SAM: The Static Analysis Module of the MAVERIC Mobile App Security Verification Platform

Alessandro Armando[1,2], Gianluca Bocci[3], Giantonio Chiarelli[3], Gabriele Costa[1], Gabriele De Maglie[1], Rocco Mammoliti[3], and Alessio Merlo[1]

[1] DIBRIS, University of Genova, Italy
name.surname@unige.it
[2] Bruno Kessler Foundation, Trento, Italy
armando@fbk.eu
[3] Poste Italiane, Roma, Italy
{boccigi2,chiare96,mammoliti.rocco}@posteitaliane.it

Abstract. The tremendous success of the mobile application paradigm is due to the ease with which new applications are uploaded by developers, distributed through the application markets (e.g. Google Play), and finally installed by the users. Yet, the very same model is causing serious security concerns, since users have no or little means to ascertain the trustworthiness of the applications they install on their devices. To protect their customers, Poste Italiane has defined the Mobile Application Verification Cluster (MAVERIC), a process for the systematic security analysis of third-party mobile apps that leverage the online services provided by the company (e.g. home banking, parcel tracking). We present SAM, a toolkit that supports this process by automating a number of operations including reverse engineering, privilege analysis, and automatic verification of security properties. We introduce the functionalities of SAM through a demonstration of the platform applied to real Android applications.

1 Introduction

Mobile devices are becoming the main access point for many security-critical online services (e.g., e-Banking). Handling valuable resources and data, they are appealing targets for security attacks. In this context, mobile applications represent a major threat. Smartphones retrieve and install software packages from unknown, possibly malicious sources. However, most of the modern mobile operating systems try to regulate the software distribution, therefore mitigating the associated risk, by means of trusted repositories, called application stores, e.g., Google Play and Apple Store. Major service providers, including Poste Italiane, participate in this ecosystem both *directly*, i.e., by publishing their Apps, and *indirectly*, i.e., through third-party apps that access web services offered by Poste Italiane. The ability to tell apart benign applications from malicious or flawed ones is therefore a primary goal for Poste Italiane. In fact,

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 225–230, 2015.
DOI: 10.1007/978-3-662-46681-0_19

the latter malicious or flawed application run on the customer's mobile devices may severely affect the security of the transactions as well as the privacy of the customer. To tackle the challenge Poste Italiane is developing the *Mobile Application Verification Cluster* (MAVERIC), a unified verification framework that provides automated support to a number of key activities ranging from mobile app verification to legal analysis. Security experts at the Poste Italiane Computer Emergency Response Team (CERT) are already using MAVERIC to systematically assess the security of the Poste Italiane mobile apps ecosystem.

In this paper we introduce the *Static Analysis Module* (SAM), a core component of the MAVERIC architecture. SAM integrates some state-of-the-art static analysis techniques for mobile application packages (APKs) and produces a detailed security assessment report containing statistics, properties of the analyzed application, and a number of additional artefacts.

The paper is structured as follows. Section 2 presents the architecture of the SAM, Section 3 describes the components of the module and Section 4 provides a brief overview of the MAVERIC web application. Finally, Section 5 concludes the paper.

2 MAVERIC and SAM

Static software analysis is a complex and multifaceted task. In most cases, static analysis methods have a precise scope. For instance, *malware detection* [6] aims at discovering whether an application carries malicious code. Instead, *code review* [7] applies to software sources for finding flaws in implementations. As they target different aspects and resources, the available static analysis techniques are often complementary and can be combined to extend their potential to new emerging scenarios. SAM integrates different static analysis approaches to support the automatic assessment of Android applications.

The architecture of MAVERIC is depicted in Fig. 1. MAVERIC leverages AppVet [9] to orchestrate a number of fully automated security analysis techniques. AppVet is open-source web service developed by NIST that supports the integration of mobile application analysis tools. It must be noted that AppVet supports basic logging and data management functionalities, but it does not include any analysis component. MAVERIC extends AppVet with several new modules and tools. Among them, SAM implements a set of components supporting the systematic security assessment of Android applications. In the near future, MAVERIC will be extended with further modules targeting other aspects of the security analysis of mobile applications such as the dynamic and legal analyses. Below, we briefly introduce the SAM sub-modules and their role.

Reverse Engineering. It gets the APK file containing the Android app and retrieves general information about the APK and its content (i.e., developer, version, release date, etc.). Moreover, it rebuilds the source code and computes metrics and statistics on it.

Permission Checking. It infers permission requests and usage from both the manifest file and the application code.

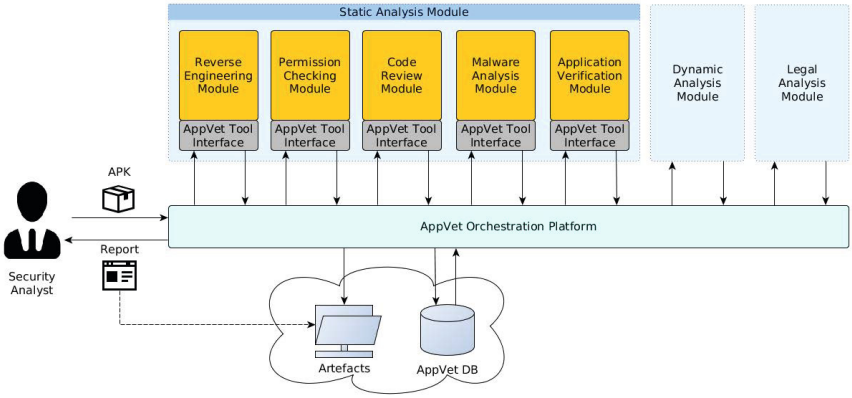

Fig. 1. Architecture of the MAVERIC platform

Code Review. It verifies whether the APK code contains some common vulnerabilities by comparing it with a list of known ones.

Malware Analysis. It processes the APK looking for malware components and known, malicious patterns.

Application Verification. It checks whether the APK complies with a security policy (specified by the analyst).

The results of the analysis are provided back to the analysts in the form of artefacts (e.g. analysis reports). In the next section, we detail the SAM submodules and we report their development status.

3 Static Analysis Techniques

In this section we present the techniques supported by SAM.

Reverse Engineering. The reverse engineering module relies on few tools for APK inspection and Java bytecode decompilation. Used software include Androguard (`https://code.google.com/p/androguard/`), APKTool (`https://code.google.com/p/android-apktool/`), DEX2JAR (`https://code.google.com/p/dex2jar/`) and CFR (`http://www.benf.org/other/cfr/`). This module recreates the resources that the developer used to build the APK. They include source code, configuration files and other resources (multimedia contents, binary data, etc.).

The extracted code is processed for finding whether the application uses *native libraries*, *dynamic class loading* or *code reflection*. Although not always dangerous, these features might cause a security breach. For instance, native code can evade VM security checks (as it is directly executed by the OS).

For each class file, the reverse engineering module returns the size in KB, the number of methods and fields. Also, it assigns an obfuscation score $o \in [0, 1]$

heuristically computed. Intuitively, o indicates the ease to perform a manual inspection of the app code, e.g., $o = 1$ stands for heavily obfuscated code. The heuristic function considers syntactic properties like length and variation of variable, method and class names.

Permission Checking. The permission checking module retrieves and processes the sets of permissions *requested* (R) and *used* (U) by the APK. The elements of R and U are listed along with their *protection level* (obtained from the API specification, see `http://developer.android.com/training/articles/security-tips.html`). The protection level ranges over {*SignatureOrSystem*, *System*, *Dangerous*, *Normal*}. Typically, higher values, e.g., *SignatureOrSystem*, denote permissions needed to access valuable resources or critical functionalities.

The module also computes the relation between U and R. Ideally, applications should statically declare exactly all the permissions they need at runtime, i.e., $U = R$. Instead, if $R \setminus U \neq \emptyset$ some permissions are requested but not used. This means that the application is somehow over-privileged. Although not necessarily dangerous, this case is in contrast with the *least privilege* principle [5]. Finally, if $U \setminus R \neq \emptyset$ some permissions used by the code are not declared. This condition can lead to runtime issues. As a matter of fact, when an unprivileged piece of code attempts an access, a security error is fired. The application carrying these instructions[1], is terminated with a security exception. Although the permissions could be obtained dynamically, e.g., granted by a another app, the application is behaving differently from what is declared in its manifest.

Malware Analysis. Malware detection has a long standing tradition and several approaches exist. A common technique is the *signature-based detection*, consisting in a comparison between application fingerprints against large databases of known malware [8]. Other methods include analysis of program semantics [3] and runtime behavioural checking [10]. These techniques consider different perspectives and can be applied to a single APK for obtaining a multi-dimensional malware profile. The malware analysis module can interact with third-party online malware detection services to do this. For instance, *VirusTotal* (`https://www.virustotal.com/`) is a state-of-the-art web application orchestrating several malware analysis tools and listing their output. Other, similar services are *NVISO ApkScan* (`http://apkscan.nviso.be/`) and *MARBLE Scan* (`http://www.marblesecurity.com/`).

Application Verification. The application verification module exploits *model checking* [4] to verify that an APK complies with a policy defined as a temporal property. The module proceeds by extracting a model of the app and verifying whether it satisfies the policy or not. Models are generated by extracting control flow graphs and by translating them into labeled transition systems. Policies are specified through a specification language called ConSpec [1], i.e., a policy language already exploited in both verification and monitoring frameworks. ConSpec uses a Java-like syntax for defining an abstract security controller.

[1] Notice that if such code is unreachable, the application includes unneeded elements which is often suspect.

The controller consists of a sequence of event-guarded rules. When one of the events takes place, the controller changes its state (defined through a set of variables) according to a statement associated to the rule. Model checking is carried out by SPIN (`http://spinroot.com/`), a state-of-the-art model checker. A similar application verification approach was used in [2] where a prototype implementation analysed hundreds of Android applications against a BYOD policy of the US Government.

Secure Code Review. Code review aims at discovering known vulnerabilities and dangerous code patterns. A source of such patterns is provided by the *OWASP top ten* (available at `https://www.owasp.org/`). Although some of the reported vulnerabilities cannot be detected by only considering a mobile application, e.g., *M1: Weak Server Side Controls*, part of them are localized in the APK code. For instance, *M2: Insecure Data Storage* describes how certain APIs can be misused by applications storing critical data in the file system. The dangerous behaviour can be encoded and verified with techniques analogous to those used for application verification (see above). For the time being, four of the OWASP top ten vulnerabilities have been encoded in ConSpec and are checked against the target applications.

4 MAVERIC Web Application

The MAVERIC platform is available at `https://130.251.1.32:80/maveric`. Anonymous users can log in through the credentials `username: guest` and `password: guest`. After user authentication, the application shows the main screen as depicted in Figure 2.

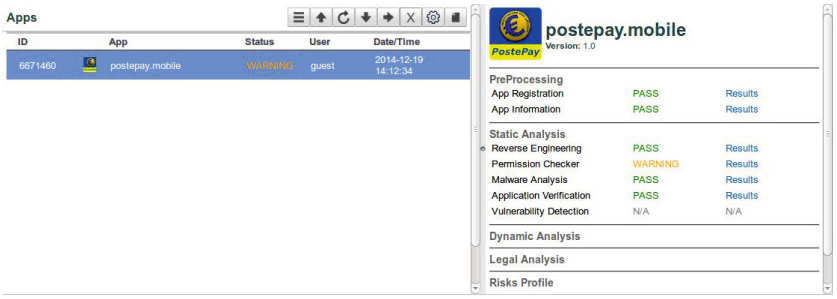

Fig. 2. The main screen of the MAVERIC web application

From the main screen, users can read the existing reports (accessible from the right panel after selecting an entry from the list). Moreover, users can submit new APKs for the analysis. After submitting a new app, the web application displays the progress of the analyses. When one of the sub-modules terminates, its report is accessible through the *Result* link next to the module name.

5 Conclusion

This paper presented SAM, the static analysis module of the MAVERIC platform. SAM provides security analysts with several functionalities for the security assessment of mobile applications. We described each of the components participating in the module and showed how they contribute to the integrated analysis process. Although it is still under development, SAM can be already applied to the security analysis of mobile code.

References

1. Aktug, I., Naliuka, K.: ConSpec – A formal language for policy specification. Science of Computer Programming 74(1-2), 2–12 (2008) Special Issue on Security and Trust
2. Armando, A., Costa, G., Merlo, A., Verderame, L.: Enabling BYOD Through Secure Meta-market. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, WiSec 2014, pp. 219–230. ACM, New York (2014)
3. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-Aware Malware Detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, SP 2005, pp. 32–46. IEEE Computer Society, Washington, DC (2005)
4. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
5. Denning, P.J.: Fault tolerant operating systems. ACM Comput. Surv. 8(4), 359–389 (1976)
6. Idika, M.: A Survey of Malware Detection Techniques. Technical report, Purdue University (February 2007)
7. McGraw, G.: Automated Code Review Tools for Security. Computer 41(12), 108–111 (2008)
8. McGraw, G., Morrisett, G.: Attacking malicious code: A report to the infosec research council. IEEE Softw. 17(5), 33–41 (2000)
9. Quirolgico, S., Voas, J., Kuhn, R.: Vetting Mobile Apps. IT Professional 13(4), 9–11 (2011)
10. Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., Zhou, S.: Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, pp. 265–274. ACM, New York (2002)