Mobile App Security Analysis with the MAVeriC Static Analysis Module

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, Via all'Opera Pia, 13, 16145, Genova, Italy*

{alessandro.armando, gabriele.costa, alessio.merlo}@unige.it

[2]*Security & Trust Unity, Fondazione Bruno Kessler, Via Sommarive 18, 38123, Trento, Italy*

armando@fbk.eu

[3]*Poste Italiane, Roma, Italy*

{boccigi2,chiare96,mammoliti.rocco}@posteitaliane.it

Abstract

The success of the mobile application model is mostly due to the ease with which new applications are uploaded by developers, distributed through the application markets (e.g. Google Play), and installed by users. Yet, the very same model is cause of serious security concerns, since users have no or little means to ascertain the trustworthiness of the applications they install on their devices. Such concerns grow up when dealing with professional scenarios like the use of mobile devices within organisations.

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 leveraging their online services (e.g. home banking, parcel tracking). MAVeriC is an ongoing project that will be completed in the next few years. At the core of the MAVeriC project lies the Static Analysis Module (SAM), a toolkit that supports automatic static analysis of mobile applications by automating a number of operations including reverse engineering, privilege analysis and automatic verification of security properties. In this paper we present the SAM that has been fully developed and tested. We introduce the functionalities of SAM through a demonstration of the platform applied to real Android applications.

Keywords: Android Security, Static Analysis, Malware Analysis, Model Checking, Policy Enforcement.

1 Introduction

Mobile devices are becoming the main access point for many critical infrastructures (e.g., e-Banking systems). Moreover, recent trends like the Bring Your Own Device (BYOD) paradigm (see [1] [2] for some references), whereby employees use their own mobile devices to carry our business tasks, promote the promiscuity of accessed data. As a consequence, there is a growing concern related to the security of mobile devices and the privacy granted to personal and corporate data.

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, e.g., Poste Italiane, participate in this ecosystem both *directly*, i.e., by publishing their Apps,

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 5, number: 4, pp. 103-119

*Corresponding author: Computer Security Lab, DIBRIS, University of Genova, Viale F. Causa, 13, 16145, Italy. Tel: +39-010-353-2344.

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 since the latter can disruptively affect the privacy of the customers as well as the company assets and reputation.

To tackle the challenge Poste Italiane is developing the *Mobile Application Verification Cluster* (MAVeriC), a unified analysis framework that provides automated support to a number of key activities ranging from mobile app verification to legal analysis. Security experts leverage MAVeriC to carry out all the steps necessary for the security assessment of the mobile applications.

In this paper we introduce the core component in the MAVeriC architecture, namely the *static analysis module* (SAM). SAM is responsible for orchestrating and composing several analysis techniques for mobile applications. The SAM gets Android apps in the form of APK files and produces a report and a number of artefacts. Reports contain both properties of and statistics about the analysed applications. For instance, code statistics include the percentage of obfuscated code. Instead, properties can be the absence of known malware, the presence of dynamic or reflected code.

This paper is structured as follows. Section 2 presents the architecture of the static analysis module, Section 3 describes the components of the module and Section4 provides a demonstration of the tool on real Android applications. Finally, Section 5 concludes the paper.

2 Static Analysis Module

Static software analysis is a complex and multifaceted task. In most cases, static analysis methods have a precise scope. For instance, *malware detection* [3] aims at discovering whether an application carries malicious fragments. Instead, *code review* [4] applies to software sources for finding flawed implementations. As they target different aspects and resources, the static analysis techniques are often complementary and can be combined to extend their potential to new emerging scenarios. To this aim, SAM integrates and orchestrates different static analysis approaches to support automatic assessment of Android applications.

Briefly, an orchestrator is responsible for invoking the analysis tools in the right order and composing their output in a single report. AppVet [5] is an orchestration platform devoted to the composition of security analysis techniques. Briefly, AppVet consists of a web application which declares few interfaces for the interaction with the analysis tools. Tools can implement one of two interfaces depending on whether their invocation must be synchronous or not. Simply, synchronous invocations wait for termination while asynchronous ones do not suspend the orchestrator execution. Apart from basic logging and data management functionalities, AppVet does not include any analysis component. In fact, SAM extends AppVet by implementing a set of components supporting the exhaustive security analysis of Android applications. The SAM architecture is sketched in Fig. 1 and the role and purpose of each tool are briefly outlined below.

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.

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

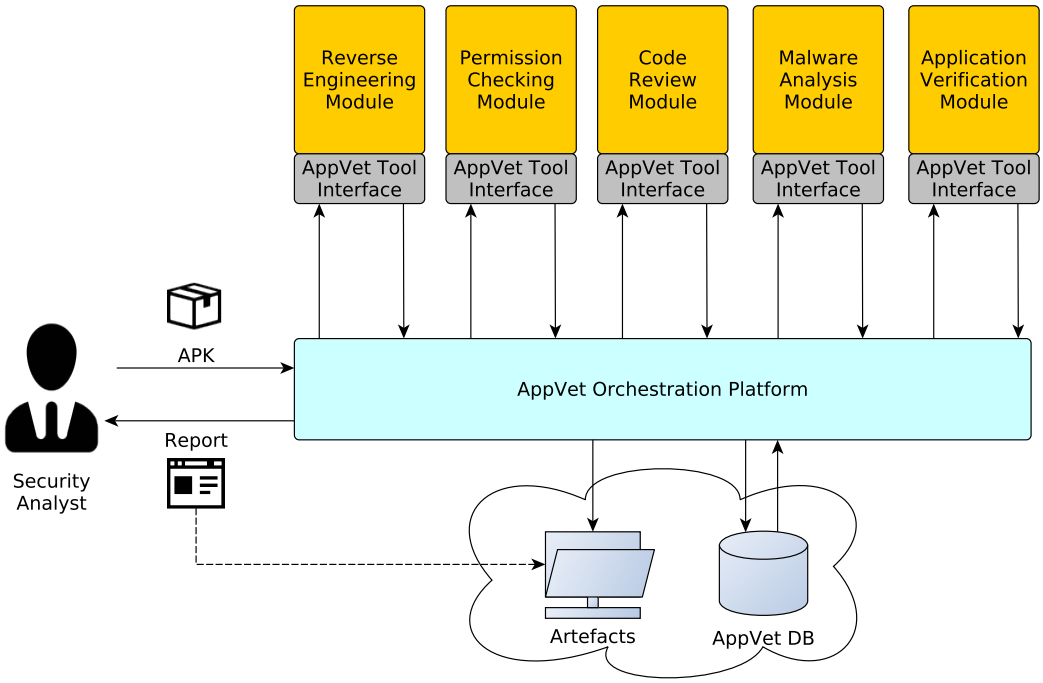

Figure 1: Architecture of the static analysis module.

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

Application Verification. It applies formal verification for guaranteeing that the APK complies with a security policy (specified by the analyst).

The results of the analysis are provided back to the users through artefacts (e.g. analysis reports and data). Artefacts are accessible through the main SAM report. In the next section, we detail the techniques and modules involved in the SAM.

3 Analysis Modules and Techniques

In this section we describe the modules presented in Section 2. For each of them, we point out the used techniques and the produced reports and artefacts.

3.1 Reverse Engineering

The reverse engineering module is responsible for retrieving and reconstructing development data from an APK. Also, the extracted resources are used for computing metrics and statistics. To do that, the module relies on few tools for APK inspection and Java bytecode decompilation. Used software include

Androguard[1], APKTool[2], DEX2JAR[3] and CFR[4].

The artefacts generated by the module are:

- source code of the application (Java classes generated through decompilation);
- manifest file (XML describing application components, permissions and more);
- non-Java resources (other files, e.g., native libraries and multimedia resources).

Reported statistics and metrics include:

- number of classes, methods and fields;
- class files size on disk;
- presence of native code as well as application transformation libraries (i.e., for dynamic class loading and code reflection);
- percentage of obfuscated code and obfuscation score.

Native code refers to instructions (e.g., libraries) which are invoked by the Java code, but executed outside the VM, i.e., directly by the hosting platform. Instead, dynamic class loading and reflection are programming techniques used for importing new instruction and modifying the application at runtime. All these techniques can lead to a new behavior which is difficult (or even impossible) to predict statically. Security analysts should be aware of the presence of these techniques so to better understand the outcome of the static analysis and consider dynamic verification procedures (e.g. application testing) which will be implemented in the MAVeriC platform in the near future.

Obfuscation is a standard approach for reducing the readability of source code and prevent manual code inspection. A classical technique consists in renaming classes and methods using extremely compact and unintelligible identifiers. From this observation, in [6] the authors mark an application as obfuscated by checking whether it declares a class named `a`. Although efficient, this method is extremely coarse-grained. As a matter of fact, applications might contain mixed obfuscated and clear code. Moreover, since obfuscators cannot modify certain instructions, e.g., invocations to system APIs, some fragments of readable code might be present also in obfuscated code.

For these reasons, the reverse engineering module assigns an obfuscation score $o \in [0,1]$ to each source Java file through heuristics. In particular, the value is computed by considering syntactic properties of the source code such as the presence of automatically generated identifiers. Intuitively, o represents the difficulty for a human reader to understand the code contained in a file. For instance, if $o = 1$ the code appears to be heavily obfuscated. This happens, for instance, when all the identifiers (class, field and method names) in the source code are artificially short (i.e., their length is equal to the minimum number of characters sufficient for having enough distinct names). When the values for all the classes are computed, they are also aggregated in an overall *obfuscation profile*. The obfuscation profile is a tuple (α, ω, σ) where α is the percentage of classes having a score greater than the median of all the obfuscation values, ω is the average obfuscation score for these classes and σ is the average obfuscation score for the remaining $1 - \alpha$ classes.

Using similar heuristics, the reverse engineering module also checks whether the application uses obfuscated strings.

[1] `https://code.google.com/p/androguard/`
[2] `https://code.google.com/p/android-apktool/`
[3] `https://code.google.com/p/dex2jar/`
[4] `http://www.benf.org/other/cfr/`

3.2 Permission Checking

Android applications request permissions for gaining the privilege to access some security-relevant resource (e.g., private data or system functionalities). When the user installs an application, he checks the list of requested permissions and grants them (otherwise the installation is cancelled). Sometimes, the application code does not match the requested permissions. As a matter of fact, applications could:

1. require a permission which is not utilized by its code or
2. carry instructions which require undeclared permissions.

Both these scenarios could be relevant for the application analysis. As a matter of fact, over-privileged applications violate the *least privilege* principle [7] and might permit/favour privilege escalation attacks (e.g., see [8]). Instead, code without appropriate permissions can cause runtime errors[5]. In both cases the actual behavior of the application does not match with the declared one, thereby resulting potentially dangerous.

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[6]). In current Android implementations there exist four levels exist (from lower to the higher): *normal*, *dangerous*, *signature* and *signatureOrSystem*. According to the Android documentation, the protection level is higher when a permission can be used to access valuable/critical resources or data. For instance, the protection level for the permission ACCESS_WIFI_STATE (which allows applications for checking whether the device is connected to a wireless network) is *normal*, while the level for READ_WIFI_CREDENTIAL (which provides access to wireless network credentials) is *signatureOrSystem*.

The module also compute the relation between U and R. If $U = R$, all the permissions that might be used at runtime are statically declared. Instead, if $R \setminus U \neq \emptyset$ some permissions are requested but not actually used. Finally, if $U \setminus R \neq \emptyset$ some permissions used by the code are not declared.

Moreover, each instruction that requires a permission is listed in the report for permitting a direct inspection of the code. A table is populated with rows containing *(i)* the referred permission *(ii)* the invocation requiring it *(iii)* the class and *(iv)* method containing the invocation.

3.3 Malware Analysis

Malware detection has a long standing tradition and several approaches exist (see [3] for a survey). Albeit a comprehensive dissertation on malware analysis on Android is out of the scope of this paper, we discuss some very recent proposals with the aim to characterize the most promising techniques that can be adopted in the SAM.

In Android, malware detection solutions can be characterized in terms of i) detection approach, ii) functionality, iii) analysis techniques, iv) analysis location and v) data storage.

Regarding i), the malware analysis can be executed through static or dynamic analysis. Static analysis is carried out on the application package before installation, while dynamic analysis is performed during the execution of the application.

Functionality refers to the kind of detection that can be *signature-based* or *behavioral-based*. *Signature-based detection* consists of a comparison of the application fingerprints against large databases of known malware [9] [10] [11]. Signature-based detection is computationally efficient but lacks in the possibility

[5]Notice that if such code is unreachable, the application is carrying instructions which are never used or loaded dynamically (see Section 3.1). Dynamic loading could even involve a second application which serves as a code proxy.

[6]`http://developer.android.com/training/articles/security-tips.html`

to identify previously unseen malware. On the other hand, *behavioral-based detection* [12] is based on learning the normal behaviors of both the system and benign applications with the aim of recognizing anomalous, unattended behaviors. Signature-based detection can lead to *false negatives* (new malware that are not recognized according to existing signature), while behavioral-based detection can lead to *false positive* (i.e. it may recognize anomalous behaviors that are not malicious at all).

Analysis techniques refer to the technologies adopted to model and evaluate both benign applications and malware. The most promising approaches are 1) Statistical, 2) Spectral, 3) Streaming, and 4) Machine Learning.

Statistical approaches [13] are based on stochastic models. Normal data are expected to fall with high probability within the domain of a stochastic model while anomalous data are likely to fall outside the same model. Thus, statistical approaches, based on properly defined stochastic models, are basically used for behavioral analysis.

The Spectral approach deals with the complexity of data and is based on the idea of geometric reduction [14]. More specifically, some data instances can be inherently dependent. To this aim, spectral approaches tend to combine dependent data and features to reduce to only independent ones. This allows reducing the complexity of both modeling and detection techniques. Spectral approaches can be applied both to signature-based and behavioral-based detection.

The Streaming approach considers data instances as continuous flows and applies discrete algorithms to detect anomalies in each of these flows. In this approach no sampling for categorizing "normal" data is performed but the detection is executed at real-time without any previous knowledge [15].

The main part of malware detectors in Android are based on machine learning techniques. This is due to the fact that machine learning algorithms increase their ability to distinguish between normal and anomalous behaviors during their execution. An introduction to the numerous available techniques can be found in [16] and in the reference therein [7].

The analysis location indicates where the analysis is actually executed. When dealing with mobile devices, the analysis can be performed *on-line*, namely directly on the device by means of proper applications (e.g. anti-virus), or *off-line* through external mechanisms (e.g. a detection server).

Finally, data storage refers to the location where data analysis are permanently stored. Also in this case, they can be stored on the device only (*on device*) or they can be stored on external servers (*off device*). In the former case, the user privacy is preserved but the detection may be less reliable. On the contrary, harvesting data from different devices and storing them on an external server may allow finer-grained and more efficient analyses.

Malware detection on Android is as old as the Android platform itself. However, in the last two years some very promising approaches to malware detection have been proposed. We argue that some of this approaches can be adopted in the MAVeriC platform for the malware detection phase. Since the MAVeriC platform currently supports only the static analysis of applications through the SAM, we focus on emerging static analysis solutions for malware detection.

DREBIN [18] and DroidDetective [19] are static analysis tools performing malware detection specifically on Android devices. DREBIN executes a behavioral based analysis by statistically inspecting the bytecode of Android applications and applies Support Vector Machines techniques to separate malware from normal behavior of Android applications. It operates in a online way and directly on the mobile device. DREBIN is a promising solution for the SAM malware analysis module since it is able to analyze applications in terms of restricted API calls, suspicious API calls, application components, intents, requested permissions and MAC addresses. Furthermore, the detection efficacy of DREBIN has been experimentally proved to be higher than other previous existing tools like sKirin[20] and RCP [21].

[7] A comprehensive discussion on analysis techniques is out of the scope of this paper but the interested reader can also refer to [17]

DroidDetective executes rule-based malware detection by decompiling application and statistically sampling the exploitation of its permissions. More specifically, DroidDetective infers the set of permissions of an application as well as the frequency of each permission exploitation from the bytecode. Then, it compares such frequencies with those of known malware to detect similar behaviors in the analyzed application. In case, DroidDetective recognizes the application as a potential new malware. Differently from DREBIN, DroidDetective operates off-line and off-device and directly on the *.apk* file, making it particularly suitable for the MAVeriC architecture.

Beyond such full-fledged tools, in the last year new static analysis techniques for malware detection have been proposed. For instance, in [19] authors proposes a statistical approach based on a probabilistic discriminative model. In detail, the proposed solution decompiles Android applications and builds up statistics related to the invocation of API calls and permission exploitation. Then, authors applies heuristics to distinguish among malware and benign applications according to proper thresholds. The proposed approach is not applicable directly on the device, thereby requiring to rely on external server for data gathering and detection. Authors state that their approach outperforms all previous machine learning-based proposals.

A different perspective is adopted in [22]. Here the aim is to build a very fast (albeit off-device) malware detection technique by applying random decision trees based on rules specialised on a reduced set of features (2-3 features at most). The solution restricts the analysis to a subset of the features of an Android application, namely the native code, the broadcast receivers and the application permissions.

All previous solutions can be adopted in the malware analysis module. Furthermore, they can be combined to enhance the detection capability by mean of *VirusTotal*[8]. VirusTotal is a web application currently deployed in the SAM that allows orchestrating several malware analysis tools and listing their output. Other, similar orchestration services that can be ported on the SAM are *NVISO ApkScan*[9] and *MARBLE Scan*[10]. However, the current deployment of the SAM does not adopt such solutions.

3.4 Application Verification

The application verification module exploits *model checking* [23] to verify that an APK complies with a given security policy. The verification process consists of *(i)* model generation, *(ii)* model composition and *(iii)* model checking.

For each application method a control flow graph (CFG) is generated according to [24]. Briefly, CFGs are data structures representing the behaviour of a piece of code in terms of its execution flows. If a method contains some security-relevant operation (according to the security policy specification – see below), corresponding labels are attached to the nodes of its CFG. Instead, if no such operation is contained in the code, the resulting CFG is empty.

Model composition consists in combining all the CFG generated from the application methods in few graphs representing the possible execution flows of the application components[11]. To perform the composition, a *call graph* is exploited. Intuitively, a call graph is another data structure which represents which parts of the application invoke each other.

Finally, the models obtained in this way are model checked against the security policy. Policies are defined through a specification language called ConSpec [25], i.e., a policy language that has been already exploited in both verification and monitoring frameworks (e.g., see [26]). ConSpec uses a Java-

[8] https://www.virustotal.com/

[9] http://apkscan.nviso.be/

[10] http://www.marblesecurity.com/

[11] Notice that Android applications do not have a single entry point. Instead they can declare several components, each of them possibly causing distinct executions/computations. See `http://developer.android.com/guide/components/index.html`.

like syntax for defining an abstract security controller. 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.

Both the models and the policy are translated into *Promela*, which is the input format for SPIN[12], a state-of-the-art model checker. SPIN explores the states of the model searching for a policy violation. If a violation exists, a corresponding trace is reported. A trace is a sequence of model states which causes a policy violation.

The application verification approach has been already presented in [27] where a prototype implementation was used to analyse hundreds of Android applications against a BYOD policy of the US Government. Experiments confirmed that applications can be effectively and efficiently verified against arbitrary security policies.

3.5 Secure Code Review

Code review aims at discovering known vulnerabilities, e.g., deriving from dangerous programming patterns. A source of such vulnerabilities is provided by the *OWASP top ten*[13]. Briefly, they include heterogeneous vulnerabilities covering several aspects of the mobile application security. Clearly, some of them cannot be detected by only considering a mobile application as they involve external entities, e.g., *M1: Weak Server Side Controls*. However, many are directly verifiable by considering the mobile code.

For instance, consider *M2: Insecure Data Storage*. It describes how certain APIs can be misused by applications when storing critical data, e.g., on SD memories. The dangerous behavior can be encoded and verified with techniques analogous to those used for application verification (see above). However, since the properties to be verified are statically defined, the process admits optimizations. In particular, security policies can be replaced with sequences of operations mocking the vulnerability. If the code contains one of the sequences, the corresponding vulnerability might be present in the application.

For the time being, vulnerability are manually encoded in our formalism. However, we plan to automatize the process by exploiting existing vulnerabilities databases[14].

4 Tool Demonstration: a real use case scenario

In this section we show how the SAM modules work in practice. The SAM is made publicly available as a web application at `https://130.251.1.32:80/maveric`.
Users can submit Android *apk* packages for analysis and receive back visual feedback and a final report. In detail, we describe the steps that allows to analyze an Android Trojan APK and we inspect the produced output.

Prerequisites. We apply our tool to the static analysis of *iCalendar*[15]. It is a SMS trojan by Zsone that was discovered and removed from Google Market (now Google Play) on May 2011. Briefly, it carries a piece of code sending a single SMS to a phone number subscribing the user to a paid service.

[12] http://spinroot.com/
[13] `https://www.owasp.org/`
[14] At the best of our knowledge, no such database is publicly available for mobile applications vulnerabilities. However, since similar resources exist for other technologies, e.g., see `http://www.exploit-db.com/`, we expect them to appear in the next few years.
[15] Downloadable at `http://www.mediafire.com/?v4c3t2u7zt87eb8`

Note. Other packages can be used in place of iCalendar. For instance, Android APK files can be obtained from Google Play through APK Downloader[16]. Also, (open source) Android packages can be freely downloaded from F-Droid[17].

4.1 Demonstration steps

1. **Access web application.** Users can access the SAM tool at `https://130.251.1.32:80/maveric` by accepting the server certificate putting the following credentials in the authentication page:

   - USERNAME: guest
   - PASSWORD: guest

   After authentication, the browser presents the main screen of AppVet, being the SAM interface. The main screen appears as depicted in Figure 2. The central frame lists previous analyses that

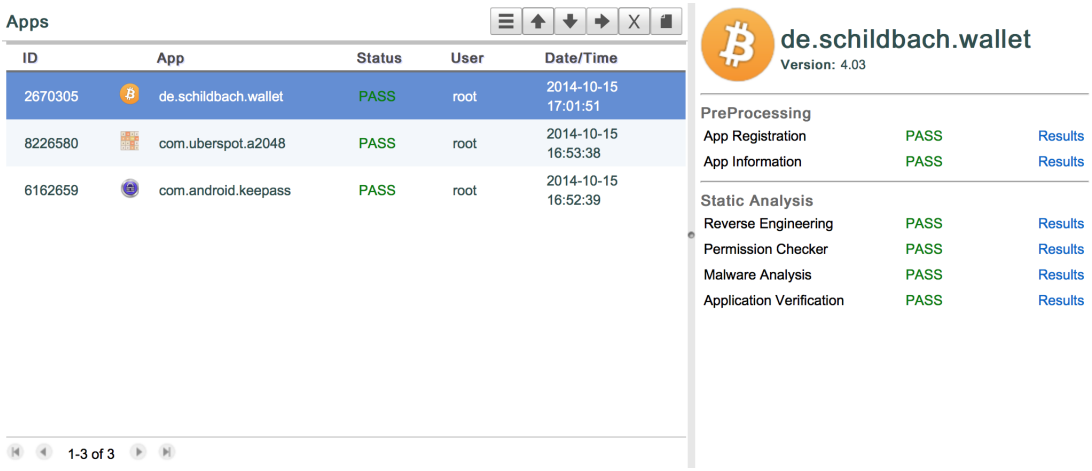

Figure 2: The SAM main page interface.

have been carried out. Results are stored and can be managed as expected (accessed, deleted and overridden). By selecting an element, the right area shows the analysis details (see below).

2. **Submit the APK.** From the main screen users can submit applications by accessing the "Submit App" section and selecting the iCalendar (or any other) APK file from the local disk. The APK will be uploaded to the server.

3. **Wait for termination.** The AppVet orchestrator invokes the SAM tools and waits for their output. During the execution of the tools, their status can be checked in the main screen. The status of a tool can be one of the following.

   - PROCESSING. Indicates that the tool activity as not finished yet.
   - PASS. The tool terminated and the APK passed the analysis.

[16] `https://lekensteyn.nl/apk-downloader/`
[17] https://f-droid.org/

- WARNING. The tool analysis produced warning messages.
- FAIL. The submitted APK did not pass the analysis.
- ERROR. The tool could not complete its computation correctly.

When a tool terminates, its output can be inspected by clicking the associated "Results" link on the right. The user has to wait until all the tools terminate, i.e., their status is not PROCESSING. The right column of the main screen should appear as in Figure 3.

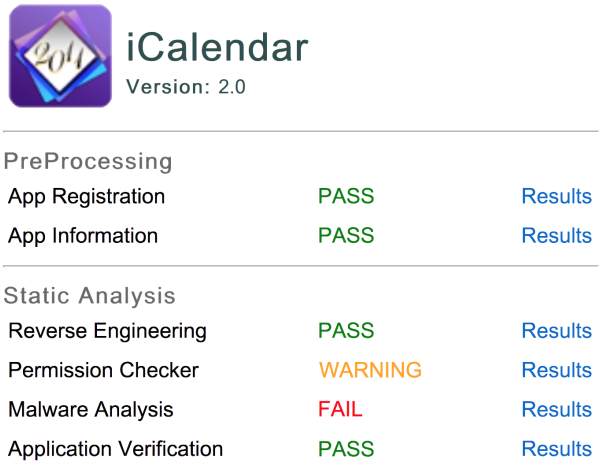

Figure 3: Tools status after analysis termination.

4. **Inspect Malware Analysis report.** Since iCalendar is a known malware, we expect it to be identified by most of the malware detection engines. As a result, the Malware Analysis tool status should report WARNING after termination. From the SAM main page results can be accessed at "Results" link of the Malware Analysis tool. The tool report contains two sections.

   (a) **Preamble.** displays generic information about the analysis.

   (b) **Malware table.** The table presents the output returned by the malware engines. For each of them the table reports the engine details (i.e., name, version and last update), a boolean flag indicating whether the APK was tagged as malware and the name of the detected malware (if one was found). Figure 4 shows (part of) the malware analysis report generated for iCalendar. As expected, most of the malware engines (41 over 53) recognized the Trojan software.

5. **Inspect Permission Checking report.** Now we want to inspect the application resources and code in order to find the known malicious behaviour (i.e., the SMS sending procedure). Since SMS usage requires specific permissions, we exploit the permission analysis for finding the suspect segments of code. The report can be accessed in the SAM main page by clicking on the "Results" link of the Permission Checker tool. The browser displays the report which consists of three blocks.

   (a) **Preamble.** Contains generic information on the APK and its permissions. Figure 5 depicts the preamble generated for iCalendar. Briefly, it requests 6 permissions (3 of them

File name: iCalendar.apk
File size: 764 KB
Analysis time: 17:44:45
Analysis date: 15/10/2014

Number of positive matches: 41 out of 53 antimalware engines.

Antimalware Engine	Malware detected	Malware type	Engine version	Last engine update
Bkav	true	MW.Clod26f.Trojan.4e44	1.3.0.4959	20141014
MicroWorld-eScan	true	Android.Trojan.Zsone.A	12.0.250.0	20141015
nProtect	false	N/A	2014-10-14.01	20141015
CMC	false	N/A	1.1.0.977	20141013
CAT-QuickHeal	true	Android.Raden.A	14.00	20141015
McAfee	true	Artemis!ACBCAD45094D	6.0.5.614	20141015

Figure 4: Malware Analysis report.

being actually unused, i.e., ACCESS_COARSE_LOCATION, RESTART_PACKAGES and RECEIVE_SMS). Also, the code contains invocations requiring undeclared permissions ACCESS_FINE_LOCATION and VIBRATE.

Package: com.mj.iCalendar
App version: 2.0
File name: iCalendar.apk
File size: 764 KB
Analysis time: 17:44:34
Analysis date: 15/10/2014

Permissions number: 6

The following permissions are used but not requested from the application:

- ACCESS_FINE_LOCATION
- VIBRATE

The following permissions are requested but not used from the application:

- ACCESS_COARSE_LOCATION
- RESTART_PACKAGES
- RECEIVE_SMS

Figure 5: Permission Checking report: preamble.

(b) **Description table.** A table listing name, protection level and description of each permission referred by the application. Among them we find the row for SEND_SMS (being labeled as *dangerous*).

(c) **Instruction table.** A list of all the operations requiring privileges that appear in the application code. Each row indicates the security-relevant API, the permission it requires, the class and the method invoking it. Figure 6 shows an excerpt of the privileged instructions table. From it we can see that the invocations using the SEND_SMS permission are SmsManager.getDefault and SmsManager.sendTextMessage. Both of them are located in the class iCalendar (method *sendSms*).

6. **Inspect Reverse Engineering report.** To inspect the report, it is necessary to return to the SAM main page and click on the "Results" link of the Reverse Engineering tool for visualizing its report. The report consists of four blocks.

Permission name	Privileged operation	Invoking class	Invoking method
ACCESS_FINE_LOCATION	android.location.LocationManager -> requestLocationUpdates(java.lang.String, long, float, android.location.LocationListener, android.os.Looper)	com.admob.android.ads.AdManager	getCoordinates(android.content.Context)
SEND_SMS	android.telephony.gsm.SmsManager -> getDefault( )	com.mj.iCalendar.iCalendar	sendSms( )
SEND_SMS	android.telephony.gsm.SmsManager -> sendTextMessage(java.lang.String, java.lang.String, java.lang.String, android.app.PendingIntent, android.app.PendingIntent)	com.mj.iCalendar.iCalendar	sendSms( )
VIBRATE	android.media.AudioManager -> getRingerMode( )	com.admob.android.ads.v	d( )

Figure 6: Permission Checking report: instruction table.

(a) **Preamble.** Contains generic application information and statistics. Figure 7 shows the

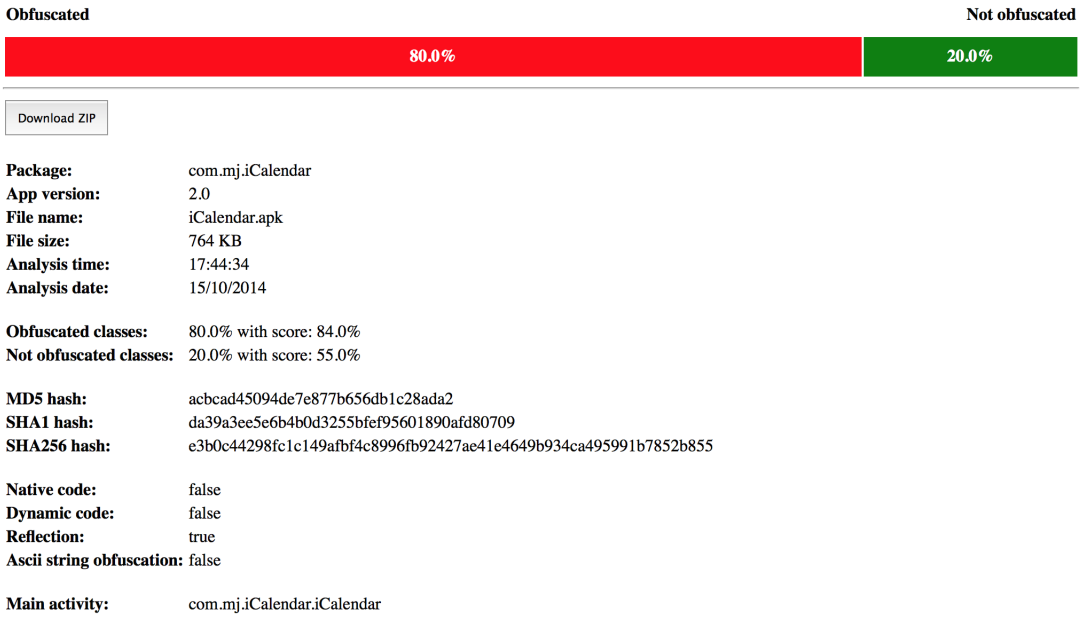

Figure 7: Reverse Engineering report: preamble.

preamble obtained for iCalendar.

(b) **Components.** Lists the interface components, i.e., activities, content providers, broadcast receivers and services, declared by the application.

(c) **Class table.** It is a table containing an entry for each Java extracted class. Each row reports the source file name, its individual obfuscation score, numbers of fields and methods and size on disk. To access detailed information, the user has to locate the iCalendar class containing the suspect method *sendSms* (see above), as depicted in Figure 8, and click on the class name to load its code in the Artefacts block (next step).

(d) **Artefacts.** This part consists of a browsable area showing the artefacts directory and the selected file content. Figure 9 shows the source code of the class iCalendar where the illegal SMS sending operation has been highlighted. By inspecting it we can easily distinguish the phone number of the paid service `"1066185829"` and the activation code `"921X1"`.

Path	Obfuscation score	Methods number	Fields number	Size (Bytes)
com/mj/iCalendar/iCalendar.java	0	20	20	7413
com/mj/iCalendar/R.java	0	1	0	442
com/mj/iCalendar/SmsReceiver.java	0	2	1	1288
com/admob/android/ads/l.java	100	2	0	148
com/admob/android/ads/k.java	100	33	10	9045

Figure 8: Reverse Engineering report: class table.

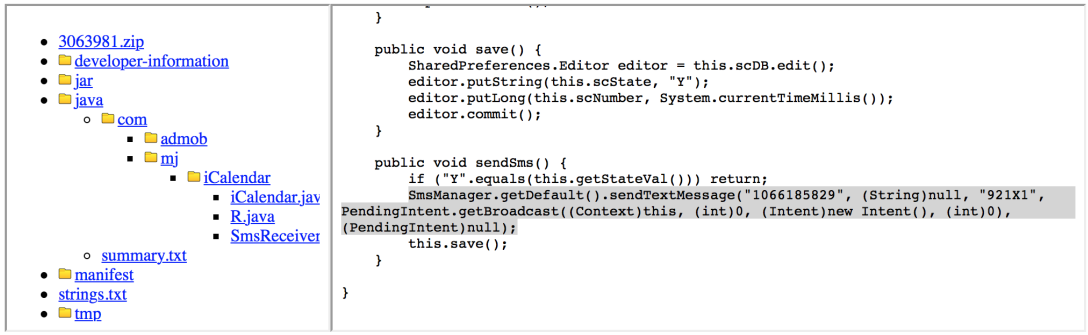

Figure 9: Reverse Engineering report: artefacts block.

7. **Inspect Application Verification report.** From the SAM main page, the user must click on the "Results" link of the Application verification tool. Its report consists of three blocks.

   (a) **Preamble.** Contains generic application information and statistics. Figure 10 shows the

File name: iCalendar.apk
File size: 764 KB
Analysis time: 17:35:29
Analysis date: 15/10/2014

Spin model checker version: 6.2.7 -- 2 March 2014

Figure 10: Application Verification report: preamble.

   preamble obtained for iCalendar.

   (b) **Policy.** Displays the security policy that has been considered during the verification. Figure 11 shows an excerpt of the security ConSpec policy used by the tool. In the current implementation of MAVeriC the policy fixed but the possibility to write down customized policies is under development. The current policy is derived from the e US Government BYOD Security Guidelines[18].

   (c) **Result.** It summarizes the output of the verification activity. Since iCalendar does not violate the given policy, the verification process returns 0 errors.

[18]Digital Services Advisory Group and Federal Chief Information Officers Council. Bring Your Own Device – A Toolkit to Support Federal Agencies Implementing Bring Your Own Device (BYOD) Programs. Technical report, White House, August 2013. Available at `http://www.whitehouse.gov/digitalgov/bringyour-own-device`.

```
SECURITY STATE
  SESSION Str[11] agency_host = "agency.gov/";
  SESSION Obj agency_url = null;
  SESSION Bool connected = false;

AFTER Obj url = java.net.URL.(Str[64] spec) PERFORM
  (spec.startWith(agency_host)) -> { agency_url = url; }
  ELSE -> { skip; }

AFTER Obj url2 = java.net.URL.(Str[0] protocol, Str[64] host, Nat[0] port, Str[0] file) PERFORM
  (host.startWith(agency_host)) -> { agency_url = url2; }
  ELSE -> { skip; }

BEFORE java.net.URL.openConnection() PERFORM
  (this.equals(agency_url)) -> { connected = true; }
  ELSE -> { skip; }

BEFORE java.io.FileOutputStream.write(*) PERFORM
  (!connected) -> { skip; }

BEFORE  android.bluetooth.BluetoothSocket.getOutputStream() PERFORM
  (!connected) -> { skip; }
```

Figure 11: Application Verification report: policy.

5 Conclusion

This paper presented SAM, the static analysis module of the MAVeriC platform. SAM provides the 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 some components are still under development, SAM can be already applied to the security analysis of mobile code. We demonstrated its features by analysing real world Android applications.

The future directions of this activity include the development of a dynamic analysis module as well as the possibility to write down customized policies and plug-in external malware engines. In particular, the dynamic analysis module will include techniques for the security assessment of applications at runtime. Its features will extend the MAVeriC platform and complement those of SAM.

References

[1] A. Armando, G. Costa, and A. Merlo, "Bring your own device, securely," in *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC'13), Coimbra, Portugal*. ACM, March 2013, pp. 1852–1858.

[2] A. Armando, G. Costa, L. Verderame, and A. Merlo, "Securing the "bring your own device" paradigm," *Computer*, vol. 47, no. 6, pp. 48–56, June 2014.

[3] M. Idika, "A Survey of Malware Detection Techniques," Purdue University, Tech. Rep., February 2007.

[4] G. McGraw, "Automated Code Review Tools for Security," *Computer*, vol. 41, no. 12, pp. 108–111, 2008.

[5] S. Quirolgico, J. Voas, and R. Kuhn, "Vetting Mobile Apps," *IT Professional*, vol. 13, no. 4, pp. 9–11, 2011.

[6] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. of the 20th USENIX Conference on Security (SEC'11), San Francisco, California, USA*. USENIX, August 2011, pp. 21–21.

[7] P. J. Denning, "Fault tolerant operating systems," *ACM Computing Surveys*, vol. 8, no. 4, pp. 359–389, December 1976.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proc. of the 13th International Conference on Information Security (ISC'10), Boca Raton, Florida, USA, LNCS*, vol. 6531. Springer-Verlag, October 2011, pp. 346–360.

[9] G. McGraw and G. Morrisett, "Attacking malicious code: A report to the infosec research council," *IEEE Software*, vol. 17, no. 5, pp. 33–41, September 2000.

[10] A. Merlo, M. Migliardi, and P. Fontanelli, "On energy-based profiling of malware in android," in *Proc. of the 2014 International Conference on High Performance Computing Simulation (HPCS'14), Bolgona, Italy*, July 2014, pp. 535–542.

[11] M. Curti, A. Merlo, M. Migliardi, and S. Schiappacasse, "Towards energy-aware intrusion detection systems on mobile devices," in *Proc. of the 2013 International Conference on High Performance Computing and Simulation (HPCS'13), Helsinki, Finland*, July 2013, pp. 289–296.

[12] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou, "Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions," in *Proc. of the 9th ACM Conference on Computer and Communications Security (CCS'02), Washington, D.C., USA*. ACM, November 2002, pp. 265–274.

[13] W. Holmes Finch, J. E. Bolin, and K. Kelley, *Multilevel Modeling Using R*, 1st ed. CRC Press, 2014.

[14] J. Wang, *Geometric Structure of High-Dimensional Data and Dimensionality Reduction*. Springer Publishing Company, Incorporated, 2012.

[15] M. Thottan, G. Liu, and C. Ji, in *Algorithms for Next Generation Networks*, ser. Computer Communications and Networks, G. Cormode and M. Thottan, Eds. Springer-Verlag, 2010, pp. 239–261.

[16] T. M. Mitchell, *Machine Learning*, 1st ed. McGraw-Hill, Inc., 1997.

[17] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, July 2009.

[18] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proc. of the 20th Annual Network & Distributed System Security Symposium (NDSS'14), San Diego, USA*. IEEE, February 2014.

[19] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Proc. of the 2014 IEEE International Conference on Communications (ICC'14), Sidney, Australia.*, June 2014, pp. 2301–2306.

[20] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09), Chicago, Illinois, USA*. ACM, November 2009, pp. 235–245.

[21] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proc. of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12), Newark, New Jersey, USA*. ACM, June 2012, pp. 13–22.

[22] W. Glodek and R. Harang, "Rapid permissions-based detection and analysis of mobile malware using random decision forests," in *Proc. of the 2013 Military Communications Conference (MILCOM'13), San Diego, California, USA*. IEEE, November 2013, pp. 980–985.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, April 1986.

[24] A. Armando, G. Costa, and A. Merlo, "Formal modeling and reasoning about the android security framework," in *Proc. of the 7th International Symposium on Trustworthy Global Computing (TGC'12), Newcastle upon Tyne, UK, LNCS*. Springer-Verlag, September 2012, vol. 8191, pp. 64–81.

[25] I. Aktug and K. Naliuka, "ConSpec – A formal language for policy specification," *Science of Computer Programming*, vol. 74, no. 1–2, pp. 2 – 12, 2008, special Issue on Security and Trust.

[26] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter, "Runtime monitoring for next generation Java ME platform," *Computers & Security*, vol. 29, no. 1, pp. 74–87, 2010.

[27] A. Armando, G. Costa, A. Merlo, and L. Verderame, "Enabling BYOD Through Secure Meta-market," in *Proc. of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec'14), Oxford, UK*. ACM, July 2014, pp. 219–230.

Author Biography

Alessandro Armando is an associate professor at the University of Genova, where he received his Laurea degree in Electronic Engineering in 1988 and his Ph.D in Electronic and Computer Engineering in 1994. His appointments include a postdoctoral research position at the University of Edinburgh (1994-1995) and one as visiting researcher at INRIA-Lorraine in Nancy (1998-1999). He is co-founder and leader of the Computer Security Lab (CSec Lab) at DIBRIS. He is also head of the Security and Trust Research Unit at the Center for Information Technologies of Bruno Kessler Foundation in Trento.

Gianluca Bocci holds a Degree in Electrical Engineering in 1996 from the Sapienza University of Rome and he is CISA, CISM, Lead Auditor ISO/IEC 27001:2013, Lead Auditor ISO/IEC 22301:2012, START Auditor, and ITILv3 certified. He started to work in Digital Equipment Corporation, next acquired by Compaq Computer Corporation and currently become Hewlett Packard, as a Security Solution Architect on many security projects for large enterprise italian Customers. Currently he is a Security professional Master at Information Security department of Poste Italiane S.p.A. in order to support the Computer Emergency Response Team activities with a focus on End User Protection and Mobile Application Security topics. He is also an active member of the Cyber Security Technological District of Poste Italiane S.p.A.. He actively collaborates with Clusit, one of the main Italian Association for Information Security.

Giantonio Chiarelli is a Cyber Security Analyst at the Poste Italiane?s Computer Emergency Response Team where he is involved in incident handling, information sharing and malicious apps detection/analysis also from an R&D perspective. He is ISO/IEC 27001:2013 Lead Auditor and CSA Cloud Star Auditor. He received a bachelor's degree in Computer Science at University of Basilicata, where he contributed to "ClickWorld2", a research project funded by the Italian "Ministry of Education, Universities and Research". He received a master?s degree in Information Engineering, Informatics, and Statistics at University La Sapienza of Rome where he was also member of a research team who contributed to analyze real network traffic exchange patterns from the "NaMeX"; in this context he also made-up a multi-level, incremental and weighted traffic classifier as a part of "ExTrABIRE" a research project funded by the European Commission.

Gabriele Costa is a researcher at the Computer Security Lab, DIBRIS, University of Genoa. His research interests include language-based security, formal methods for security analysis, distributed systems security, and security verification and enforcement. Costa received a PhD in computer science from the University of Pisa, Italy.

Gabriele De Maglie received a M.Sc. in Computer Engineer in 2014 at the University of Genova with a thesis focused on reverse engineering and static analysis of Android applications. It has been the lead developer and tester of the SAM. His main interests are mobile and web security as well as Android and Java development.

Rocco Mammoliti holds a Degree in Electronic Engineering and a Master's Degree in Security from the CASD (Ministry of Defense - Centre for High Studies). He has several years of experience in scientific research activities within the field of Information Security, Engineering and Biomedicine at CNR (Italian National Research Council). He has co-authored several scientific papers on topics related to Nonlinear Time Series Modeling, Multivariate Statistical Data Analysis, Information Security, Cryptography and Data Hiding.

Alessio Merlo got a Laurea degree in Computer Science in 2005 at University of Genova. He received his PhD in Computer Science from University of Genova (Italy) in 2010 where he worked on performance and access control issues related to Grid Computing. He is currently serving as an Assistant Professor at the Computer Security Lab (CSec Lab) at DIBRIS, University of Genoa. His currently research interests are focused on security issues related to Web, distributed systems (BYOD), and mobile (Android platform).