (MoST'12)*.

William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP'14)*. ACM, New York, NY, 1–6.

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS'11)*.

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. 153–169.

Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom'14)*. ACM, New York, NY, 519–530.

Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 2013. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'13)*. ACM, New York, NY, 21–32.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM* 58, 2, 44–46.

Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection analysis for Java. In *Proceedings of the 3rd Asian Conference on Programming Languages and Systems (APLAS'05)*. 139–160.

Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*.

Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*. 229–240.

Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 224–234.

Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 599–609.

D. Maier, T. Muller, and M. Protsenko. 2014. Divide-and-conquer: Why Android malware cannot be stopped. In *Proceedings of the 2014 9th International Conference on Availability, Reliability, and Security (ARES'14)*. 30–39.

Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. BareDroid: Large-scale analysis of Android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*. ACM, New York, NY, 71–80.

Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. 2015. UIPicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 993–1008.

Sashank Narain, Amirali Sanatinia, and Guevara Noubir. 2014. Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*. ACM, New York, NY, 201–212.

Nicolas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Ph.D. Dissertation. University of Cambridge. http://valgrind.org/docs/phd2004.pdf.

Patrick Northcraft. 2014. Android: The Most Popular OS in the World. Retrieved September 24, 2016, from http://www.androidheadlines.com/2014/02/android-popular-os-world.html.

Damien Octeau, Somesh Jha, and Patrick McDaniel. 2012. Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'12)*.

Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android with EPIC: An essential step towards holistic security analysis. In *Proceedings of the USENIX Security Symposium*.

Rohan Padhye and Uday P. Khedker. 2013. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the ACM International Workshop on State of the Art in Java Program Analysis (SOAP'13)*.

ProGuard. ProGuard Home Page. 2002. Retrieved September 24, 2016, from http://proguard.sourceforge.net/.

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*.

Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic large-scale dynamic analysis of Android applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY'13)*.

Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R. B. Butler. 2015. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'95)*.

Martin C. Rinard. 2001. Analysis of multithreaded programs. In *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*. 1–19.

Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. 2013. AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. ACM, New York, NY, 221–232.

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1995. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'95)*. 651–665.

Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the USENIX Security Symposium*.

Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Steven Y. Ko, and Lukasz Ziarek. 2014. Information flows as a permission mechanism. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. *More Sound Static Handling of Java Reflection*. Springer, 485–503.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 17–30.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 485–495.

Statista. 2015. Number of Available Applications in the Google Play Store from December 2009 to February 2016. Retrieved September 24, 2016, from http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

D. Sufatrio, J. J. Tan, T.-W. Chua, and V. L. L. Thing. 2015. Securing Android: A survey, taxonomy, and challenges. *ACM Computing Surveys* 47, 4, 58:1–58:45.

Mingshen Sun, Mengmeng Li, and John C. S. Lui. 2015. DroidEagle: Seamless detection of visually similar Android apps. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'15)*. ACM, New York, NY, 12.

Kimberly Tam, Aristide Fattori, Salahuddin J. Khan, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'15)*.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java bytecode optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*.

Timothy Vidas and Nicolas Christin. 2014. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer, and Communications Security (ASIA CCS'14)*. ACM, New York, NY, 447–458. DOI:http://dx.doi.org/10.1145/2590296.2590325

Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of Google Play. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM, New York, NY, 221–233.

Wala. 2006. WALA. Retrieved September 24, 2016, from https://github.com/wala/WALA.

Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby Robby. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'14)*.

Zheng Wei and David Lie. 2014. LazyTainter: Memory-efficient taint tracking in managed runtimes. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'14)*. ACM, New York, NY, 27–38.

Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on Android security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'13)*. 623–634.

Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective real-time Android application auditing. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan de Halleux, and Michal Moskal. 2012. User-aware privacy control via extended static-information-flow analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.

Zhi Xu, Kun Bai, and Sencun Zhu. 2012. TapLogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)*. ACM, New York, NY, 113–124.

Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the USENIX Security Symposium*.

Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY, 1043–1054.

Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014b. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*. ACM, New York, NY, 25–36.

Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014a. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'14)*.

Mu Zhang and Heng Yin. 2014. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*.

Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY, 611–622.

Received March 2016; revised August 2016; accepted August 2016