DROIDFAX: A Toolkit for Systematic Characterization of Android Applications

Haipeng Cai
Washington State University, Pullman, USA
hcai@eecs.wsu.edu

Barbara G. Ryder
Virginia Tech, Blacksburg, USA
ryder@cs.vt.edu

Abstract—**As the Android app market keeps growing, there is a pressing need for automated tool supports to empower Android developers to produce quality apps with higher productivity. Yet existing tools for Android mostly aim at security and privacy protection, primarily targeting end users and security analysts. Towards filling this gap, we present DROIDFAX, a toolkit that targets the developers to help them comprehensively understand Android apps regarding their code structure and behavioral traits. To that end, DROIDFAX features a systematic app characterization in multiple dimensions and views, through lightweight code analysis and profiling of both ordinary method calls (including those via reflection and exceptional control flows) and inter-component communications (including those within and across apps). The toolkit also includes a statement coverage tracker that works directly on bytecode and a dedicated tracer of events occurred during app executions. Applying DROIDFAX in two use cases has resulted in important findings about app behavioral patterns and an advanced security defense technique for Android. Empirical results also showed promising efficiency and scalability of DROIDFAX for practical adoption. A demo video for DROIDFAX can be viewed here or downloaded here.**

I. INTRODUCTION

As developers in other application domains, those of Android applications (*apps*) need a variety of tools to help them enhance development productivity while achieving and maintaining product quality. Yet, of numerous tools developed for Android to date, the majority target app-store analysts and end users concerning security and privacy [1]–[3]. In contrast, there are not sufficient tool supports immediately serving the developers. Among others, tools that assist developers with understanding app construction and behaviors can potentially empower them to avoid making choices that cause excessive costs and/or compromise quality during app development. Moreover, by knowing good choices based on this understanding, the developers can produce apps that have smaller security attack surfaces.

A few tools do exist that are capable of addressing a broader scope than security/privacy for Android apps, but they typically focus on coarse-level characteristics (e.g., app interaction with execution environments via network traffic [4], [5] and file operations [6]). Others are mostly limited to a specific aspect of app behaviors (e.g., in terms of system calls [7] and framework APIs [8]). While useful for obtaining a high-level understanding about app behaviors, these tools do not provide immediate information for systematic and in-depth examination of code-based app characteristics.

Towards filling this gap, we present DROIDFAX, a toolkit that directly serves Android app developers, helping them gain a comprehensive understanding of structural and behavioral traits of apps as regards to how they are coded and executed. DROIDFAX characterizes given apps through lightweight code analysis, instrumentation, and profiling. The substrate of this toolkit is a (Dalvik) bytecode manipulation and analysis framework that enables tracing of both ordinary method calls and Intent-based inter-component communications (ICCs). All method and ICC invocations are characterized rather than only particular calls to SDK APIs, including those made through reflection or via control flows due to exception-handling constructs. The communication characterization addresses ICCs linking components within individual apps as well as those across different apps. DROIDFAX works at application level requiring no modifications of the Android framework. Thus, it is not subject to portability issues or changes of the Android SDK which constantly evolves.

On top of the underlying framework, instrumented apps are exercised with either manual or automated inputs to produce method call and ICC traces. Based on these traces, a dedicated data analysis component of DROIDFAX derives various run-time app characteristics. Notably, DROIDFAX systematically characterizes each app in three orthogonal dimensions (*structure*, *communication*, and *security*) and three complementary views (*static*, *callsite*, and *instance*). The static app characteristics are directly computed from app package files (APKs) through lightweight static analysis. To accommodate needs for understanding additional aspects of app executions, our toolkit also includes two specialized tools for Android: a statement coverage tracker that works directly on APKs (requiring no access to app source) and a dedicated tracer that monitors events during app executions.

We have successfully applied DROIDFAX for a dynamic characterization study of randomly sampled apps which revealed important findings about app behaviors [9]. In another use case, comparative results from DROIDFAX on benign versus malicious apps led to the development of an advanced app security defense tool that achieved state-of-the-art effectiveness with superior capability and robustness [10]. Our empirical results also show promising efficiency and scalability of DROIDFAX for practical adoption.

II. DROIDFAX ARCHITECTURE

The overall architecture of DROIDFAX is shown in Figure 1. The three major components of DROIDFAX (as marked by numbered gray boxes) correspond to three phases of its process: *code analysis*, *tracing*, and *data analysis*. Also denoted are the three *inputs* to and three kinds of *outputs* of the toolkit (as marked by boxes of dashed boundaries). The process flow (indicated by arrowed lines) depicts how these inputs are used to produce the outputs through the three phases summarized below. The design of the two specialized tools is also unified in this architecture.

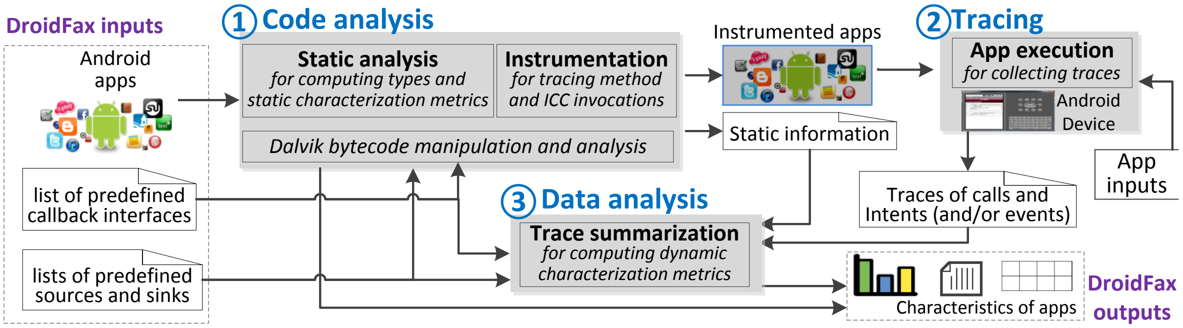
DroidFax inputs
Android apps
list of predefined callback interfaces
lists of predefined sources and sinks
① Code analysis
Static analysis
for computing types and static characterization metrics
Instrumentation
for tracing method and ICC invocations
Dalvik bytecode manipulation and analysis
Instrumented apps
Static information
② Tracing
App execution
for collecting traces
Android Device
App inputs
Traces of calls and Intents (and/or events)
③ Data analysis
Trace summarization
for computing dynamic characterization metrics
Characteristics of apps
DroidFax outputs

Fig. 1: DROIDFAX architecture, which shows its composition and process for systematic characterization of Android apps.

The *code analysis* phase takes the app(s) for characterization and instruments them to enable tracing. The second input, a list of callback interfaces, is used for implementing the event tracer. The main outputs of this phase are the instrumented apps. In addition, static characteristics are computed directly from app APKs, using the third input: lists of *sources* (APIs that allow user apps to access sensitive/private information) and *sinks* (APIs that send data out of the invoking app). Moreover, static information of the apps is computed to serve the subsequent phase of data analysis. Static analysis in this phase is built on our Dalvik bytecode analysis framework [11], which deals with low-level binary manipulation.

The *tracing* phase takes the instrumented apps and executes them on an Android device to collect traces. Either manual operations or automatically generated tests can be used as the app inputs required for triggering the executions. DROIDFAX incorporates utilities for using Monkey [12] to drive app executions with random inputs on an emulator [13], during which users can supplement with manual inputs as well. This phase outputs traces of all method and ICC invocations. For the event tracer, it outputs traces of events instead or additionally.

The last phase *data analysis* computes the multi-view and multi-dimensional characteristics from the traces it takes as main inputs. It also takes the static information computed in the first phase along with the lists of sources and sinks and the list of callback interfaces. This phase outputs dynamic characterization metrics. Finally, all (static and dynamic) characterization results are reported in a graphical, textual, or tabular format, which are the ultimate outputs of DROIDFAX.

III. CODE ANALYSIS

DROIDFAX performs two lightweight steps in code analysis: *instrumentation* for dynamic characterization, and *static analysis* for computing both static and dynamic characteristics.

A. Instrumentation

DROIDFAX performs instrumentation only at application level, without modifying the Android platform. For each given app, the instrumentation is realized by inserting probes in the app bytecode. For ease and extensibility of our implementation, each probe is simply an invocation of a run-time monitor. Different probes and associated run-time monitors are needed for tracking different execution information. All run-time monitors are organized as one third-party library, which can be modified by users for customized tracing, without re-instrumenting the apps (if already instrumented) nor changing the instrumenter in the code-analysis component of DROIDFAX. After instrumentation, the code of these monitors is built into the resulting APK and will be invoked when the instrumented app runs to the associated probes. Currently there are four different monitors in this component, as described below.

Method call monitor. After each callsite in the app code, a probe invoking this monitor is inserted. For ordinary static calls, the monitor reports the name of invoked method at the callsite (along with package and class name prefixes). For reflective calls, it reports the receiver object of the call such that the method actually called can be resolved at runtime. In addition, the caller name is also passed to the monitor, so that a dynamic call relationship in the form of *caller→callee* can be recorded during the tracing phase. To be practically adoptable, DROIDFAX also monitors callsites in exception-handling constructs, including methods invoked in `catch` and `finally` blocks [14].

Intent monitor. A probe calling this monitor is inserted before each ICC callsite. To resolve the run-time value of each Intent field, the entire Intent object is passed to the monitor, which will dump all Intent fields during the tracing phase. To compute app characteristics related to ICCs, the calling context of the ICC must be recorded as well. Thus, the monitor also records the ICC callsite itself and its (enclosing) caller method.

Event monitor. Android apps feature an event-driven programming paradigm. Each event is associated with a callback that responds to the occurrence of the event. There are two main classes of callbacks in Android: (1) *lifecycle methods* which respond to events related to the platform's management of the lifecycles of an app and its components, and (2) *event handlers* which respond to all other kinds of events, including system and user-interface (UI) events. DROIDFAX performs *specialized* instrumentation for monitoring callbacks so as to trace the associated events as required by the event tracer. Instead of probing the callsites directly, a probe that calls the event monitor is inserted at the *entry* of each callback that is defined in the app code. The reason for this special treatment is that the callbacks are typically invoked by the Android framework, which DROIDFAX does not instrument. The list of callback interfaces as part of the inputs for DROIDFAX setup (Figure 1)is used here to recognize callbacks.

Statement coverage monitor. DROIDFAX probes after each branch in an app to compute the statement coverage of run-time inputs to the app in real time during its executions. The coverage of a branch implies the coverage of statements that depend on the outcome of that branch. This is another specialized instrumentation, as needed by the statement coverage tracker. DROIDFAX works at app level, so it only reports the coverage for code available in the app APK.

DROIDFAX provides flexible options allowing for skiping any of the above instrumentations. While the instrumentation for event monitoring mainly serves the event tracer, users can opt for tracing events along with method calls and/or ICCs.

B. Static Analysis

To derive characteristics about app executions (e.g., component and callback distribution), DROIDFAX needs a few kinds of type information for its data-analysis phase: (1) the component type (e.g., *Activity*, *ContentProvider*) of each method (according to the inheritance relationship between its enclosing class and any of the four app component types in Android), (2) the type of callback interface (e.g., *System*, *UI*) implemented by each of such enclosing classes, and (3) the type of methods (and enclosing classes) in terms of three code layers: user code, third-party library, and the Android SDK. Computing (1) and (2) is realized through a class hierarchy analysis, while obtaining (3) requires parsing the manifest file of the app APK: user code is identified through the package information in the manifest, Android SDK code is identified according to the known list of SDK packages, and the rest is considered third-party library code.

Other static information computed during this step includes (1) total numbers of components of each type, (2) total numbers of classes and methods in each code layer, (3) total numbers of callbacks in each category, and (4) total number of Intent sending and receiving callsites (i.e., incoming and outgoing ICCs). This information is derived from the code of apps (rather than their execution traces), and is mainly used for calculating various kinds of coverage statistics (e.g., coverage of user-code methods, coverage of sources and sinks, etc.).

IV. TRACING

The computation of all dynamic characterization metrics relies on the traces collected at runtime. For each instrumented app, the *tracing* component of DROIDFAX installs the app to and then launches it on a pre-configured Android emulator. Next, the Monkey input generator is started to feed the installed app. The resulting traces are collected through Logcat [15], the standard logging utility as part of the Android SDK that pulls the outputs of logging APIs invoked during app executions to the host machine that runs the emulator. The trace content may include any combinations of (1) method call relationships, (2) ICC Intent objects, and (3) description of events, depending on the instrumentation options used in the first phase. A single file stores all traces per app, and the order of different trace items is that in which corresponding run-time monitors are invoked in the instrumented app. To compute characteristics related to inter-app ICCs, DROIDFAX launches a pair of apps simultaneously on the same emulator, with Monkey feeding each alternately for an equal amount of time (as user specified). Traces from both apps are stored in a single file, with the APK package names differentiating traces from each app. Note that the inner workings of DROIDFAX is orthogonal to the input source: other input generators of users' choice can be utilized in place of Monkey.

The trace produced by the coverage tracker is different: it immediately gives the result of the characterization, thus it needs no further analysis. The tracker produces a single line of coverage number in percentage as soon as the number increases by at least 1%. The last line indicates the final coverage attained by the inputs.

V. DATA ANALYSIS

The *data-analysis* component computes a variety of characterization metrics, in different perspectives (called *dimensions*) and granularity levels (called *views*). To capture the *general* structural and behavioral patterns of apps, each of these metrics is defined as a relative statistics rather than an absolute number.

A. Characterization Dimensions

DROIDFAX characterizes each app in terms of three orthogonal dimensions each focusing on a different aspect of the code structure and behavioral traits of the app: *structure*, *communication*, and *security*.

Metrics in the *structure* dimension characterize the structure of an app in terms of class/method distribution and categorization. In particular, structure metrics are percentages (1) of method calls targeting each of the three layers of app code, at the granularity levels of class and method, (2) of method calls within each layer and between any two layers, and (3) of method calls that are callbacks, in total and in each category. DROIDFAX computes metrics (1) and (2) according to the code layer membership of classes and methods in the static information passed from the *code-analysis* phase. To compute metrics (3), the list of callback interfaces is utilized. We manually created this list where the interfaces are first classified into two first-level categories (*System* and *UI*) and then further divided into ten sub-categories (five in each first-level category, e.g., *System_Status* within *System* category, and *Dialog* within *UI* category).

Metrics in the *communication* dimension characterize the communication of each component in an app with other components of the same app and with components in other apps. In particular, communication metrics are percentages (1) of components (i.e., the endpoints of ICCs) of each type, (2) of ICCs in each of four categories based on their being internal or external (two components of an ICC are in the same app or not) and implicit or explicit (one component of the ICC specifies explicitly the other component or not), and (3) of ICCs in each of three categories based on the payloads in the associated Intent being carried in the `data` or `extras` field of that Intent, or both. DROIDFAX computes these metrics referring to the component types associated with each method as part of the static information.

Metrics in the *security* dimension characterize app traits concerning security in terms of accesses to sensitive data and operations. In particular, security metrics are percentages of method calls that are sources or sinks, in total and in each source/sink category. To compute these metrics, DROIDFAX refers to the lists of sources and sinks where each source (sink) is labeled with the category the data (operation) it accesses (e.g., *Contact_Information* and *Network_Management*).

B. Characterization Views

To offer understandings of app characteristics from different perspectives, DROIDFAX computes the above metrics both based on app code only and from app execution traces. This differentiation leads to each metric in three complementary granularity levels (i.e., *views*): *static*, *callsite*, and *instance*. The latter two are *dynamic* views.

Specifically, metrics in the *static* view count classes and methods appeared in the app code, while metrics in the

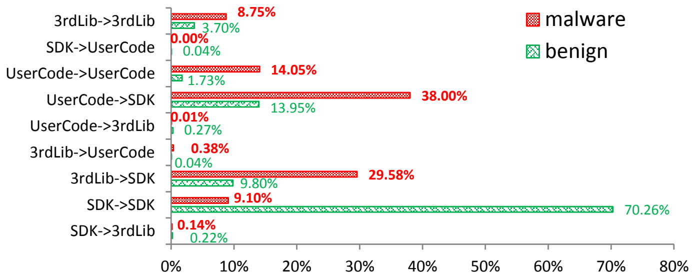

Fig. 2: Part of our findings with DroidFax: benign apps tend to have dominating calls *from* the SDK and considerable calls *to* user code, while malware tends to behave in the opposite.

dynamic view count those that appeared in app traces. Further, metrics in the *callsite* view consider the presence of methods and their enclosing classes based on the associated callsites covered, ignoring the call frequency of the callsites. The *callsite* view thus captures the *diversity* of class/method invocations. Metrics in the *instance* view count all instances of class/method invocations, capturing run-time app behaviors reflected by call *frequencies*.

Combining the static information and app traces, DroidFax also computes various kinds of coverage statistics at method and class/component levels. In particular, the metrics are percentages of methods, classes, components, callbacks, sources/sinks, and ICCs in various categories as described above, that are found in the traces over the respective totals found in the code. For most metrics, DroidFax reports the full distribution of metric values of all apps in graphical formats (charts and boxplots). For other metrics, it reports the mean metric values with variance in tabular or textual formats

VI. Applying DroidFax

This section briefly reports two use cases of our toolkit and its efficiency results, and discusses its limitations.

A. Use Cases

Use case 1. We have recently applied DroidFax to a behavioral characterization study of 125 apps randomly chosen from Google Play and 62 pairs among these apps [9]. The study was focused on the dynamic characteristics of apps for understanding Android application programming and security. From the outputs of DroidFax, we have derived many new insights about Android app behaviors (e.g., inter-layer call distribution as shown in Figure 2) and made a number of recommendations on optimizing app analysis for better cost-effectiveness tradeoffs. Although not documented in [9], we also found, among many other findings, that component distribution in the *static* view is noticeably less skewed than in the *dynamic* views (e.g., `Service` components are considerably invoked in code but not much at runtime).

Use case 2. We also have used DroidFax to identify 70 features that effectively distinguish behaviors of malware versus benign apps by comparing DroidFax results between these two groups [10]. With a set of 610 sample apps, we have developed a security classifier based on those features that achieved state-of-the-art effectiveness with superior robustness (e.g., against reflection and resource/syscall obfuscation).

We also successfully applied the coverage tracker of DroidFax as a stand-alone tool for benchmark selection according to a coverage criterion, in both use cases above. Currently, we are using the event tracer in our toolkit for another research project. While our toolkit aims to immediately assist mobile developers, we envision its broader use for various research purposes as well.

B. Efficiency and Scalability

Given the goals of the above two use cases, they enabled only those instrumentations for tracing all method calls and ICC Intents. For the total of 735 apps of 4KB to 26MB in size that it has been applied to, DroidFax's time costs in the *code analysis* phase ranged from 5 to 85 seconds (29 seconds in an average case). The instrumentation led to app code size increase by at most 2%. The run-time slowdown caused by the instrumentation and profiling was up to 3%. The tracing time and the trace storage space, while expected to dominate the overall toolkit run-time and storage cost respectively, will depend on how long users want/need to manipulate the instrumented apps. Finally, the data analysis phase of DroidFax took no more than 25 seconds. These numbers suggest that our toolkit can well scale to practical characterization of a large set of apps.

C. Limitations

To minimize the time overhead of tracing, DroidFax currently stores traces of different kinds all in a plain textual format. While we have not encountered trace storage challenges so far as we applied DroidFax, other users may do with apps that produce large traces quickly or in cases where long executions of apps are needed. A solution would be to trade the time overhead for better storage efficiency of DroidFax by adopting more sophisticated trace indexing and/or storage techniques (e.g., hierarchical tracing [16]).

Another limitation of DroidFax lies in its static analysis being subject to heavy/complex code obfuscation. In particular, complicated renaming of classes and methods may impede the recognition of component types and/or callback categories. As a result, metrics in the static view may not be computed correctly and instrumentation probes could be incorrectly placed. To overcome such issues, users may use a deobfuscator to preprocess the apps before applying DroidFax to them.

VII. Conclusion and Future Work

We presented DroidFax, a toolkit that systematically characterizes given Android apps with a diverse set of metrics in multiple dimensions and views to offer a comprehensive understanding on the code structure and behavioral traits of the apps. We have applied DroidFax in two recent use cases, which enabled new empirical findings and development of a superior technical solution for app security defense. The toolkit also offers specialized tools for event tracing and statement coverage tracking. The release package of DroidFax with usage documentation and video demo is available here.

An immediate next step with DroidFax is to use it in a longitudinal study for understanding how the Android ecosystem evolves. By characterizing apps from different groups (e.g., by year and SDK versions) and comparing app characteristics among these groups, insightful evolutionary patterns of the ecosystem can be derived and more advanced applications based on the results may be developed. We also plan to expand the characterization scope of the toolkit by including metrics on deeper analysis of app behaviors.

REFERENCES

[1] D. J. Tan, T.-W. Chua, V. L. Thing *et al.*, "Securing Android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–45, 2015.

[2] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[3] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, p. 76, 2017.

[4] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: multi-layer profiling of Android applications," in *Proceedings of ACM International Conference on Mobile Computing and Networking*, 2012, pp. 137–148.

[5] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards automatic fingerprinting of Android apps," in *Proceedings of IEEE International Conference on Computer Communications*, 2013, pp. 809–817.

[6] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proceedings of IEEE Computer Software and Applications Conference*, vol. 2, 2015, pp. 422–433.

[7] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 apps later: A view on current Android malware behaviors," in *Proceedings of International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, pp. 3–17.

[8] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[9] H. Cai and B. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proceedings of International Conference on Software Maintenance and Evolution*, 2017.

[10] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Unified dynamic detection of Android malware," Tech. Rep. TR-17-01, January 2017, http://hdl.handle.net/10919/77523.

[11] H. Cai and B. Ryder, "Understanding application behaviours for Android security: A systematic characterization," Virginia Tech, Tech. Rep. TR-16-05, May 2016, http://hdl.handle.net/10919/71678.

[12] Google, "Android Monkey," http://developer.android.com/tools/help/monkey.html, 2015.

[13] ——, "Android emulator," http://developer.android.com/tools/help/emulator.html, 2015.

[14] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.

[15] Google, "Android logcat," http://developer.android.com/tools/help/logcat.html, 2015.

[16] H. Cai and R. Santelices, "TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging," in *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 489–493.