Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework

Martín Barrère, Gaëtan Hurel, Rémi Badonnel, Olivier Festor

Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework

Martín Barrère, Gaëtan Hurel, Rémi Badonnel and Olivier Festor

INRIA Nancy Grand Est - LORIA, France
Email: {barrere, hurel, badonnel, festor}@inria.fr

***Abstract*—Mobile computing devices and the services offered by them are utilized by millions of users on a daily basis. However, they operate in hostile environments getting exposed to a wide variety of threats. Accordingly, vulnerability management mechanisms are highly required. We present in this paper a novel approach for increasing the security of mobile devices by efficiently detecting vulnerable configurations. In that context, we propose a modeling for performing vulnerability assessment activities as well as an OVAL-based distributed framework for ensuring safe configurations within the Android platform. We also describe an implementation prototype and evaluate its performance through an extensive set of experiments.**

I. Introduction

The overwhelming technological advances in the broad sense of mobile computing have made end users to experience real computers in their pockets. Android[1] [1], a Linux-based operating system for mobile devices, is nowadays the election of millions of users as the platform for governing their mobile devices. Only in the first quarter of 2012, worldwide sales of smartphones to end users reached 144.4 million units where Android-based devices leaded the market share owning the 56.1% followed by iOS[2] with 22.9% [5]. However, despite of the many security improvements that have been done since Android's creation, the underlying operating system as well as services and applications have also evolved providing room for new vulnerabilities. Moreover, the open and barely protected mobile environment facilitates attackers to take advantage of such vulnerabilities. Sensitive data handled by mobile users becomes easily exposed. Under this perspective, managing vulnerabilities is a crucial and challenging task that must be addressed in order to ensure safe configurations and to increase the overall security of the system.

Once a vulnerability is discovered in almost any typical software product, its patch cycle normally describes a time gap until the vulnerability is disclosed, another time span until the patch is available and yet another time span until the end user applies the patch [26]. It is usually during this period that attackers activity takes place. Within the Android environment, this issue gets worse. Android is distributed as open source and device manufacturers and telecommunications carriers customize it in order to provide specific services as well as added value to their customers. When a patch is released by Google, an extra time gap will occur until the manufacturer adapts it to work with its own hardware and another time span will pass until the patch is released by the carrier [31]. In addition to this problem, several application markets allow to fast distribute third party applications with only some security checks expecting that the community identifies and reports malicious software. With thousands of applications in the market, Android users are very likely to encounter malware[3] on their devices [7].

Such scenario imperatively requires solutions for rapidly identifying new vulnerabilities and minimizing their impact. Even though no patch might be available for a new vulnerability at a given time, countermeasures can be taken in order to mitigate the problem until the disclosure of an official patch. In that context, vulnerability assessment mechanisms are highly required in order to increase the vulnerability awareness of the system. In addition, mobile devices usually have limited resources thus optimized lightweight tools should be developed to ensure efficiency without losing functionality. Moreover, there are no current solutions built over solid foundations as well as open and mature standards that foster its adoption and speed up general vulnerability information exchange.

In light of this, we propose a novel approach for increasing the security of the Android platform, though it could be applied over other mobile platforms as well, using the OVAL[4] language [14] as a means for describing Android vulnerabilities. We put forward a mathematical model that supports the assessment strategy and a lightweight framework that efficiently takes advantage of such knowledge in order to detect and prevent configuration vulnerabilities. We also present an implementation prototype as well as an extensive set of experiments that shows the feasibility of our solution.

Finally, the remainder of this paper is organized as follows. Section II describes existing work and their limits. Section III presents our approach for modeling the vulnerability assessment process. Section IV details the proposed framework describing its architecture and the proposed strategy for performing self-assessment activities. Section V depicts the internals of our implementation prototype. Section VI shows an extensive set of experiments and the obtained results. Section VII presents conclusions and future work.

[1]Android is developed by Open Handset Alliance, led by Google [11]
[2]Apple iOS [3]
[3]Malicious software including virus, worms and spyware among others
[4]Open Vulnerability and Assessment Language

II. Related work

Android is an open source operating system that integrates some security features by design. It uses the Dalvik virtual machine [4] for executing end user applications written in Java [13]. It is not the same standard Java virtual machine used in most popular platforms such as Linux, Mac OS X or Windows. It has its own API[5] that is almost the same as the standard one. The Dalvik virtual machine takes the Java application classes and translates them into one or more *.dex* (Dalvik Executable) files generating optimized and smaller code. The internal design of the Android platform provides important security features such as the sandbox execution approach [31]. Such approach executes Android applications within separate instances of the Dalvik virtual machine that in turn are represented by different Linux kernel processes. In order to manage the underlying system resources, Android uses an access control policy based on unique identifiers for each application to ensure that they can not interfere between each other.

Despite of the many security features provided by the Android platform [24], [29], end users still face security threats due to existing vulnerabilities within the system itself, misuse of personal data performed by applications and malicious third party software [23], [25]. Several approaches have been proposed for analyzing Android applications and their risks [22], [27]. These contributions provide a strong support for increasing the security of the Android platform. Nevertheless, vulnerability assessment mechanisms have been barely or not at all discussed. Currently, dozens of security applications exist for the Android platform developed by different providers [7], [10], [16]. However, they generally use private knowledge sources as well as their own assessment techniques, and they do not provide standardized and open means for describing and exchanging vulnerability descriptions within the community.

Much of the work done in vulnerability analysis has defined the assessment infrastructure using its own vulnerability specification language arising compatibility and interoperability problems. Languages such as VulnXML [15] have been developed as an attempt to mitigate these problems and to promote the exchange of security information among applications and security entities. However, these languages are only focused on web applications covering a subset of the existing vulnerabilities in current computer systems. In order to cope with these problems, the MITRE corporation [8] has introduced the OVAL language [14], an information security community effort to standardize how to assess and report upon the machine state of computer systems. OVAL is an XML-based language that allows to express specific machine states such as vulnerabilities, configuration settings, patch states. Real analysis is performed by OVAL interpreters such as Ovaldi [12] and XOvaldi [21]. Several related technologies have evolved around the OVAL language. NIST [9] is responsible for the development of emerging technologies including the SCAP[6] protocol [18] and the XCCDF[7] language [32]. The SCAP protocol is a suite of specifications that includes OVAL and XCCDF, and it can be used for several purposes, including automating vulnerability checking, technical control compliance activities, and security measurement. XCCDF is a language for authoring security checklists/benchmarks and for reporting results of checklist evaluation. The use of SCAP, particularly OVAL and XCCDF, not only allows to specify vulnerabilities, but also to bring a system into compliance through the remediation of identified vulnerabilities or misconfigurations. While OVAL provides means for describing specific machine states, XCCDF allows to describe certain actions that should be taken when these states are present on the system under analysis.

Several previous contributions have taken advantage of public vulnerability databases [17] and the use of the OVAL language for performing vulnerability assessment activities in large scale networks [28]. Currently, OVAL repositories offer a wide range of vulnerability descriptions though Android is not yet an official supported platform. In this work, we have instrumented our approach with an experimental OVAL extension for Android within the OVAL Sandbox project [14]. Such extension enables practitioners and experts within the field to specify known vulnerabilities for Android in a machine-readable manner and at the same time, it promotes the exchange and enrichment of Android security information within the community. Our work aims at defining a solution for increasing the security of Android devices by capitalizing Android vulnerability descriptions specified with the OVAL language. Such security advisories are automatically integrated in a distributed architecture where self-assessment activities are performed in order to ensure safe mobile configurations.

III. Vulnerability assessment process model

The process by which vulnerabilities are assessed is critical for efficiently analyzing a target system and minimizing computation costs at the same time. In this section we present a mathematical model that defines and efficiently supports the vulnerability assessment process. Usually, a vulnerability can be understood as a logical combination of properties that if observed in a target system, the security problem associated with such vulnerability is present on that system. Properties can vary depending on the nature of the vulnerability being described, some examples are: a specific process is running (e.g., httpd), a specific port is open (e.g., 80), the system has a specific version (e.g., 2.6.10.rc). Frequently, one property is required by several vulnerability descriptions and naturally one vulnerability description may require several properties. Under this perspective, the set of vulnerability descriptions that constitutes a knowledge base can be compactly represented by using a boolean pattern matrix PM defined as follows:

[5]Application Programming Interface

[6]Security Content Automation Protocol

[7]eXtensible Configuration Checklist Description Format

$$PM = \begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_m \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & \cdots & p_n \end{array} \\ \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \end{array} \quad a_{i,j} \in \{0,1\}$$

Each matrix row encodes the properties required to be observed for the vulnerability v_i to be present. Thus, each entry $a_{i,j}$ denotes if the vulnerability v_i requires the property p_j. Considering for instance a scenario with three vulnerabilities v_1, v_2 and v_3, a pattern matrix PM can be built as follows:

$$\left.\begin{array}{l} v_1 = (p_1, p_3, p_5) \\ v_2 = (p_2, p_4) \\ v_3 = (p_1, p_2, p_5) \end{array}\right\} \quad PM_{3,5} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The pattern matrix can also provide useful information for performing statistics. The $vflatten$ operation aggregates the number of times that each property occurs within the whole set of known vulnerabilities. The resulting vector provides an indicator that helps to identify most common properties involved in vulnerabilities. Such indicator provides valuable information that can be used for closer monitoring and controlling critical components changes.

$$vflatten(PM) = (\sum_{i=1}^{m} a_{i1}, \sum_{i=1}^{m} a_{i2}, \ldots, \sum_{i=1}^{m} a_{in})$$

Other useful metric can be extracted from the pattern matrix when the aggregation operation is performed horizontally, as indicated by $hflatten$. A column vector is obtained from its application where each entry j denotes the amount of properties required by each vulnerability v_j. This metric can be utilized, among other uses, for identifying those vulnerabilities that are most likely affected by changes performed in the environment, thus assessment activities should be taken into account as well.

$$hflatten(PM) = (\sum_{j=1}^{n} a_{1j}, \sum_{j=1}^{n} a_{2j}, \ldots, \sum_{j=1}^{n} a_{mj})^T$$

The state of a system can be encoded in the same manner as done with vulnerabilities, indicating for those properties under control, which ones are present and which ones are not. Thus, a system state is a boolean vector s defined as follows:

$$s = (s_1, s_2, \ldots, s_n) \quad s_i \in \{0,1\}$$

Each entry s_i takes the value 1 if the property p_i is present in the system and 0 if it is not. Considering these constructs, the results of performing the vulnerability assessment process over a given system is defined by the following equation:

$$w = hflatten(PM) - [PM * s^T] \tag{1}$$

$$\Downarrow$$

$$w = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} \\ \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{mj} \end{pmatrix} - \left[\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \times \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \right]$$

The resulting assessment vector $w = (w_1, w_2, \cdots, w_m)$ denotes the status of each vulnerability v_i in the target system. The semantic of the vector w is given by the *Kronecker delta* function as follows:

$$\delta_i = \begin{cases} 0, & \text{if } i \neq 0 \\ 1, & \text{if } i = 0 \end{cases}$$

A null entry w_i indicates that the vulnerability v_i is present in the system while non null values denotes the absence of the corresponding vulnerability. This fact can be understood as a distance metric where a positive value indicates a positive distance between the vulnerability and the target system, and a null distance indicates that the vulnerability is actually in the system. Computing matrix operations in optimized manners constitutes a field that has been studied for years [30]. The integration of the proposed model into real computing systems can take advantage of such expertise providing a compact and efficient representation for performing vulnerability assessment activities.

IV. An OVAL-based framework for assessing Android vulnerabilities

The previous model establishes a well-founded process for assessing vulnerabilities in an efficient manner. By taking advantage of OVAL security advisories, such model can be used for efficiently increasing the security of mobile computing devices. Mobile devices have become a daily useful resource for connecting people, entertainment, working, managing personal data and much more. This fact attracted the attention of legitimate users of these pocket-computers but also from attackers. In only the first semester of 2011, malware for the Android platform has grown at 250% [7]. It is critical to develop open security frameworks that can speed up the knowledge exchange among community users and also being able to take advantage of such information in order to augment their own security. In this section we present our approach for efficiently increasing the security of Android-based devices by automatically evaluating OVAL-based vulnerability descriptions and reporting analysis results.

A. Architecture and main components

We have designed the proposed architecture illustrated in Fig. 1 as a distributed infrastructure composed of three main building blocks: (1) a knowledge source that provides existing security advisories, (2) Android-based devices running a self-assessment service and (3) a reporting system for storing analysis results and performing further analysis. The overall process is defined as follows. Firstly at step 1, the Android device periodically monitors and queries for new vulnerability descriptions updates. This is achieved by using a web service provided by the security advisory provider. At step 2, the

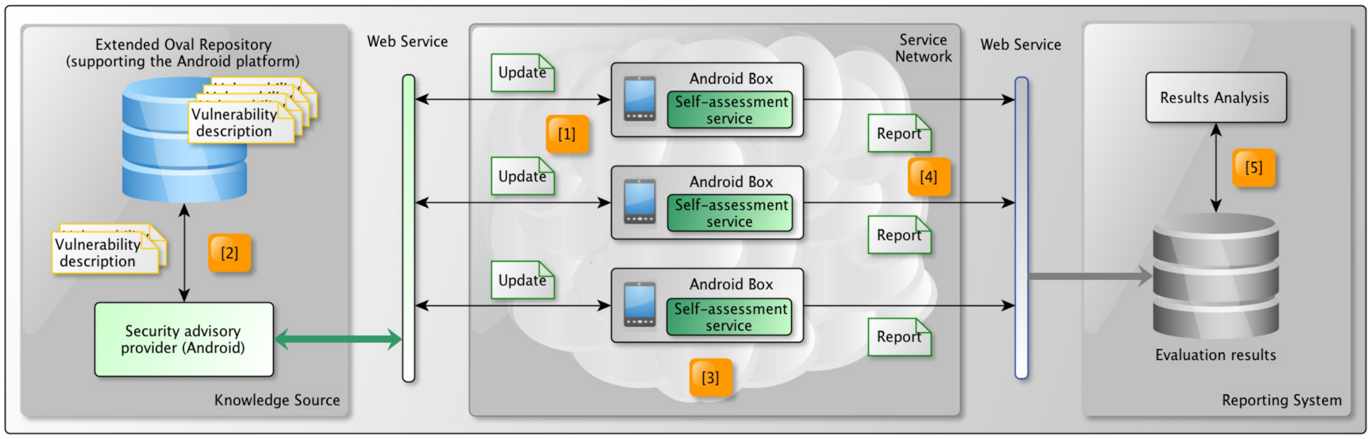

Fig. 1: OVAL-based vulnerability assessment framework for the Android platform

provider examines its database and sends back new found entries. The updater tool running inside the Android device synchronizes then its security advisories. When new information is available or configuration changes occur within the system, a self-assessment service is launched in order to analyze the device at step 3. At step 4, the report containing the collected data and the results of the analyzed vulnerabilities is sent to a reporting system by means of a web service request. At step 5, the obtained results are stored and analyzed to detect potential threats within the Android device. In addition, this information can also be used with different purposes such as forensic activities or statistical analysis.

Within the proposed approach, vulnerabilities are described by using OVAL definitions. An OVAL definition is intended to describe a specific machine state using a logical combination of OVAL tests that must be performed over a host. If such logical combination is observed, then the specified state is present on that host (e.g. vulnerability, specific configuration) [14]. Under a logical perspective, this combination can be understood as a first order formula where each OVAL test corresponds to an atomic unary predicate over that system [19]. The model presented in Section III denotes these predicates as the set of properties $P = \{p_1, p_2, \ldots, p_n\}$. P represents all the predicates (OVAL tests) involved in the vulnerability descriptions (OVAL definitions) available within our knowledge source. In this manner, a boolean matrix PM representing each involved OVAL test for each OVAL definition can be easily built in order to perform assessment activities. The self-assessment component depicted in Fig. 1 constitutes a critical building block because it is in charge of orchestrating the entire lifecycle of the framework in an automatic manner. Hence, optimized algorithms for performing self-assessment activities are highly required. In order to achieve this objective, we have designed and implemented a strategy that uses the model presented in Section III for minimizing the system components required to be assessed.

B. Optimized assessment strategy

Due to the limited resources provided usually by mobile devices, it is important to optimize the use of such elements without losing functionality and performance. The proposed assessment strategy takes this issue into account and minimizes computation costs by using a boolean pattern matrix PM that represents known vulnerabilities and a system state vector s that holds the current system properties. The overall assessment is then efficiently performed using both the pattern matrix and the system vector defined in Section III. Within our approach, two types of events can trigger self-assessment activities: (i) when changes occur in the system and (ii) when new vulnerability definitions are available. Algorithm 1 depicts the overall strategy for treating such events and minimizing the number of OVAL tests to be re-evaluated. In order to explain the proposed algorithm, we put forward an illustrative example that considers both situations and uses the matrix $PM_{3,5}$ illustrated in Section III. Let consider the property $p_2 = \{\textit{Package X has version Y}\}$ and the system state $s = (1, 0, 0, 0, 1)$ meaning that only the properties p_1 and p_5 are present in the system. Within the OVAL language, p_2 is described using an OVAL test that involves an OVAL *package_object* with its attribute *name* = X and an OVAL *package_state* with its attribute *version* = Y.

Let suppose now that an event of type *package_updated* has occurred in the system affecting the package X (line 1). Usually, a complete evaluation of each OVAL definition involving the OVAL test that describes the property p_2 should be carried out. However, only the truth value of the involved OVAL test for p_2 is required for recomputing the results of all the descriptions affected. In order to achieve this, the objects affected by the event are retrieved (line 2) and compared with the objects related to the system properties (lines 3-4). If the object of one property is seen to be affected (line 5), the property represented by an OVAL test is re-evaluated and reflected in the system state (lines 6-7). Within our example, such optimization point will only assess and change the second entry of the system state s. Due to both events are disjoint (system changes at line 1 and definition update at line 11), we now explain the end of the algorithm for the first case and then we discuss the behavior for the second case. Let suppose that the new value for the package version is Y thus the new system state becomes $s = (1, 1, 0, 0, 1)$. Once the

```
    Input: Event event, PatternMatrix matrix, SystemState state
    Output: VulnerabilityList list
 1  if $event$ is of type $SystemChange$ then
 2      $objs \leftarrow getAffectedObjectsByEvent(e)$;
 3      foreach Property $p \in state$ do
 4          $o \leftarrow getObjectFromProperty(p)$;
 5          if $o \in objs$ then
 6              $result \leftarrow evaluateProperty(p)$;
 7              $updateSystemState(state, p, result)$;
 8          end
 9      end
10  end
11  if $event$ is of type $DefinitionUpdate$ then
12      $defs \leftarrow getDefinitionsFromEvent(e)$;
13      $props \leftarrow getPropertiesFromDefinitions(defs)$;
14      foreach Property $p \in props$ do
15          if $p \notin state$ then
16              $addEmptyPropertyColumn(matrix, p)$;
17              $addEmptyPropertyColumn(state, p)$;
18              $result \leftarrow evaluateProperty(p)$;
19              $updateSystemState(state, p, result)$;
20          end
21      end
22      foreach Definition $d \in defs$ do
23          $addAndLoadDefinitionRow(matrix, d)$;
24      end
25  end
26  $w \leftarrow hSumMatrix(matrix) - (matrix * state)$;
27  $index \leftarrow 0$;
28  foreach Entry $v \in w$ do
29      if $v = 0$ then
30          $vulnDef \leftarrow getVulnDef(index)$;
31          $addToOutputList(list, vulnDef)$;
32      end
33      $index \leftarrow index + 1$;
34  end
```

Algorithm 1: Efficient event-based vulnerability assessment algorithm

assessment of the OVAL test for p_2 has been done, the overall assessment result is achieved by performing two operations between boolean matrices (line 26), within our example, as given by Equation 2.

$$w = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix} - \left[\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right] = \begin{pmatrix} 1 \\ 1 \\ \boxed{0} \end{pmatrix} \qquad (2)$$

For each entry in the result vector w (line 28), we use the *Kronecker delta* function (line 29) in order to detect if the vulnerability represented by that entry is present in the target system. If it is the case, the vulnerability definition is added in the output detected vulnerability list (lines 30-31). Within our example, it can be observed that the change performed in the system has exposed itself to new security risks due to the presence of the vulnerability v_3.

The second situation involves the arrival of new vulnerability descriptions (line 11). In this case, both the pattern matrix PM and the system state s have to be extended so as to cover the new properties involved in the OVAL definitions. In order to achieve this, the new definitions are retrieved from the event (line 12), and the properties involved within such definitions are analyzed (lines 13-14). For each uncovered property (line 15), an extension process must be applied. The extension process for the pattern matrix PM will include new columns with null entries for the new properties within existing vulnerability definitions (line 16). The system state s is extended (line 17) and updated as well with the result of the property assessment (lines 18-19). It is important to notice that the arrival of new vulnerability definitions does not imply changes on the system and that the assessment results for known properties are already loaded in the system state, thus there is no need to re-evaluate them again. Finally, for each new vulnerability definition (line 22), a new row is added in the pattern matrix PM indicating the required properties for that vulnerability to be present (line 23). The final assessment procedure is then performed in the same manner as explained in the first situation (lines 26-34). The proposed strategy constitutes a critical part of our framework and it has been integrated into our implementation prototype, which is the heart of the next section.

V. Implementation prototype

In order to provide a computable infrastructure to the proposed approach, a running software component inside Android capable of performing self-assessment activities is required. Currently, 60.3% of Android users operate their devices using *Gingerbread* (versions 2.3.3 to 2.3.7, API level 10) and a total of 79.3% operate versions starting at 2.3.3 until its last release *Jelly Bean* (version 4.1, API level 16) [2]. Our implementation prototype has been developed to be compliant with Android platforms starting at version 2.3.3, thus covering almost 80% of the Android market share. In this section, we describe the prototyping of our solution as well as the high-level operation performed during the assessment activity.

The implementation prototype has been purely written in Java [13] and is composed of four main components: (1) an update system that keeps the internal database up-to-date, (2) a vulnerability management system in charge of orchestrating the assessment activities when required, (3) an OVAL interpreter for the Android platform and (4) a reporting system that stores the analysis results internally and sends them to an external reporting system. Fig. 2 depicts the main operational steps performed during the self-assessment activity and the connection with the mentioned four main components. The prototype is executed as a lightweight service that is running on background and that can be awakened by two potential reasons. The first one is that the update system in charge of monitoring external knowledge sources has obtained new vulnerability definitions; the second one is that changes in the system have occurred hence it is highly possible that some vulnerability definitions need to be re-evaluated. The prototype is still in an early development phase so we only cover some system events such as when a package has been installed.

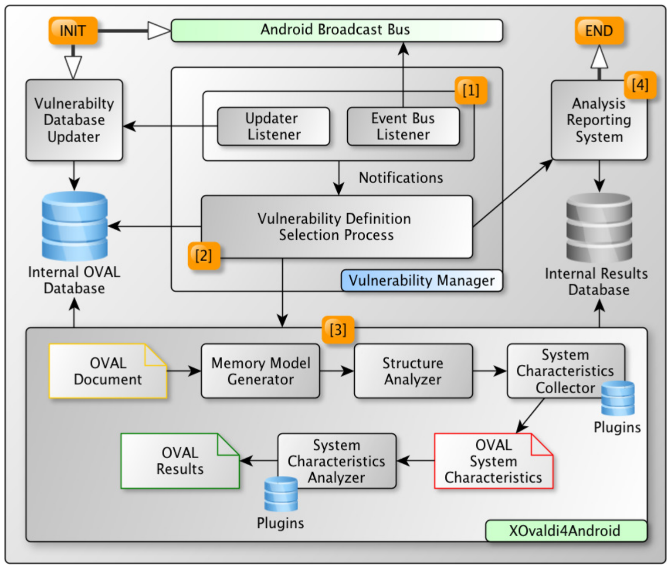

Fig. 2: Self-assessment service high-level operation

In order to be aware of these two potential self-assessment triggers, two listeners remain active as shown at step 1. The updater listener listens the vulnerability database updater component and will be notified when new vulnerability definitions become available. The event bus listener uses the Android broadcast bus to capture notifications about system changes. If new vulnerability definitions are available or system changes have been detected, a vulnerability definition selection process is launched at step 2. This process is in charge of analyzing the cause that has triggered the self-assessment activity and deciding which assessment tasks must be performed by actually implementing the Algorithm 1. At step 3, the vulnerability manager component uses the services of XOvaldi4Android in order to perform the corresponding assessment activity. At step 4, the results of the assessment are stored in the internal results database and sent to the external reporting system by performing a web service request. Finally, a local notification is displayed to the user if new vulnerabilities have been found in the system.

XOvaldi4Android plays a fundamental role within the proposed framework because it is in charge of actually assess the Android system. XOvaldi4Android is an extension of XOvaldi [21], a multi-platform and extensible OVAL interpreter. We have ported the XOvaldi system to the Android platform obtaining a 94 KB size library. We have used the Eclipse development environment and the ADT plugin [2] for Eclipse to easily manage development projects for Android. The interpreter uses the JAXB[8] technology [6] for automatically generating its internal OVAL-based data model. This technology provides means not only for modeling XML documents within a Java application data model but also for automatically reading and writing them. Such feature provides to the interpreter the ability to evolve with new OVAL versions as well as extensions, in this work for the Android platform, with almost no developing cost. As shown in Fig. 2, the high-level operation performed by XOvaldi4Android follows the same assessment process proposed by OVAL. In order to provide extensibility features, the interpreter decouples the analysis of the OVAL structure from the actual collection and evaluation activities by using a plugin repository. While the former is implemented as the core of the interpreter, each plugin provides injectable functionality (collection and evaluation) for the specific type of OVAL test it was built for. In this manner, declarative extensibility of the interpreter is achieved by automatic code generation using the JAXB technology and functional extensibility is supported by its plugin-based architecture.

VI. PERFORMANCE EVALUATION

Devices with limited resources imperatively require well-designed and optimized software that take care of such elements. In this section we present an analytical evaluation of the proposed mathematical model as well as a technical evaluation that involves a comprehensive set of experiments showing the feasibility and scalability of our solution.

A. *Analytical evaluation*

Within the proposed approach, the vulnerability assessment process is governed by Equation (1). Given n as the number of system properties being monitored and m the number of available vulnerability definitions, the complexity of computing the result vector w is $n \times m$. Considering the worst case ($n = m$), the complexity is $O(n^2)$. Being $hflatten(PM)$ a known value, the number of operations performed during the process are n boolean multiplications plus $n - 1$ integer sums for each vulnerability definition. Then, the total number of boolean multiplications is $m \times n$ and the total number of integer sums is $m \times (n - 1)$. Hence, $m \times (n + (n - 1)) \approx n^2$ arithmetic operations are performed for assessing the entire knowledge repository in the worst case.

Considering a knowledge repository with 1000 vulnerability definitions involving 1000 different system properties, the size of the pattern matrix PM is 10^6. This means that the assessment process defined by the model will perform 10^6 arithmetic operations for assessing the entire knowledge base. Considering MFLOPS[9] as the performance measure, though boolean and entire operations are cheaper than floating point operations, the assessment requires 1 MFLOP. Within our experimental devices *Samsung Galaxy Gio* running Android 2.3.3, we have measured an average of 8.936 MFLOPS. With this information, we can infer that a dedicated application of our strategy over a 10^6 size matrix takes less than 1 second in almost any standard Android-based device.

Moreover, latest models may achieve more than 100 MFLOPS meaning that a knowledge source of 10000 vulnerability definitions involving 10000 different properties could be mathematically assessed in less than 1 second. Currently, the OVAL repository [14] offers 8747 UNIX vulnerability

[8]Java Architecture for XML Binding

[9]Million Floating Point Operations Per Second

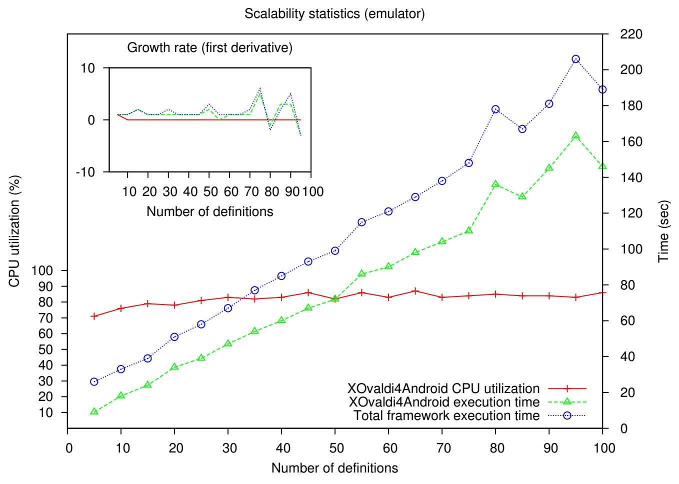

Fig. 3: Scalability statistics in a simulated environment

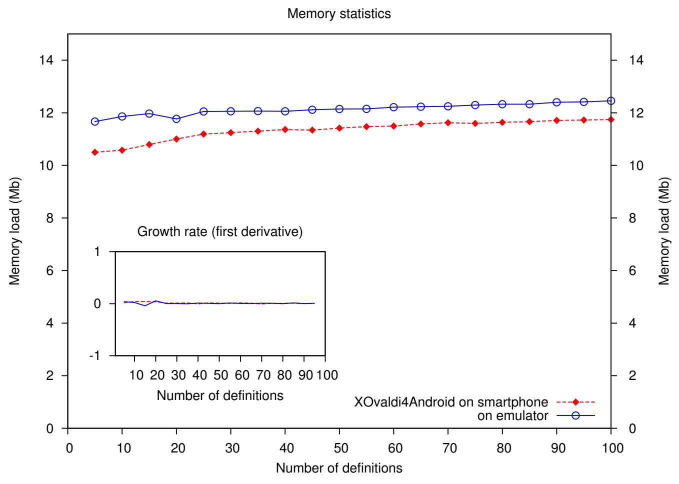

Fig. 5: Memory load in both emulated and real device

definitions including all versions and families after years of contributions made by the community. Such scenario provides real facts making the proposed approach highly suitable for efficiently performing vulnerability assessment activities.

B. Technical experimentation

We have performed several experiments in order to analyze the behavior of our implementation prototype. The proposed methodology cyclically tests the framework without other applications running in foreground. The OVAL definitions set is increased by 5 each time until a set of 100 definitions is evaluated. The used OVAL definitions are similar in size containing in average two OVAL tests. For instance, the vulnerability with the *CVE-2011-3874* id permits a locally installed application to gain root privileges by causing a buffer overflow within *libsysutils*. This vulnerability only affects specific Android versions (first OVAL test) and requires the existence of the library *libsysutils* (second OVAL test). Fig. 3 illustrates the behavior of our implementation prototype over the emulated Android device. We analyze three performance dimensions:

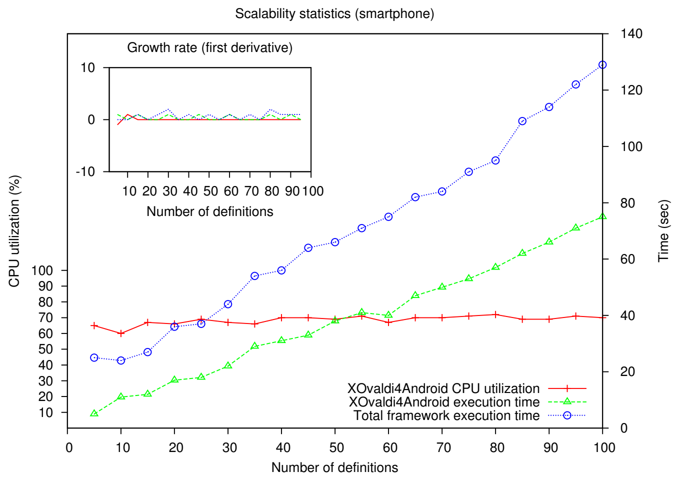

Fig. 4: Scalability statistics in a real device

(1) the CPU utilization when XOvaldi4Android is executed (red solid line with crossings), (2) the XOvaldi4Android execution time (green dashed line with triangular points) and (3) the total framework execution time (blue dashed line with rounded points). During the XOvaldi4Android execution, we have observed a stable and linear behavior in terms of CPU utilization, consuming 80% in average. Its execution time is also stable as shown by the first derivative within the inner graph. While assessing 50 definitions takes about 72 seconds, 100 definitions takes almost twice the time. The overall execution time across the framework, including database updates and reporting results, shows the same behavior though slightly increased in time due to the sequential execution of its components. It is important to notice that these experiments consider extreme cases. As a matter of fact, only new definitions or a small set of definitions affected by system changes will be evaluated in most situations.

In order to analyze the framework behavior using a real device, we have performed the same experiments using a standard smartphone *Samsung Galaxy Gio S5660* (CPU 800 MHz, 278 MB RAM, Android 2.3.3). Fig. 4 illustrates the obtained results. We can observe the same behavior on each curve as with the emulated device, describing a linear growth for each analysis dimension as shown in the inner graph. Nevertheless, we have also detected an improvement in terms of speed and resource usage. The average value for the CPU utilization is now about 65%. In addition, the execution time of XOvaldi4Android is almost half the emulator execution time, taking 38 seconds for analyzing 50 vulnerabilities and 75 for 100 vulnerabilities. This is probably due to a slower emulated CPU. The overall execution time is also reduced due to the faster execution of the vulnerability assessment process. However, its growth rate, though linear, is faster because the internetwork connections are real in this case.

As a final but not less important dimension to analyze, we have experimented with the memory load. Within this analysis, we have considered the allocated memory required by XOvaldi4Android when it is executed. The system classifies the allocated memory in two categories, native and Dalvik, taking in average 40% for native memory and 60% for Dalvik memory. Fig. 5 illustrates the total memory load considering both, over the emulator and the smartphone. We have observed an almost constant utilization of the RAM memory. Within the emulator (blue solid line with rounded points), XOvaldi4Android requires 12 MB in average (4.8 MB of native memory, 7.2 MB of Dalvik memory). Within the smartphone (red dashed line with rhomboid points), XOvaldi4Android requires a little less memory, 11 MB in average (4.4 MB of native memory, 6.6 MB of Dalvik memory).

VII. Conclusions and future work

Vulnerability management constitutes a complex activity that must be addressed in order to increase the overall security of computing devices. In that context, we have proposed an approach for supporting vulnerability assessment tasks as the first key step for integrating this activity within the Android platform. We have put forward a mathematical model as well as an optimized strategy that provides solid foundations for its instantiation on this platform. We have proposed a lightweight framework that enables the integration of OVAL security knowledge into the management plane of mobile Android-based devices. By maintaining low-consumption services monitoring the system, the proposed approach minimizes heavy task executions by only triggering assessment activities when configuration changes are detected or new vulnerability definitions are available. In light of this, we have developed an implementation prototype that efficiently performs self-assessment activities by following the proposed optimized strategy. We have also performed an analytical evaluation of the proposed model as well as an extensive set of technical experiments that shows the feasibility of our solution.

For future work we plan to analyze protection mechanisms of the assessment framework itself as well as collaborative techniques for exchanging security information among neighboring devices over secure channels. We aim at distributing the resulting improved implementation prototype within the community as open source. In addition, botnets such as the one built by the *DroidDream* malware in 2011 are an emerging mobile trend [7]. We also aim at extending our previous work [20] for quantifying compliant network nodes involved in distributed vulnerabilities in order to describe massive attack scenarios within mobile environments. Finally, we state that real autonomy can be achieved if mobile devices are capable of closing the vulnerability lifecycle by performing corrective tasks as well. In that context, we also plan to analyze remediation strategies for correcting vulnerable configurations, leading us closer to get real autonomic solutions.

Acknowledgements

This work was partially supported by the EU FP7 Univerself Project and the FI-WARE PPP.

References

[1] Android. http://www.android.com/. Last visited on August, 2012.
[2] Android Developers. http://developer.android.com/. Last visited on August, 2012.
[3] Apple iOS. http://www.apple.com/ios/. Last visited on August, 2012.
[4] Dalvik Virtual Machine. http://www.dalvikvm.com/. Last visited on August, 2012.
[5] Gartner. http://www.gartner.com. Last visited on August, 2012.
[6] Java Architecture for XML Binding. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/. Last visited on August, 2012.
[7] Lookout Mobile Security. https://www.mylookout.com/mobile-threat-report. Last visited on August, 2012.
[8] MITRE Corporation. http://www.mitre.org/. Last visited on May, 2012.
[9] NIST, National Institute of Standards and Technology. http://www.nist.gov/. Last visited on August 5, 2012.
[10] Norton Mobile Security. http://us.norton.com/norton-mobile-security/. Last visited on August, 2012.
[11] Open Handset Alliance. http://www.openhandsetalliance.com/. Last visited on August 5, 2012.
[12] Ovaldi, the OVAL Interpreter reference implementation. http://oval.mitre.org/language/interpreter.html. Last visited on August, 2012.
[13] The Java Platform. http://www.oracle.com/technetwork/java/. Last visited on August, 2012.
[14] The OVAL Language. http://oval.mitre.org/. Last visited on August, 2012.
[15] VulnXML. http://www.oasis-open.org/committees/download.php/7145/AVDL%20Specification%20V1.pdf. Last visited on August, 2012.
[16] X-Ray for Android. http://www.xray.io/. Last visited on August, 2012.
[17] M. S. Ahmed, E. Al-Shaer, M. M. Taibah, M. Abedin, and L. Khan. Towards Autonomic Risk-aware Security Configuration. *In Proceedings of the IEEE Network Operations and Management Symposium (NOMS'08)*, pages 722–725, April 2008.
[18] J. Banghart and C. Johnson. The Technical Specification for the Security Content Automation Protocol (SCAP). *NIST Special Publication*, 2009.
[19] M. Barrère, R. Badonnel, and O. Festor. Supporting Vulnerability Awareness in Autonomic Networks and Systems with OVAL. *In Proceedings of the 7th IEEE International Conference on Network and Service Management (CNSM'11)*, October 2011.
[20] M. Barrère, R. Badonnel, and O. Festor. Towards the Assessment of Distributed Vulnerabilities in Autonomic Networks and Systems. *In Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'12)*, April 2012.
[21] M. Barrère, G. Betarte, and M. Rodríguez. Towards Machine-assisted Formal Procedures for the Collection of Digital Evidence. In *Proceedings of the 9th Annual International Conference on Privacy, Security and Trust (PST'11)*, pages 32 –35, July 2011.
[22] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. *CoRR*, abs/1206.5829, 2012.
[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.
[24] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50–57, January-February 2009.
[25] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, October 2011.
[26] S. Frei, D. Schatzmann, B. Plattner, and B. Trammel. Modelling the Security Ecosystem - The Dynamics of (In)Security. In *Proceedings of the Workshop on the Economics of Information Security (WEIS'09)*, June 2009.
[27] S. Li. Juxtapp and DStruct: Detection of Similarity Among Android Applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012.
[28] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. *on USENIX Security*, 2005.
[29] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *Security Privacy, IEEE*, 8(2):35–44, March-April 2010.
[30] V. Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.
[31] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong To Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT'11)*, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.
[32] N. Ziring and S. D. Quinn. Specification for the Extensible Configuration Checklist Description Format (XCCDF). *NIST*, March 2012.