AFRL-RI-RS-TR-2016-150

DARKDROID: EXPOSING THE DARK SIDE OF ANDROID MARKETPLACES

UNIVERSITY OF CALIFORNIA

JUNE 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-150 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2016	FINAL TECHNICAL REPORT	JUNE 2012 – DEC 2015

4. TITLE AND SUBTITLE

DARKDROID: EXPOSING THE DARK SIDE OF ANDROID MARKETPLACES

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-12-2-0101

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Christopher Kruegel, Giovanni Vigna, Engin Kirda, William Robertson

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
97

5f. WORK UNIT NUMBER
77

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California, Santa Barbara
Office of Research
3227 Cheadle Hall
Santa Barbara, CA 93106-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-150

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Our DarkDroid project aims to protect Android devices from their many security threats. In DarkDoid, we developed novel approaches to Android security issues based on both static and dynamic analysis techniques to pinpoint a variety of malicious behaviors, including logic bombs, dynamically-loaded code, GUI-based deception attacks, DOS-related attacks, and evasive apps that use the complexity of the Android framework to disguise automatic analysis systems. Moreover, our approaches can detect apps containing both intentional and unintentional vulnerabilities, such as unsafe code loading mechanisms and misuse of crypto APIs. We also studied and discovered several weaknesses in Android's framework and permission system; and we explored the behavior and possibility of sandboxing an understudied, yet critical security aspect -- native code components in Android apps.

15. SUBJECT TERMS

Mobile Security, Static Analysis, Dynamic Analysis, Malware Detection, Vulnerability Scanning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK WILLIAMS**
U	U	U	UU	21	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

1.0 SUMMARY

1.1 Problem Description

Due to the discovery of Android-based malware and an assumed lack of security awareness among the rapidly growing base of users of Android-powered devices, there is a clear and pressing need to supplement the Android security model with the means to detect malicious code and remove the applications that contain such code from Android marketplaces.

DarkDroid meets this need as a system to automatically detect malicious code in Android applications, using sophisticated static code analysis to check an application for the presence of code paths that have the potential to violate the confidentiality, integrity, and/or availability of critical data.

Detecting malicious code in the Android OS has two main challenges: First, programs make use of complex data structures (such as hash tables) and polymorphic classes with virtual methods. Second, Dalvik (Android) applications interact with the user in non-trivial ways, and multiple applications can collude (exchange messages) to achieve a single, malicious goal. We propose novel static analysis techniques that improve the precision of the analysis (and data-flow tracking) when facing complex data structures, user interaction (through intents), and virtual method calls. This guarantees that we identify and capture malicious code with a low number of false positives, even when malware authors attempt to obfuscate their actions. Moreover, because our static analyses critically depend upon the correctness of the underlying infrastructure, we will provide a formal basis for assurance that components of that infrastructure are correct with respect to a specification of their intended behavior.

1.2 Research Goals

To achieve our objective of creating a precise and comprehensive static analysis system for Android applications, we propose the following high-level goals:

1.2.1 Develop a comprehensive threat model for Android applications in a DoD context.

As a relatively new platform, it is almost certain that new types of attacks will emerge against Android applications and system components. Therefore, it is critical to formalize a model of potential attacks against Android applications from a DoD perspective.

1.2.2 Develop sophisticated whole-system static analyses to detect malicious Android applications.

Current static analysis systems (such as Kirin) have been deployed on Android systems with mixed results. The false positives detected by such systems are primarily the result of the systems analyses being restricted to single applications. Due to Android's security platform, malware could distribute necessary permissions over a series of applications, thereby bypassing current state-of-the-art detection systems.

Rather than single-application analyses, DarkDroid will utilize whole-system analyses. By analyzing an application in the context of the particular device that it will execute on, we can

improve the precision of our static analyses. With full knowledge of a system configuration, we can determine how a particular application will interact with other applications present on the system, and with the OS itself. This allows us to resolve the specific behavior resulting from invocations of components belonging to other applications or the system. This distinction is analogous to the difference between analyzing a dynamically-linked native application in the presence of its dependent libraries and analyzing the same application in isolation. In the case of Android applications, however, we note that the use of application markets greatly facilitates the whole-system approach by making the requisite environmental applications easily available for analysis.

1.2.3 Develop a robust model of mobile user input events to improve the precision of our static analyses.

To perform accurate static analysis of Android applications, it is critical that user interactions are taken into account. The omission of user interaction modeling in existing techniques is a central means by which malicious applications can trivially conceal evidence of their attacks; the DarkDroid project will improve upon the state-of-the-art to remove this opportunity for evasion.

1.2.4 Develop a trustworthy Android analysis infrastructure.

Because our static analyses critically depend upon the correctness of the underlying analysis infrastructure, we will provide a formal basis for assurance that components of that infrastructure are correct with respect to a specification of their intended behavior. More specifically, we propose developing a provably correct translator between Dalvik and our IR, such that important security-relevant semantics of Android applications are preserved across the transformation to the equivalent IR representation. Given an observationally equivalent IR representation of the application, we implement our analyses on top of the IR.

1.2.5 Develop an instrumentation framework to harden Android applications against attacks.

An instrumentation framework will extend the attack detection and prevention capabilities of our approach to runtime, by rewriting critical sections of potentially vulnerable or malicious Android applications. It is foreseeable that some malicious application will not be removable (perhaps due to the costs involved, or to instances of shared libraries). In these cases, our system would detect the malicious code and automatically rewrite it into something benign prior to installation on the device. Such an instrumentation framework presents a versatile capability that would allow legacy marketplace applications to be hardened against both known and emerging threats.

1.3 Expected Impact

This project will yield a comprehensive static analysis of Android applications, the impact of which will be significant. In particular, DarkDroid will guarantee the absence of broad classes of malicious code from Android applications, allowing sophisticated threat modeling of applications that are both for the general public and specific to DoD operations. Given the

anticipated low false positive rates and the detailed output to render human analysis easy, it will be possible to detect malicious applications before they are deployed on an Android marketplace. This ensures the security of the applications users and their data, such as soldiers on the battlefield that rely on DoD mobile applications to exchange information about the location of the enemy and improvised explosive devices (IEDs).

2.0 METHODS ASSUMPTIONS AND PROCEDURES

2.1 Detailed Description of Technical Approach

Considering a specific set of recognized research challenges, below we identify technical milestones for each of our five research goals:

2.1.1 DoD Android application threat model

The first milestone for this task is to produce an initial threat model based on known malware samples. This model will serve as a preliminary reference of civilian threats, although we anticipate that further research and discussion with DoD contacts will result in a significant evolution of the model. Consequently, the next milestone is a refined threat model that incorporates subsequent insights into how Android applications might be used and deployed in support of DoD missions. The final milestone of this task is a compiled database of malware composed of both public as well as custom-developed samples. This malware database will illustrate in a concrete manner representative threats encoded in the refined DoD Android application threat model.

2.1.2 Whole-system static analyses

Since high-fidelity system configurations are a fundamental requirement for whole-system static analysis, the first milestone for this task is to design and implement a system to reliably collect accurate system configurations from Android devices. Once this step has been completed, the next milestone will be an enumeration of different security properties that can be enforced using a priori knowledge of a system configuration for instance, restriction of data flows between applications to those authorized by policy. The final milestone for this task will be to design and implement a set of static analyses that will leverage system configurations in order to detect potential violations of the Android security properties, or identify opportunities for runtime enforcement through bytecode rewriting.

2.1.3 User input modeling

A static representation of the so-called back-stack is necessary to accurately model user input. Therefore, the first milestone for this task is to extend the analysis system with capabilities to manage multiple different back-stacks. The next milestone will be the successful static resolution of intents with regard to an a priori known system configuration. The final milestone for this task is reached when the analysis successfully resolves alternative execution paths depending on result values of launched Activities.

2.1.4 Trustworthy analysis infrastructure

The first milestone for this task will be to develop a structural operational semantics for DVM bytecode and our IR using an interactive theorem prover such as Coq. These will serve as a basis for the next milestone, namely that of a formal specification for a semantics-preserving translator from DVM bytecode to our IR. In principle, the next milestone will be to prove a

semantic preservation theorem for this specification, though we anticipate that this effort might occur concurrently with the specification's development in an iterative fashion. The final milestone will be the validation of an executable DVM-to-IR translator extracted from the formal specification.

2.1.5 Instrumentation framework.

The first milestone of this task is to develop a formal specification for a semantics preserving translator from JVM to DVM bytecode within the theorem proving environment selected as part of the above task. The next milestone will consist of proving the requisite semantic preservation theorem. The validation of the extracted executable translator will serve as the final milestone for this task.

2.2 Comparison with Current Technology

The Android OS, and mobile operating systems in general, are a relatively new application development platform. While mobile device vulnerabilities have been demonstrated by both academic and industry researchers, the mobile platform has historically not seen comparable levels of sophisticated, organized criminal behavior to those observed for more traditional platforms. However, with mobile devices playing an increasingly prominent role, new Android-specific malware is a near-certainty.

As noted, there are currently no whole-system, static analyses available for the Android OS. There are single-application analyses, such as Kirin. Kirin is a system that statically determines the set of permissions requested by an Android application during installation. From this permission set, Kirin determines whether an application is malicious based upon a specification of potentially dangerous combinations of permissions. For example, an application that requests permission to access GPS location data as well as the Internet has all the necessary permissions to build a detailed movement profile of the device on which it is installed, and, therefore, its user. However, since Kirin makes no attempt to verify whether an application actually contains malicious behavior, Kirin is unable to distinguish between a malicious application and one that merely allows a user to perform location-aware searches such as finding the nearest cafe.

Regarding instrumentation framework, there is a substantial amount of prior work that uses rewriting techniques. For native applications, both Pin and Dyninst allow for post-compilation binary rewriting on a number of platforms. There are also several frameworks for dynamically rewriting JVM bytecode, such as Apache BCEL and ObjectWeb ASM. Each of these frameworks provide rich instrumentation capabilities, including the ability to modify existing code and to inject new code. However, to our knowledge, there is no corresponding framework available for the Dalvik VM.

2.3 Deliverables Description

Achieving the goals of the DarkDroid project will result in a set of concrete deliverables that will be well-positioned for technology transfer. In our first task, we will create a comprehensive threat model that can be used to model attacks against DoD mobile resources. This model will be accompanied by a set of representative proof-of-concept attacks. We anticipate delivery of this threat model at the three-month mark of the project.

In our second task we will deliver a robust and extensible platform for performing precise static analysis of Android applications. This platform will incorporate system configuration information gleaned from installation target devices as well as a precise model of user interaction in order to improve the precision of these analyses. We anticipate delivery of the platform at the end of the project period. Furthermore, comprehensive documentation pertaining to the improvement over state-of-the-art techniques will be part of the deliverables for this task.

In our third task we will provide models of malicious code. The data-flow based models should be delivered by month 20. Models based on data value analysis by month 28; and non data-flow based models delivered at the three-year mark of the project.

Finally, our fourth task of defenses against malicious code will develop an instrumentation framework that will complement the static analysis platform by allowing for runtime enforcement of device-specific security invariants through bytecode rewriting. We anticipate completion of this task and delivery of the framework at the end of the project.

2.4 Technology Transition and Technology Transfer Targets

In concrete form, DarkDroid is composed of an advanced set of tools that together comprise an advanced malware detection and mitigation infrastructure for Android applications. While realizing this infrastructure will require significant advances in several areas, including static program analysis and verification, we are well-positioned to implement transitionable tools to DoD. Our team has a strong track record in creating performant implementations of cutting-edge research ideas. Fundamental to the design of the DarkDroid infrastructure is composability; that is, our tools will follow a cohesive design vision while using a modular approach that encourages flexibility, interoperability, and reuse. Our modern development infrastructure and software engineering practices allow us to rapidly prototype and refine implementations of our research ideas. In particular, we leverage distributed version control, flexible issue tracking, and lightweight testing to quickly construct high-quality systems.

4.0 RESULTS AND DISCUSSIONS

The DarkDroid project led to a series of projects, many of which were published in top-tier conferences within the security and software engineering fields. This section discusses the main projects and technologies we developed, and how these contributions have a positive impact for the security community and for the users of mobile devices.

4.1 Misuse of Crypto APIs

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. However, developers might not always use these primitives in a completely secure way. Thus, we developed a static analysis tool, called CryptoLint, which determines whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We developed program analysis techniques to automatically check programs on the Google Play marketplace, and found that 10,327 out of 11,748 applications using cryptographic APIs (88% overall) make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. Our paper[1] described several interesting insights and many technical details.

4.2 Dynamic Code Loading in Android Apps

The design of the Android system allows applications to load additional code from external sources at runtime. On the one hand, malware can use this capability to add malicious functionality after it has been inspected by an application store or anti-virus engine at installation time. On the other hand, developers of benign applications can inadvertently introduce vulnerabilities. Thus, any functionality based on dynamic code loading mechanisms, might pose a significant security risk and thus require special attention.

In one of our papers[2], we systematically analyzed the security implications of the ability to load additional code in Android. We developed a static analysis tool to automatically detect attempts to load external code using static analysis techniques, and we performed a large-scale study of 1,632 popular applications from the Google Play store, showing that loading external code in an insecure way is a problem in as much as 9.25% of those applications and even 16% of the top 50 free applications. For example, we found that an ad framework had a self-update functionality to download and execute code over HTTP. We immediately notified the vendor of the ad framework, and, shortly after, they acknowledged the vulnerability (that could have affected up to 10 million users), and eventually they deployed a fix. We also showed how malware can use code-loading techniques to avoid detection by exploiting a conceptual weakness in current Android malware protection mechanisms. Finally, we proposed modifications to the Android framework that enforce integrity checks on code to mitigate the threats imposed by the ability to load external code.

4.3 Third-Party Patch Application and Monitoring for Android Apps

The Android platform is somewhat unique in that it is open-source, and both device manufacturers and carriers can customize it for their own purposes. An unfortunate side effect of this, however, is that there is often a significant lag between the discovery of new vulnerabilities

and the distribution of patches to customized Android images. Exacerbating this problem is the disincentive for both device manufacturers and carriers to provide long-term support for the devices they sell. Rather, it is more profitable to convince users to purchase new devices. As a result, there is a large population of Android devices that contain known vulnerabilities, presenting a target-rich environment for malware authors and other malicious actors.

Since carriers and manufacturers cannot necessarily be relied upon for timely patches, we designed PatchDroid[3], a system for distributing and monitoring third-party patches for custom Android images. PatchDroid uses in-memory patching to maintain compatibility with standard Android platform security features, and incorporates a number of analyses to apply patches, monitor their stability, and perform automated rollback if instability is introduced. Our evaluation demonstrated that a number of major root escalation exploits could be prevented across a variety of Android devices. This work presents a viable path forward for reducing the number of exploitable Android phones in the wild, resulting in a safer mobile ecosystem.

4.4 Automatically Detecting Implicit Control Flow Transitions through the Android Framework

A wealth of recent research proposes static data flow analysis for the security analysis of Android applications. One of the building blocks upon which these analysis systems rely is the computation of a precise control flow graph. The callback mechanism provided and orchestrated by the Android framework makes the correct generation of the control flow graph a challenging endeavor. From the analysis point of view, the invocation of a callback is an implicit control flow transition facilitated by the framework. Existing static analysis tools model callbacks through either manually curated lists or ad-hoc heuristics. In one of our works[3], we demonstrated that both approaches are insufficient, and allow malicious applications to evade detection by state-of-the-art analysis systems.

To address the challenge of implicit control flow transitions (i.e., callbacks) through the Android framework, we implemented and evaluated a systematic treatment of this aspect. Our implementation, called EdgeMiner, statically analyzes the entire Android framework to automatically generate API summaries that describe implicit control flow transitions through the Android framework. We use EdgeMiner to analyze three major versions of the Android framework. EdgeMiner identified 19,647 callbacks in Android 4.2, suggesting that a manual treatment of this challenge is likely infeasible. Our evaluation demonstrates that the current insufficient treatment of callbacks in state-of-the-art analysis tools results in unnecessary imprecision. For example, FlowDroid misses a variety of leaks of privacy sensitive data from benign, off-the-shelf Android applications because of its inaccurate handling of callbacks. Of course, malicious applications can also leverage this blind spot in current analysis systems to evade detection at will. To alleviate these drawbacks, we make our results publicly available and demonstrate how these results can easily be integrated into existing state-of-the-art analysis tools. Our work allows existing tools to comprehensively address the challenge of callbacks and identify previously undetected leakage of privacy-sensitive data.

4.5 Deception and Countermeasures in the Android User Interface

Mobile applications are part of the everyday lives of billions of people, who often trust them with sensitive information. These users identify the currently focused app solely by its visual appearance, since the GUIs of the most popular mobile OSes do not show any trusted

indication of the app originating the graphic display shown to the user. For one of our projects, we analyzed in detail the many ways in which Android users can be confused into misidentifying an app, thus being deceived into giving sensitive information to a malicious app. Our analysis of the Android platform APIs, assisted by an automated state-exploration tool, led us to identify and categorize a variety of attack vectors (some previously known, others novel, such as a non-escapable full-screen overlay) that allow a malicious app to surreptitiously replace or mimic the GUI of other apps and mount phishing and click-jacking attacks. Limitations in the system GUI make these attacks significantly harder to notice than on a desktop machine, leaving users completely defenseless against them. To mitigate GUI attacks, we have developed a two-layer defense. To detect malicious apps at the market level, we developed a tool that uses static analysis to identify code that could launch GUI confusion attacks. We showed how this tool detects apps that might launch GUI attacks, such as ransomware programs. Since these attacks are meant to confuse humans, we have also designed and implemented an on-device defense that addresses the underlying issue of the lack of a security indicator in the Android GUI. We add such an indicator to the system navigation bar; this indicator securely informs users about the origin of the app with which they are interacting (e.g., the PayPal app is backed by *PayPal, Inc.*). We demonstrated the effectiveness of our attacks and the proposed on-device defense with a user study involving 308 human subjects, whose ability to detect the attacks increased significantly when using a system equipped with our defense. This project is described in detail in one of our papers[4].

4.6 Characterizing Loops in Android Applications

When performing program analysis, loops are one of the most important aspects that needs to be taken into account. In the past, many approaches have been proposed to analyze loops to perform different tasks, ranging from compiler optimizations to Worst-Case Execution Time (WCET) analysis. While these approaches are powerful, they focus on tackling very specific categories of loops and known loop patterns, such as the ones for which the number of iterations can be statically determined. As part of the DarkDroid project, we developed a static analysis framework to characterize and analyze generic loops, without relying on techniques based on pattern matching. For this work, we focus on the Android platform, and we implemented a prototype, called Clapp, that we used to perform the first large-scale empirical study of the usage of loops in Android applications. In particular, we used our tool to analyze a total of 4,110,510 loops found in 11,823 Android applications. In the paper describing Clapp in detail[5], we reported, as part of our evaluation, the results of our empirical study. We show how our analysis was able to determine that the execution of 63.28% of the loops is bounded, and we discuss several interesting insights related to the performance issues and security aspects associated with loops. We believe this tool can be useful to pinpoint “anomalous” loops that might significantly drain the battery of the device and mount a denial-of-service attack.

4.7 Large-Scale Analysis of Android Apps on Real Devices

To protect Android users, researchers have been analyzing unknown, potentially-malicious applications by using systems based on emulators, such as the Google’s *Bouncer* and *Andrubis*. Emulators are the go-to choice because of their convenience: they can scale horizontally over multiple hosts, and can be reverted to a known, clean state in a matter of seconds. Emulators, however, are fundamentally different from real devices, and previous

research has shown how it is possible to automatically develop heuristics to identify an emulated environment, ranging from simple flag checks and unrealistic sensor input, to fingerprinting the hypervisor's handling of basic blocks of instructions. Aware of this aspect, malware authors are starting to exploit this fundamental weakness to evade current detection systems. Unfortunately, analyzing apps directly on bare metal at scale has been so far unfeasible, because the time to restore a device to a clean snapshot is prohibitive: with the same budget, one can analyze an order of magnitude less apps on a physical device than on an emulator.

In one of our projects, we proposed *BareDroid*, a system that makes bare-metal analysis of Android apps feasible by quickly restoring real devices to a clean snapshot. Our physical-device testing platform works by loading on a device a custom "recovery" partition (containing the code that is first executed during the boot of an Android device). This partition is reloaded from a computer to a device at every device's boot to ensure its integrity. Our custom "recovery" verifies the integrity of a bundle (stored on the device), containing an uncompromised Android operating system. After the integrity of this bundle is verified, it is decompressed and booted.

In our paper[6], we show how *BareDroid* is not detected as an emulated analysis environment by emulator-aware malware or by heuristics from prior research, allowing *BareDroid* to observe more potentially malicious activity generated by apps. Moreover, we provide a cost analysis, which shows that replacing emulators with *BareDroid* requires a financial investment of less than twice the cost of the servers that would be running the emulators. We released *BareDroid* as an open source project, in the hope it can be useful to other researchers to strengthen their analysis systems.

4.8 Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications

In the past, we studied how benign and malicious Android applications are dynamically loading external code components. Our study showed that several real-world benign applications implemented this technique in a vulnerable way, so that it was possible to mount remote code execution attacks. In our previous work, we also proposed several modifications to the Android framework to mitigate this class of threats. This design choice has the advantage that the proposed system is transparent to the developer, but it has the disadvantage that it is difficult to be adopted by the community, as several modifications to the framework are required. For this reason, we attempted to mitigate the impact of this class of vulnerabilities by following a different approach. In particular, we designed and implemented a new Android library that, similarly to the existing ones, allows a developer to dynamically load additional components, while at the same time preventing her from introducing this class of security vulnerabilities. This library, which we called *Grab 'n Run*, provides several interfaces that resemble the standard ones. For example, our library implements a new interface called SecureDexClassLoader, which on the one hand provides an interface that is similar to the standard DexClassLoader library, while, on the other hand, it also forces the developer to 1) cryptographically sign each component that needs to be dynamically loaded, and to 2) verify, at loading time, that such component has been not modified. We believe this component to be ready to be used for real-world applications, and, for this reason, we recently open-sourced it. The technical details and design choices are discussed in another of our papers[7].

4.9 Native-Code Sandboxing

Current static analysis techniques for Android applications operate at the Java level, i.e., they analyze either the Java source code or the Dalvik bytecode. However, Android allows developers to write code in C or C++ that is cross-compiled to multiple binary architectures. Furthermore, the Java-written components and the native code components (C or C++) can interact. Native code can access all of the Android APIs that the Java code can access, as well as alter the Dalvik Virtual Machine, thus rendering static analysis techniques for Java unsound or misleading. In addition, malicious apps frequently hide their malicious functionality in native code or use native code to launch kernel exploits. It is because of these security concerns that previous research has proposed native code sandboxing, as well as mechanisms to enforce security policies in the sandbox. However, it is not clear whether the large-scale adoption of these mechanisms is practical: is it possible to define a meaningful security policy that can be imposed by a native code sandbox without breaking app functionality?

As part of the DarkDroid project, we performed an extensive analysis of the native code usage in 1.2 million Android apps[8]. We first used static analysis to identify a set of 446k apps potentially using native code, and we then analyzed this set using dynamic analysis. This analysis demonstrates that sandboxing native code with no permissions is not ideal, as apps' native code components perform activities that require Android permissions. However, our analysis provided very encouraging insights that make us believe that sandboxing native code can be feasible and useful in practice. In fact, it was possible to automatically generate a native code sandboxing policy, derived from our analysis, that limits many malicious behaviors while still allowing the correct execution of the behavior witnessed during dynamic analysis for 99.77% of the benign apps in our dataset. The usage of our system to generate policies would reduce the attack surface available to native code and, as a further benefit, it would also enable more reliable static analysis of Java code.

4.10 Detection of Logic Bombs in Android Apps

Existing static analyses are effective in detecting the presence of most malicious code and unwanted information flows. However, certain types of malice are very difficult to capture explicitly by modeling permission sets, suspicious API calls, or unwanted information flows. One important type of such malice is malicious application logic, where a program (often subtly) modifies its outputs or performs actions that violate the expectations of the user. Malicious application logic is very hard to identify without a specification of the "normal," expected functionality of the application. We refer to malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances as a logic bomb. This is a powerful mechanism that is commonly employed by sophisticated malware, Advanced Persistent Threats (APTs), and state-sponsored attacks: in fact, in this scenario, the malware is designed to target specific victims and to only activate under certain circumstances.

For the DarkDroid project, we developed a novel static analysis tool called *TriggerScope*, which we believe constitutes the first step towards detecting logic bombs. In the paper describing this project[9], we proposed trigger analysis, a new static analysis technique that seeks to automatically identify triggers in Android applications. Our analysis combines symbolic execution, path predicate reconstruction and minimization, and inter-procedural control-dependency analysis to enable the precise detection and characterization of triggers, and it overcomes several limitations of existing approaches. We implemented a prototype of our

analysis, and we evaluated it over a large corpus of 9,582 benign apps from the Google Play Store and a set of trigger-based malware, including the recently-discovered HackingTeam's *RCSAndroid* advanced malware. Our system is capable of automatically identifying several interesting time-, location-, and SMS-related triggers, with a low false-positive rate (0.38%), and it achieves 100% detection rate on the malware set. We also showed how existing approaches, specifically when tasked to detect logic bombs, are affected by substantial false positive and false negative rates. Finally, we discussed the logic bombs identified by our analysis, including two previously-unknown backdoors in benign apps.

5.0 CONCLUSIONS

We believe the techniques and prototypes developed through the DarkDroid project constitute a significant step forward for the security of mobile devices and applications. In particular, our work resulted in novel approaches based on both static and dynamic analysis techniques, which are able to precisely pinpoint a variety of malicious behaviors, such as logic bombs, dynamically-loaded code, GUI-based deception attacks, DOS-related attacks, and evasive apps that use the complexity of the Android framework to foil automatic analysis systems. Moreover, we developed a variety of approaches to detect apps containing (unintentional or, in the worst case, intentional) vulnerabilities, such as unsafe code loading mechanisms and misuse of crypto APIs. We also studied and discovered several weaknesses in the Android framework itself and in its permission system, and we explored the behavior and possibility of sandboxing native code components in Android apps, which are an understudied, yet critical, security aspect of the Android ecosystem.

For the future, we are planning to extend the applicability and improve the performance and accuracy of all projects and prototypes we worked on during these years. However, we are also planning to start investigating emerging aspects and technologies related to the Android ecosystem. In particular, we plan to explore the following two directions.

First, Android apps are becoming increasingly complex and, for performance reasons, more and more developers are opting to implement parts of the app's functionality in a low-level language (e.g., C, C++), generating the so-called "native code" components. From the security point of view, this represents a risk, since the switch to low-level languages will introduce vulnerabilities related to memory corruption. Thus, this aspect will inevitably make the attack surface bigger. Unfortunately, bugs in Android native code component are difficult to find and, to date, no analysis system has been developed that focuses on the efficient and effective discovery of such bugs. Moreover, finding bugs in these components is more challenging than finding bugs in traditional binaries: in fact, Android native code components do not have a single entry point, and they interact with the rest of the app (written in Java) through JNI. The analysis of these components thus presents several interesting research challenges that we are planning to address.

Second, we want to investigate the security implications of a recent proposal of a technology to perform cross-device tracking through the usage of the ultrasound channel. In particular, this technology aims at tracking the connection among two (or multiple) devices so that they can be linked together and perform, for example, advertisement retargeting. There are multiple companies already implementing and offering a solution based on this technology, and they rely on identifying “beacons” that are transmitted from one device to another through the ultrasonic space of the audio channel. The more this technology becomes widely used, the bigger the impact of security implications will be. We are planning to investigate on this aspect, to formalize the threat, and to propose practical solutions for the concerned users.

6.0 REFERENCES

[1] Egele, Manuel, Brumley, David, Fratantonio, Yanick, Kruegel, Christopher, "An Empirical Study of Cryptographic Misuse in Android Applications," Proceedings of the ACM Conference on Computer and Communications Security (CCS), November, 2013.

[2] Poeplau, Sebastian, Fratantonio, Yanick, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2014.

[3] Mulliner, Collin, Oberheide, Jon, Robertson, William, Kirda, Engin, “PatchDroid: Scalable Third-Party Patches for Android Devices,” Annual Computer Security Applications Conference (ACSAC), New Orleans, LA US, Dec 2013.

[4] Cao, Yinzhi, Fratantonio, Yanick, Bianchi, Antonio, Egele, Manuel, Kruegel, Christopher, Vigna, Giovanni, Chen, Yan, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2015.

[5] Bianchi, Antonio, Corbetta, Jacopo, Invernizzi, Luca, Fratantonio, Yanick, Kruegel, Christopher, Vigna, Giovanni, "What the App is That? Deception and Countermeasures in the Android User Interface," Proceedings of the IEEE Symposium on Security and Privacy (SP), May, 2015.

[6] Fratantonio, Yanick, Machiry, Aravind, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, "CLAPP: Characterizing Loops in Android Applications," Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE), September, 2015.

[7] Mutti, Simone, Fratantonio, Yanick, Bianchi, Antonio, Invernizzi, Luca, Corbetta, Jacopo, Kirat, Dhilung, Kruegel, Christopher, Vigna, Giovanni, "BareDroid: Large-Scale Analysis of Android Apps on Real Devices," Proceedings of the Annual Computer Security Applications Conference (ACSAC), December, 2015.

[8] Falsina, Luca, Fratantonio, Yanick, Zanero, Stefano, Kruegel, Christopher, Vigna, Giovanni, Maggi, Federico, "Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications," Proceedings of the Annual Computer Security Applications Conference (ACSAC), December, 2015.

[9] Afonso, Vitor, Bianchi, Antonio, Fratantonio, Yanick, Doupe, Adam, Polino, Mario, de Geus, Paulo, Kruegel, Christopher, Vigna, Giovanni, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2016.

[10] Fratantonio, Yanick, Bianchi, Antonio, Robertson, William, Kirda, Engin, Kruegel, Christopher, Vigna, Giovanni, "TriggerScope: Towards Detecting Logic Bombs in Android Apps," Proceedings of the IEEE Symposium on Security and Privacy (SP), May, 2016.

APPENDIX: PUBLICATIONS

Afonso, Vitor, Bianchi, Antonio, Fratantonio, Yanick, Doupe, Adam, Polino, Mario, de Geus, Paulo, Kruegel, Christopher, Vigna, Giovanni, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2016.

Bianchi, Antonio, Corbetta, Jacopo, Invernizzi, Luca, Fratantonio, Yanick, Kruegel, Christopher, Vigna, Giovanni, "What the App is That? Deception and Countermeasures in the Android User Interface," Proceedings of the IEEE Symposium on Security and Privacy (SP), May, 2015.

Bianchi, Antonio, Fratantonio, Yanick, Kruegel, Christopher, Vigna, Giovanni, "NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running Stock Android," Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), October, 2015.

Carter, Patrick, Mulliner, Collin, Lindorfer, Martina, Robertson, William, Kirda. Engin, "CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes." Financial Cryptography and Data Security (FC), Barbados, Feb 2016.

Cao, Yinzhi, Fratantonio, Yanick, Bianchi, Antonio, Egele, Manuel, Kruegel, Christopher, Vigna, Giovanni, Chen, Yan, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2015.

Egele, Manuel, Brumley, David, Fratantonio, Yanick, Kruegel, Christopher, "An Empirical Study of Cryptographic Misuse in Android Applications," Proceedings of the ACM Conference on Computer and Communications Security (CCS), November, 2013.

Falsina, Luca, Fratantonio, Yanick, Zanero, Stefano, Kruegel, Christopher, Vigna, Giovanni, Maggi, Federico, "Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications," Proceedings of the Annual Computer Security Applications Conference (ACSAC), December, 2015.

Fratantonio, Yanick, Bianchi, Antonio, Robertson, William, Egele, Manuel, Kruegel, Christopher, Kirda, Engin, Vigna, Giovanni. "On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users," Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), July, 2015.

Fratantonio, Yanick, Machiry, Aravind, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, "CLAPP: Characterizing Loops in Android Applications," Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE), September, 2015.

Fratantonio, Yanick, Bianchi, Antonio, Robertson, William, Kirda, Engin, Kruegel, Christopher, Vigna, Giovanni, “TriggerScope: Towards Detecting Logic Bombs in Android Apps,” Proceedings of the IEEE Symposium on Security and Privacy (SP), May, 2016.

Mulliner, Collin, Oberheide, Jon, Robertson, William, Kirda, Engin, “PatchDroid: Scalable Third-Party Patches for Android Devices,” Annual Computer Security Applications Conference (ACSAC), New Orleans, LA US, Dec 2013.

Mutti, Simone, Fratantonio, Yanick, Bianchi, Antonio, Invernizzi, Luca, Corbetta, Jacopo, Kirat, Dhilung, Kruegel, Christopher, Vigna, Giovanni, “BareDroid: Large-Scale Analysis of Android Apps on Real Devices,” Proceedings of the Annual Computer Security Applications Conference (ACSAC), December, 2015.

Ozcan, Ahmet T., Gemicioglu, Can, Onarlioglu, Kaan, Weissbacher, Michael, Mulliner, Collin, Robertson, William, Kirda, Engin, “BabelCrypt: The Universal Encryption Layer for Mobile Messaging Applications,” Financial Cryptography and Data Security (FC), Isla Verde, PR, Jan 2015.

Poeplau, Sebastian, Fratantonio, Yanick, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,” Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), February, 2014.

7.0 ACRONYMS/GLOSSARY

API	application program interface
APT	advanced persistent threat
BCEL	bytecode engineering library
DoD	Department of Defense
DOS	denial of service
DVM	Dalvik Virtual Machine
GUI	graphical user interface
HTTP	hypertext transfer protocol
IND-CPA	indistinguishability under chosen-plaintext attack
IR	intermediate representations
JNI	Java Native Interface
JVM	Java Virtual Machine
OS	operating system
SMS	Short Message Service