# Towards a Generic Framework for Automating Extensive Analysis of Android Applications

Li Li, Daoyuan Li
SnT
University of Luxembourg
{li.li,daoyuan.li}@uni.lu

Alexandre Bartel
EC SPRIDE
TU Darmstadt
alexandre.bartel@ec-spride.de

Tegawendé F. Bissyandé,
Jacques Klein, Yves Le Traon
SnT, University of Luxembourg
firstName.lastName@uni.lu

## ABSTRACT

Despite much effort in the community, the momentum of Android research has not yet produced complete tools to perform thorough analysis on Android apps, leaving users vulnerable to malicious apps. Because it is hard for a single tool to efficiently address all of the various challenges of Android programming which make analysis difficult, we propose to instrument the app code for reducing the analysis complexity, e.g., transforming a hard problem to a easy-resolvable one. To this end, we introduce in this paper Apkpler, a plugin-based framework for supporting such instrumentation. We evaluate Apkpler with two plugins, demonstrating the feasibility of our approach and showing that Apkpler can indeed be leveraged to reduce the analysis complexity of Android apps.

## CCS Concepts

**•Software and its engineering → Software notations and tools;**

## Keywords

Android; Static Analysis; Generic Framework; Apkpler

## 1. INTRODUCTION

The Android operating system has been progressively invading our cyber-physical space, being embedded in a large portion of hand-held devices such as smartphones and increasingly in home appliances such as TV sets. At Google I/0 2014, the company revealed that there were over 1 billion monthly active Android users, up from the 538 millions recorded in June 2013. A key point in favor of the adoption of Android is its large developer base which produces perhaps the largest set yet of millions of applications for entertainment, games, videos, financial management, etc. Expectantly, expert and "wanna be" hackers see in this platform an opportunity for distributing malicious applications. The recent Kaspersky's security bulletin has revealed that more than 98% of mobile malware found target the Android platform. To remedy this situation, the research community has to provide algorithms, techniques and tools to support a thorough analysis of Android applications when they are pushed to a market.

*SAC 2016, April 04 - 08, 2016, Pisa, Italy*
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3739-7/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851784

Due to scalability needs and requirements of specific test environments to perform dynamic analysis, a lot of current research on security testing of Android applications focus on static analysis. It allows to directly process the application code (source or bytecode) to report potential malicious implementation. This approach is doomed to yield over-approximation results because on one hand the entire code, even dead code which would never be executed at runtime, is analyzed and on the other hand different static analysis settings (e.g., path-insensitive, context-insensitive) will also introduce imprecision results. Nevertheless, heuristics can be devised to reduce the number of false alarms, or the results of static analysis can be used to refine a dynamic analysis process.

Currently, a number of tools have been developed for static analysis of Android applications by different independent persons (or teams). These tools however share some functionality but present each different drawbacks or deficiencies. A lot of work effort is thus recurrently wasted to reinvent the wheel, missing the opportunity to really go beyond the state-of-the-art and tackle the hardest issues. Furthermore, Android is fairly new, with its own specificities, and thus there are relatively few specific tools that are efficient in the analysis of applications. As an example, the FlowDroid [2] state-of-the-art tool does not deal with Inter-Component Communication (ICC), one of the most important features of Android. One solution would be to work on improving such tools to add new capabilities. Unfortunately, experiences by us and others, have shown that this endeavour is challenged by the monolithic implementation of the tools, even when they are open-sourced. It is also difficult to reuse the output of such tools as input of other analysis tools because of compatibility issues. Another alternative is to leverage on existing sophisticated Java-targeted tools. These tools however cannot directly analyze Android apps, although some of the functionalities that they implement are of interest for the Android platform. Finally, even when one wants to integrate the functionality implemented by other tool, he/she finds this endeavour challenging as the Android ecosystem for app analysis does not provide a platform for maintaining, advertising and distributing reusable functionality components. We propose, in this paper, to fill this gap through a generic and extensible framework for the analysis of Android Applications.

The framework will focus on making the applications analyzable by a variety of tools, instead of making tools to support different analysis approaches. Thus, to be generic, the framework will take as input an Android application and will output a new equivalent Android application having its code instrumented following different strategies depending on the need of the analysis and the capabilities of the analysis tool. Each instrumentation strategy of the code is performed by a plugin which addresses a specific issue for reducing the analysis complexity. For example, an instrumentation strategy could consist in replacing ICC in an Android app with traditional Java method calls to allow its full analysis by tools which do not take into account this specificities of Android. With the proposed framework, we intend to make readily reusable, through integration, a number of strategies devised in the literature, e.g., we could integrate a recent work called IccTA [9], an ICC solving strategy, to our framework. Finally, the proposed framework will improve the analysis outcome of state-of-the-art tools by allowing them to cover more Android apps and to produce more sound results.

In this paper we make the following contributions:

- We discuss the main challenges for performing efficient static analysis on Android apps, and propose a generic and extensible framework with a plugin system which allows developers to integrate new strategies for reducing the complexity of analyzing Android apps.
- We introduce the prototype implementation of the framework as well as a few of plugins which are now available in an open source project[1].
- Finally, we present the current status in the implementation of the platform and list the major future work in the hope of creating a synergy in the Android research community for contributing to this framework.

## 2. STATIC ANALYSIS OF ANDROID APPLICATIONS

We now summarize a few common challenges for tools in the analysis of Android applications in Section 2.1. Subsequently, we present an outlook to these challenges by discussing in Section 2.2 how our approach deals with them.

### 2.1 Challenges

When analyzing Android apps, researchers and practitioners are faced with several challenges. Some of them are Java-specific, because Android source code is written in Java. However, a number of them are mainly due to Android-specific peculiarities. We review these challenges to motivate our work.

**Dalvik bytecode.** Although Android apps are primarily developed in Java, they run in a Dalvik virtual machine. Thus, all app packages (apks) are distributed on markets with Dalvik bytecode, and only a relatively few are distributed with source code in open source repositories. Consequently, a static analyzer for Android must be capable of directly tackling Dalvik bytecode. Thus, most Java and Java bytecode analyzers, which could have been leveraged, are actually useless in the Android ecosystem. As an example, the mature FindBugs[2] tool, which has demonstrated its capabilities to discover bugs in Java bytecode, can not readily be exploited for Android programs.

[1]https://github.com/lilicoding/Apkpler
[2]http://findbugs.sourceforge.net

**Program entry point.** Unlike programs in most general programming languages such as Java and C, Android apps do not have a `main` method. Instead, each app program contains several entry points which are implicitly called by the Android framework at runtime. Consequently, it is tedious for a static analyzer to build a global call graph of the app. Instead, the analyzer must first search for all entry-points and build several call graphs with no assurance on how these graphs connect to each other, if ever.

**Component Lifecycle.** In Android, unlike in Java or C, different components of an application, have their own lifecycle. Each component indeed implements its lifecyle methods which are called by the Android system to start/stop/resume the component following environment needs. For example, an application in background (i.e., invisible lifetime), can first be stopped, when the system is under memory pressure, and later be restarted when the user attempts to put it in foreground. Unfortunately, because these lifecycle methods are not directly connected to the execution flow, they hinder the soundness of some analysis scenarios.

**Inter-Component Communication (ICC).** Android has introduced a special mechanism for allowing an application's components to exchange messages through the system to components of the same application or of other applications. This communication is usually triggered by specific methods, hereafter referred to as ICC methods. ICC methods use a special parameter, containing all necessary information, to specify their target components and the action requested. Similarly to the lifecycle methods, ICC methods are actually processed by the system which is in charge of resolving and brokering it at runtime. Consequently, static analyzer will find it hazardous to hypothesize on how components connect to one another unless using advanced heuristics. As an example, FlowDroid, one of the most-advanced static analyzers for Android, fails to take into account ICCs in its analysis.

**Libraries.** An android apk is a standalone package containing a Dalvik bytecode consisting of the actual app code and all library suites, such as advertisement libraries and burdensome frameworks [11]. These libraries may represent thousands of lines of code, leading to the size of actual app to be significantly smaller than the included libraries. This situation causes two major difficulties: (1) the analysis of an app may spend more time vetting library code than the real code; (2) the analysis results may comprise too many false positives due to the analysis of library "dead code". As an example, analyzing all method calls in an apk to discover the set of permissions required may lead to listing permissions which are not actually necessary for the actual app code.

**Java-specific challenges.** Since Android apps are mainly written in Java, developers of static analyzers for such apps are faced with the same challenges as with Java programs, including the issues of handling dynamic code loading, reflection, native code integration, multithreading and the support of polymorphism.
In the case of dynamic code loading and reflective calls, it is currently difficult to statically handle them. The classes that are loaded at runtime are often practically impossible to analyze since they often sit in remote locations, or may be generated on the fly.

Addressing the issue of native code is a different research adventure. Most of the time, such code comes in a compiled binary format, making it difficult to analyze. Analyzing multi-threaded programs is challenging as it is complicated to characterize the effect of the interactions between threads. Besides, to analyze all interleavings of statements from parallel threads usually result in an exponential analysis times [16]. Finally, polymorphic features also add extra difficulties for static analysis. As an example, let us assume that method $m_1$ of class $A$ has been overridden in class $B$ ($B$ extends $A$). For statement $a.m_1()$, where $a$ is an instance of $A$, a static analyzer in default will consider the body of $m_1()$ in $A$ instead of the actual body of $m_1()$ in $B$, even $a$ was instantiated from $B$ (e.g., with $A\ a\ =\ new\ B()$). This obvious situation is however tedious to resolve in practice by most static analyzers and thus leads to unsound results.

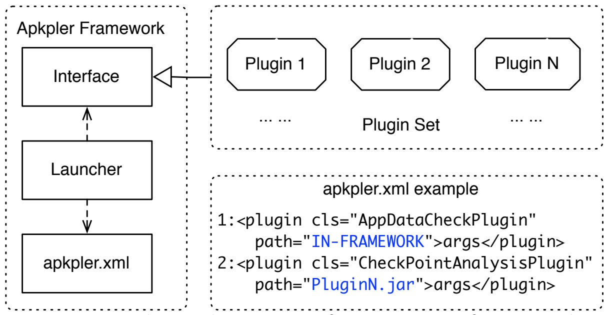
Let us remind at this point, that some challenges introduced in this section have already mature solutions to address them (e.g., point-to analysis for polymorphic feature). However, all the analyzers (existed or new developed) have to tackle them independently which renovates the wheels, resulting in resources wasted. On the contrary, all the analyzers will benefit from it if we solve each challenge as a plugin of our framework.

## 2.2 Outlook

The current situation in Android research is such that none of the state-of-the-art static analysis tools is addressing properly all the challenges listed above. Besides, regularly a new challenge is revealed and the different tools become obsolete. Let us consider two of the most widely used static analysis tools, namely FlowDroid [2] and Epicc [15]. The former does not support ICC and partially takes into account reflection. The latter does support ICC but makes the assumptions that reflection is not used in the programs. This leads to a situation where none of the tools can perform thorough analysis. To improve their analysis, we propose to feed each tool with apps that conform to their assumptions. Indeed, we believe it is unrealistic to attempt to improve the implementation of each analysis tool. A component-based approach (where functionality of each analysis tool can be leveraged in a huge framework) is also tedious to implement as it would require that each development team redesigns their tool to conform to a unique interface.

We thus propose a generic plugin-based framework for reducing the complexity of analyzing Android apps. This will indirectly enhance the output of existing state-of-the-art tools by enabling them to yield more sound results. In our framework, a plugin is designed to solve an analysis challenge. It also enables cooperation among multiple plugins to solve a complex challenge. Instead of modifying the analysis tools themselves, plugins instrument the app code to provide ideal inputs that will benefit all analysis tools. The operation of instrumentation consists in adding/removing/updating idempotent code (i.e., without modifying the functionality) to create a new app. However, currently we do not offer a guarantee that the new app could be executed.

The plugins could implement solutions from the literature to the mentioned challenges. For instance, we could package the solution of TamiFlex [4] in a plugin that deals with the Java reflection challenge. We could also package a solution from IccTA to avoid the limitations of tools that do not support ICC.



```
1:<plugin cls="AppDataCheckPlugin"
    path="IN-FRAMEWORK">args</plugin>
2:<plugin cls="CheckPointAnalysisPlugin"
    path="PluginN.jar">args</plugin>
```

Figure 1: The architecture of our generic framework.

## 3. FRAMEWORK PROTOTYPE

The main idea behind the design of the framework is about reducing the analysis complexity of Android apps by eliminating the challenges in the apps themselves rather than attempting to continuously re-implement state-of-the-art tools, with the risks of creating increasingly buggy tools. Thus, we aim to shift the efforts on developing independent (and non-reusable) tools onto developing plugins that could be viewed as "sanitizers" of apps, which could benefit all tools.

### 3.1 Architecture

The architecture of our framework, Apkpler, is shown in Figure 1. Apkpler is made up of three components: *Interface*, *Launcher* and *apkpler.xml*.

*Interface* is a public library, which provides the interfaces that every plugin needs to implement. It provides a means for developing a new plugin in a straightforward and easy way. Basically, only one method is needed to be overridden, which frees developers from being tied up with the framework but rather giving them a chance to only focus themselves on the plugin logic. *Interface* also provides some helper utilities (e.g., `SootHelper`) that all its plugins can benefit from. Besides, it maintains a key value mapping that enables plugins to sharing data.

*apkpler.xml* is a configuration, which hosts a sequence of plugins that are currently available in the plugin set. Users of Apkpler can customize *apkpler.xml* for their specific purpose. There are two ways to specify a plugin's path. The first one is simply setting it as `IN-FRAMEWORK`, which implies that the plugin (class in practice) is located in *Launcher*'s class path, e.g., *Launcher* directly accesses the plugin's source code through adding the required plugin projects on its build path. The second one is to specify the plugin's library path. In this case, *Launcher* uses the reflection mechanism to access the specified class.

*Launcher* is the entry-point of Apkpler. At first, it loads *apkpler.xml* to extract a sequence of plugins the user specifies. Then, it launches all the plugins one by one. Note that because of *apkpler.xml*, *Launcher* does not need to be modified for different aims of instrumentation.

### 3.2 Work Process

The work process of our framework is shown in Figure 2. As input, the framework is fed with an Android app whose Dalvik bytecode is transformed into Jimple code which is an intermediate representation used by Soot [7], a Java optimization framework. This transformation process is powered by Dexpler [3], a Dalvik to Jimple code translator.

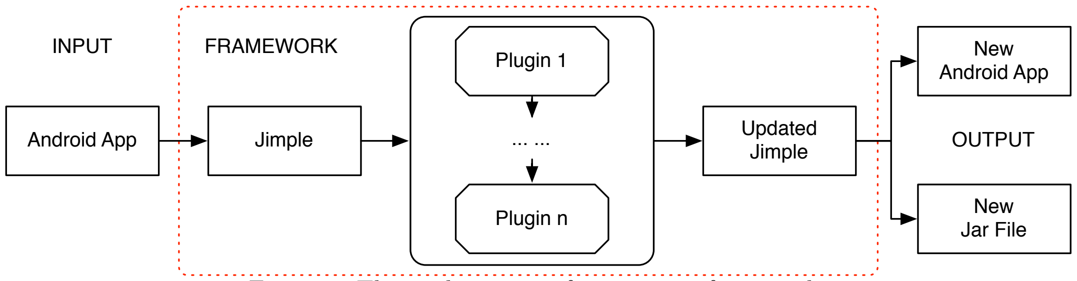


Figure 2: The work process of our generic framework.

Next, one can select and load a series of plugins to instrument the Jimple code for specific purposes (e.g., removal of library code). After the instrumentation step, the updated Jimple code can either be translated to Java bytecode and archived into a JAR file, or translated back into Dalvik bytecode and re-packaged into a new Android application.

The newly generated Android application is functionally equivalent to the original application, but is more convenient for analysis. In the case where the output is a new Jar file, some plugins are automatically selected (e.g., designation of the main entry point) to remove essential Android-specific characteristics.

## 3.3 Checkpoint mechanism

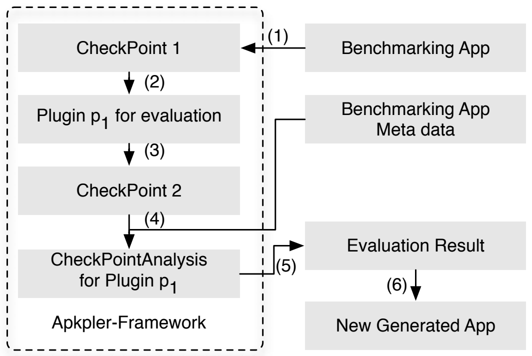


Figure 3: The work process of the *checkpoint* mechanism.

In order to evaluate the efficiency of integrated plugins, Apkpler provides a *checkpoint* mechanism for the plugin writers to build test suites for assessing the performance of their plugins. Thus, the *checkpoint* mechanism plays the role of a statistic helper, which is able to report the performance of a plugin in terms of time cost, instrumentation impacts, etc., and eventually can be used to build a ranking system to discriminate efficient plugins (among multiple similar-functionality plugins).

Given a plugin $p_1$, e.g., an approach for simplifying analysis of Android apps, that must be evaluated, and a set of benchmarking apps, that we know their meta-data (the expected results if $p_1$ applied) in advance. Figure 3 illustrates the work process of applying our *checkpoint* mechanism for $p_1$. Particularly, *checkpoints* (`CheckPoint 1` and `CheckPoint 2`) are performed by a built-in plugin called `AppDataCheckPlugin`, which records the current state (e.g., the execution time and the call graph) of a given app. Developers (or plugin verifiers) are however allowed to provide a meta-data file with the specifications of an app according to the given plugin as well as a customized version of the plugin-specific `CheckPointAnalysis` for $p_1$. These elements are used in the checkpoint mechanism to check whether the instrumentation impacts for a given app match the expected specifications, available as a meta-data file.

## 3.4 Summary of benefits

We now recall three main benefits to the adoption of the proposed framework: 1) **Easing new development:** developers may now readily implement a new analysis tool by focusing on the basic static analysis process and some specific searches. Indeed, any other common strategy, such as reduction of ICC or removal of library code, will be provided in the form of a framework plugin; 2) **Addressing challenges collaboratively:** if a challenge is so important that it hinders static analysis tools, a synergy could be created around the development of a corresponding plugin. Existing implementations from the literature may also be directly integrated; and 3) **Leveraging state-of-the-art tools:** with our proposed framework, existing tools, including FlowDroid, may now be enabled to perform more extensive analysis.

Overall, Apkpler reduces the analysis complexity of Android apps to indirectly enhance the analyzing ability of Android app analyzers. Given a feature $f_i$ (e.g., ICC) which represents an app's basic characteristic that an analyzer needs to tackle in order to yield that feature-aware results. Let us assume that the analysis complexity of an Android app $a$ is $W$, which corresponds to the analysis complexity of its $n$ features (from $f_1$ to $f_n$). If we assume that the analysis complexity of feature $f_i$ is $w_i$, then $W = \sum_{i=1}^{n} w_i$. Given a plugin $p_i$ that removes feature $f_i$ from $a$ (e.g., by identically transforming to other features). After launching Apkpler with $p_i$, the new generated app $a'$ of $a$ does not contain feature $f_i$ and its analysis complexity consequently reduces to $W - w_i$.

## 4. EVALUATION

Our primary evaluation addresses the following research questions.

**RQ1** What is the time performance introduced by Apkpler itself? Is it scalable for a large set of Android apps?

**RQ2** Is Apkpler able to enhance existing tools to perform more extensive analysis?

**RQ3** How is Apkpler able to evaluate the effectiveness of integrated plugins?

All the experiments discussed in this section are performed on a Core i7 CPU running a Java VM with 8GB of heap size.

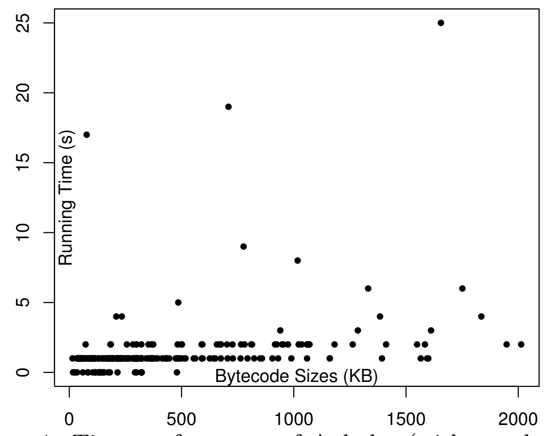


Figure 4: Time performance of Apkpler (without plugins) against the bytecode size.

## 4.1 RQ1: Time performance of Apkpler

We evaluate the scalability of Apkpler by assessing the runtime performance of the framework, in particular for unpacking and repackaging Android apps. For Apkpler to be usable in practice in real-world scenarios with hundreds of applications. We evaluate the time performance by running the framework with the ICC plugin described in Section 4.2 on a set of 300 Android apps randomly selected from Google Play store. Fig. 4 shows the overhead cost of Apkpler (i.e., without considering the time cost of plugins). This figure shows that the run time required by Apkpler is less than 5 seconds for over 95% of the apps, regardless of the app's bytecode size (i.e., the size of *classes.dex*). This evaluation reveals an acceptable overhead costed by Apkpler.

## 4.2 RQ2: Extensive analysis support for existing analyzers

To evaluate the feasibility of our approach, we have implemented a few plugins for Apkpler. In this section, we show two of them to illustrate how our approach is able to support extensive analysis for state-of-the-art Android app analyzers.

*Renaming duplicate class.* Android builds on the Linux kernel whose file systems follow a strict case-sensitive naming. An Android app can thus contain two classes with the same names which only differ in the case. However, because some PC-based file systems (e.g., Mac OS by default) are case-insensitive, such names will cause problems for PC-based analyzers (e.g., Soot [7] and Dare [13]). When one dumps the app, a class will be overwritten by its duplicated sibling (names with different case), leading to errors during analysis[3]. We have noticed that under Mac OS 10.9.2, analysis of both Soot and Dare are affected with the case-sensitive issue.

Previously, a Soot plugin[4] was introduced to solve this issue. However, this plugin was not reusable for Dare. In order to address it for any other tool we implemented a plugin for our framework to detect and consistently rename any such duplicated sibling classes. This plugin allowed Dare to successfully analyze 448 (nearly 36%) more apps from the Genome [19] dataset on the Mac platform.

*Supporting ICC.* A substantial portion of the functionality of an Android app (e.g., starting the music service, launching a phone call, etc.) is realized through communication between components, which on the other hand also offer opportunities for "malicious apps" to exploit. Unfortunately, most state-of-the-art tools, including FlowDroid [2], PCLeaks [10] and AndroidLeaks [6] fail to take ICCs into account in their analysis, yielding unsound results.

To address this challenge for static analysis, we have implemented a plugin based on IccTA [9] which is a taint analysis tool for detecting ICC-based privacy leaks in Android apps. The plugin instruments the app code by adding glue code to directly connect components using the Java class access mechanism (e.g., new instance) and building on the ICC links reported by Epicc [15] and IC3 [14]. As a result, the newly generated app does not contain ICC calls. We have instrumented some apps from the DroidBench benchmarks[5] and fed them to FlowDroid which then became able to successfully detect all inter-component leaks. It is worth to mention that the instrumented apps (by applying Apkpler) does not slow down the analysis of FlowDroid (13 seconds on average), comparing to the time cost of analyzing the original apps (12 seconds on average).

## 4.3 RQ3: Benchmarking integrated plugins

As introduced in Section 3.3, the *checkpoint* mechanism of our framework provides a means for Apkpler to evaluate the effectiveness of integrated plugins. In this section, we provide an example of leveraging the *checkpoint* mechanism through a case study: we evaluate the `Rename duplicate class plugin` (cf. RDC-plugin) introduced in Section 4.2.

In order to evaluate RDC-plugin, we have built a benchmark app set containing 20 apps randomly selected from such apps that suffer from the duplicated class name problem in the Genome project. At each checkpoint, before and after executing RDC-plugin, the states of apps are logged into different files separately. Further more, we implement a RDC-specified checkpoint analysis plugin (cf. RDC-CA-plugin) through our built-in framework, which analyzes the aforementioned two files and computes the instrumentation impacts. By comparing the impacts with pre-defined metadata of the benchmark apps, RDC-CA-plugin is able to decide whether a given test (a benchmark app) passes or not.

In our example, RDC-plugin passes 19 test cases, failing one test case. Further investigation discovers that the reason is caused by the existence of inner classes. Let us take class $x$ as an example, the plugin correctly renames class $x$ but misses the chance to also rename its inner class $x\$1$, leaving class $x\$1$ unchanged. The latest version of RDC-plugin has taken into account this situation.

## 5. RELATED WORK

To the best of our knowledge, Apkpler is the first approach that aims at reducing the complexity of Android apps in the hope of enhancing the output of analyzers. There are however a number of frameworks(e.g., [12] for training) for

[3]e.g., Soot reported a *"class com.adwo.adsdk.s read in from a class file in which com.adwo.adsdk.S was expected"* error on a Genome [19] app whose SHA 256 hash is 0015AE7C27688D45F79170DCEA16131CE5579-12A1A0C5F3B6B0465EE0774A452

[4]Commit 585fe047c18425653ba79a5c609c236f96f90aa9

[5]http://sseblog.ec-spride.de/tools/droibench/

the development of applications. Faruki et al. [5] presents an automated app vetting and malware analysis framework, which yields unified results that combine the strengths of several complementary approaches. Wei et al. [17] propose a general framework called Amandroid for security vetting of Android apps. Like Apkpler, Amandroid also enables a plugin-system for specific analysis. However, it limits itself on static taint analysis, i.e., from *source* to *sink*. Apkpler however, by focusing on instrumenting the app, will improve various kinds of analysis. Furthermore, although for simplicity we focused on discussing static analysis, our framework is also able to support dynamic analysis needs. For example, a plugin could be dedicated to injecting specific statements into analyzed apps to be able to control the app's execution flow (e.g., to skip time bombs, which cause the app to be executed only at a certain time) at runtime.

Code instrumentation has been widely used in the field of Android app analysis [1]. For example, IccTA [9] instruments Android ICC to explicitly connect two components to enable inter-component static taint analysis. AppSealer [18] instruments vulnerability-specific patches for Android applications to prevent component hijacking attacks. Unfortunately, contrary to Apkpler, those instrumentation are not currently reusable.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a generic plugin-based framework called Apkpler which leverages reusable plugins to instrument Android apps for reducing their analysis complexity. The newly generated app (with reduced analysis complexity) can directly benefit state-of-the-art analysis tools by supporting them performing more extensive analysis.

As future work, we plan to enrich Apkpler in a number of directions (as follows), so as to produce a full-fledged framework.

`Large set of plugins.` We are currently implementing more plugins to be integrated into the framework (e.g., we have implemented a plugin for reducing Android reflections). A few specific analysis challenges have been addressed in different tools from the literature. We plan to invite authors of such approaches to package their implementation into a reusable Apkpler plugin. For example, We are interested in integrating ApkCombiner [8] into our framework, in favor of existing analyzers to prevent collusion attacks among multiple Android apps.

`Detection of conflicts.` In a second direction, we plan to address the possibility for two plugins to conflict in the way they instrument the app code. Thus, we can provide a consolidation check during the selection of plugins to load for instrumenting an Android app.

`Support for dynamic analysis.` Finally, we plan to extend the framework to support dynamic analysis. To that end, we must ensure (or at least verify) that the instrumentation performed by the plugins still produce code that is executable.

## 7. ACKNOWLEDGMENT

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

## 8. REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *RV*, 2013.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[3] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*, 2012.

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[5] Parvez Faruki, Shweta Bhandari, Vijay Laxmi1 Manoj Gaur, and Mauro Conti. Droidanalyst: Synergic app framework for static and dynamic app analysis. 2015.

[6] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *TRUST*, 2012.

[7] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS 2011*.

[8] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *IFIP SEC*, 2015.

[9] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[10] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *TrustCom*, 2014.

[11] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. An Investigation into the Use of Common Libraries in Android Apps. In *Technique Report*, 2015.

[12] Jeremy Ludwig, Robert Richards, Bart Presnell, and Dan Fu. A general framework for developing training apps on android devices. In *I/ITSEC*, 2012.

[13] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *FSE*, 2012.

[14] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *ICSE*, 2015.

[15] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.

[16] Martin Rinard. Analysis of multithreaded programs. In *Static Analysis*, pages 1–19. Springer, 2001.

[17] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.

[18] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[19] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.