A Probabilistic Cost-efficient Approach for Mobile Security Assessment

Martín Barrère, Gaëtan Hurel, Rémi Badonnel and Olivier Festor

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

***Abstract*—The development of mobile technologies and services has contributed to the large-scale deployment of smartphones and tablets. These environments are exposed to a wide range of security attacks and may contain critical information about users such as contact directories and phone calls. Assessing configuration vulnerabilities is a key challenge for maintaining their security, but this activity should be performed in a lightweight manner in order to minimize the impact on their scarce resources. In this paper we present a novel approach for assessing configuration vulnerabilities in mobile devices by using a probabilistic cost-efficient security framework. We put forward a probabilistic assessment strategy supported by a mathematical model and detail our assessment framework based on OVAL vulnerability descriptions. We also describe an implementation prototype and evaluate its feasibility through a comprehensive set of experiments.**

I. Introduction

Nowadays, the use of mobile devices and related services is exponentially increasing. It is more and more common to observe how people get used to take advantage of new incoming mobile technologies. This fact results in a rapid growth of the end-user population and a large-deployment of the entire global mobile network. Indeed, it is expected that the number of mobile-connected devices will exceed the number of people on Earth by the end of 2013 [1]. Our work is particularly focused on the Android platform [2] being currently the leading operating system for smartphones and tablets [3]. End-users rely on these ubiquitous devices for performing dozens of activities that handle a considerable amount of sensitive data. Nevertheless, underlying applications, services and the operating system itself present vulnerabilities that may expose end-users to a wide range of security threats such as denial of service and privacy bypass attacks [4], [5]. The Android market provides users with thousands of applications by using a fast distribution methodology. However, security checks done before releasing third party applications are not sufficient enough making Android users very likely to encounter malicious and vulnerable software on their devices [6]. In addition, vulnerability patch cycles are particularly slower in the Android platform thus increasing their security exposure even more [7].

Mobile devices are widely used with different purposes such as telephony, Internet browsing, handling of personal information, messaging and gaming. In addition, background and transparent services are also executed for controlling the overall behavior of each device. All these activities have a consumption of resources that should be taken to a minimum in order to maximize the performance and responsiveness of these mobile devices. Sometimes users may prefer to deactivate security processes such as antivirus software instead of having a short battery lifetime. This is a blocking point that we are trying to tackle. Indeed, the large-scale deployment of mobile devices combined with present security issues and their limited resources poses hard challenges that must be addressed. Such scenario makes it clear the need for non-invasive, lightweight and effective security solutions able to efficiently increase vulnerability detection capabilities in mobile environments.

In light of this, we propose in this paper a novel approach for performing vulnerability assessment activities on Android-based devices in a cost-efficient manner. The proposed approach centralizes main logistic vulnerability assessment aspects as a service while mobile clients only need to provide the server with required data to analyze known vulnerabilities described with the OVAL[1] language. By configuring the analysis frequency as well as the percentage of vulnerabilities to evaluate at each security assessment, the proposed framework permits to bound client resource allocation and also to outsource the assessment process. Our strategy consists in distributing evaluation activities across time thus alleviating the workload on mobile devices, and simultaneously ensuring a complete and accurate coverage of the vulnerability dataset. This technique results in a faster assessment process, typically done in the cloud, and considerably reduces the resource allocation on the client side. Our main contributions are: (1) a mathematical model that formally supports the proposed approach, (2) an OVAL-based security assessment framework as well as cost-efficient strategies for evaluating known vulnerabilities in Android-based devices, (3) an implementation prototype as well as an extensive set of experiments that shows the feasibility of our approach.

The remainder of this paper is organized as follows. Section II describes existing work and their limits. Section III presents the mathematical model that supports our probabilistic assessment approach. Section IV illustrates our framework describing its architecture and the strategy for performing assessment activities. Section V describes our implementation prototype and the set of experiments performed to validate our solution. Section VI presents conclusions and future work.

[1]Open Vulnerability and Assessment Language [8]

ISBN 978-3-901882-53-1, 9th CNSM and Workshops ©2013 IFIP

II. Related work

Many security features have already been developed within the Android platform [9]. However, there still exist important security issues that must be addressed [4]. In that context, managing vulnerabilities constitutes a critical activity that is composed of three main sub-activities, namely, (I) assessing and identifying vulnerabilities, (II) classifying them, (III) remediating and mitigating found vulnerabilities [10]. Currently, several vulnerability assessment solutions are available for the Android platform [6]. However, these tools do not provide standard means for describing and exchanging the vulnerabilities they are able to assess. Languages such as VulnXML [11] have been developed as an attempt to mitigate these problems but they are only focused on web applications thus only covering a subset of current existing vulnerabilities.

As an effort for standardizing the enumeration of known security vulnerabilities, MITRE Corporation [12] has introduced the CVE[2] language [13]. However, the CVE dictionary only provides means for informing about their existence, not for their assessment. In light of this, MITRE has developed the OVAL language [8] as an effort to standardize the process by which the state of a computer system can be assessed and reported. 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 [8] and XOvaldi [14]. In order to provide an automated and comprehensive security model, NIST [15] has introduced the SCAP[3] protocol [16]. The SCAP protocol includes the OVAL specification but also XCCDF[4] [17] and CVSS[5] [18]. XCCDF is a language conceived as a means for bringing a system into compliance through the remediation of identified vulnerabilities or misconfigurations. CVSS on the other hand provides an open an standardized method for rating IT vulnerabilities.

These technologies have already been used in previous scientific contributions [19]. The OVAL language has been utilized for performing vulnerability assessment activities in large scale networks [20], [21]. However, the vulnerability management process also involves remediation activities when vulnerabilities are found. Therefore, change management techniques are also required for ensuring coherent automated security processes [22], [23]. In our previous work we have proposed a self-assessment solution based on the OVAL language for detecting vulnerabilities on the Android platform [7]. However, the proposed approach does not outsource assessment activities thus potentially requiring considerable resource allocation levels. Our current work aims at reducing the resources affected by assessment activities on target mobile devices and simultaneously ensuring high vulnerability detection accuracy by using a centralized probabilistic approach. Oriented to be a cloud vulnerability assessment service, we have already scheduled large-scale experimentations on mobile cloud computing platforms such as the one proposed in [24].

[2]Common Vulnerabilities and Exposures

[3]Security Content Automation Protocol

III. Probabilistic vulnerability assessment

When developing mobile solutions, limited resources present on mobile devices must be carefully managed in order to maximize the performance and responsiveness of such devices. In that context, the model proposed in this paper aims at minimizing the resource consumption at the target device, e.g. battery, CPU, and at the same time maximizing the vulnerability assessment accuracy.

A. Model overview

Each time a security analysis is made, vulnerabilities descriptions are analyzed in order to detect security weaknesses on a target device. In this work, we use the OVAL language for describing vulnerabilities. Vulnerabilities are represented by means of OVAL definitions. Each OVAL definition logically combines OVAL tests that represent atomic checks or evaluations over the target device. Each OVAL test in turn can be referenced by different OVAL definitions and contains an OVAL object that describes the component to be analyzed, and an OVAL state that describes the properties expected to be observed on the specified component. The test result will be $true$ if the component actually exhibits the specified state, and $false$ otherwise. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of available OVAL tests. Then, the set of known vulnerability descriptions $V = \{v_1, v_2, \ldots, v_m\}$ constituting our knowledge source can be built by respecting the following rules:

i. if $t_i \in T$, then $t_i \in V \quad (i \in \mathbb{N})$
ii. if $\alpha, \beta \in V$, then $(\alpha \diamond \beta) \in V \quad \diamond \in \{\wedge, \vee\}$
iii. if $\alpha \in V$, then $(\neg\alpha) \in V$.

Traditional assessment mechanisms usually evaluate these vulnerabilities in a one-step fashion by analyzing the whole set of vulnerability descriptions at once. Such methodology is highly time and resource-consuming. Our approach aims at dealing with this problem by probabilistically distributing vulnerability assessment activities across time and restricting resources affected by this task. Fig. 1 exemplifies both regular and probabilistic approaches where a set of vulnerabilities involving eight single tests is evaluated during four periods of time. The regular approach analyzes the whole body of vulnerabilities at each period thus evaluating all tests each time. This is accurate but constitutes an extremely heavy task. The probabilistic approach on the other hand selects only a subset of tests to execute in order to cover a subset of vulnerabilities each time. Tests are probabilistically selected according to their utility on the resolution of vulnerability evaluations as well as the elapsed time since their last analysis. The test selection process constitutes the heart of this section and it is detailed in the following subsections. By following this methodology, the probabilistic approach highly reduces the activity load and resource allocation at each security analysis while rapidly converging to a complete assessment of the vulnerability set.

The probabilistic approach is also depicted in Fig. 1 where only tests t_3 and t_4 are evaluated and tagged at period 1. At period 2, tests t_5 and t_6 are evaluated and tagged but also t_4,

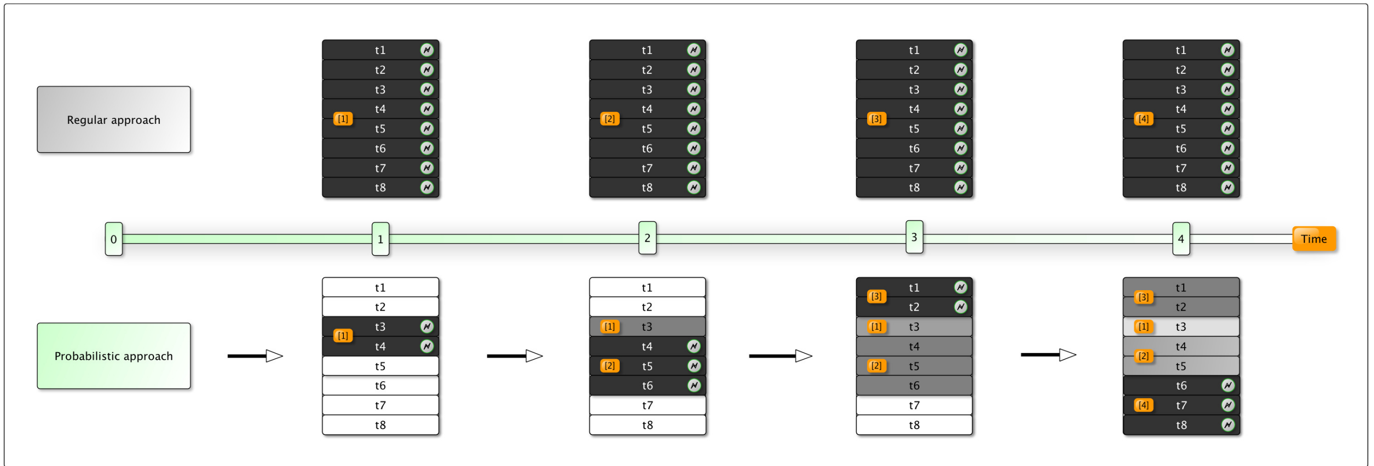

Fig. 1: Regular vs. probabilistic approach

probably due to a high utility value thus being re-evaluated once again. Test t_3 has not been selected at this period thus becoming one period older in terms of its evaluation, illustrated with a less intense grey color. At period 3, tests t_1 and t_2 are evaluated while test t_3 becomes two periods older, and t_4, t_5 and t_6 only one. At period 4, tests t_6, t_7 and t_8 are selected for evaluation thus completing the whole vulnerability assessment. Notice the re-evaluation of t_6, probably due to a high utility value again. The selection process continues like this across time thus t_3, the oldest evaluated test so far, will have a higher probability of being selected but it will still compete with other high utility tests during future selection processes. The idea is that high utility tests are more frequently evaluated but low utility tests are also evaluated as they become older. Therefore, test starvation is avoided ensuring the convergence towards the analysis of the complete set of known vulnerabilities.

The proposed model considers different parameters that allow the user to adapt it according to specific needs, namely, (1) a threshold λ that indicates the percentage of vulnerabilities that must be evaluated at each security analysis, and (2) a time interval δ that specifies the amount of elapsed time between each security analysis. The overall idea is that during each security analysis made with frequency δ, an iterative evaluation process is performed, statistically guided by the utility that each test has over the current vulnerability database as well as the elapsed time since their last evaluation. Tests are probabilistically selected until the desired threshold λ is achieved. In order to minimize the load impact over mobile devices, the process by which tests are selected is critical because of two reasons, firstly it must consider the most useful tests at each security analysis and secondly, it must ensure that all tests will be eventually executed. These concepts are presented in the next subsections.

B. Test utility analysis

Within the proposed model, the utility of a test aims at expressing a metric that combines the ability of this test to speed up the overall evaluation and its security impact on the target system. Such concept relies on: (1) how much the body of vulnerability descriptions can be reduced towards a complete coverage when its value is determined, and (2) the security impact of the vulnerabilities in which the test is involved in. The concept of reduction refers to the idea of how much closer we are to determine the truth value of the vulnerabilities under analysis when a test value is known. For instance, let v be a vulnerability description with the form $v = t_1 \wedge (t_2 \vee t_3)$. If the value of t_1 is known and it is *false*, then there is no need to evaluate t_2 and t_3 as the final value for v will be *false* no matter what values take t_2 and t_3. In this case, the utility of t_1 is higher than the utility of t_2 and t_3 because its evaluation could potentially eliminate the need to evaluate the remaining tests in the formula. The other way around however is not true; if t_2 is *false* then t_3 must be evaluated, if it is *true* then t_1 must be evaluated. No matter what value takes t_2, a second test must always be evaluated. The same phenomenon occurs with t_3. Therefore, t_1 fits better in this situation and it will have a higher utility value than t_2 and t_3. During the reduction process, if the evaluation result of t_1 is $false$ then v will be reduced to $v = false$ thus completing the evaluation. If the result evaluation of t_1 is $true$ instead, then v will be reduced to $v = true \wedge (t_2 \vee t_3) = t_2 \vee t_3$. The process will then continue over t_2 and t_3 until obtaining the truth value for v.

In order to facilitate the quantification of the utility of a test, vulnerabilities are represented as formulas in conjunctive normal form (CNF). A vulnerability expressed in CNF is a conjunction of clauses where each clause is a disjunction of tests as follows:

$$v_i^{CNF} = \bigwedge(\bigvee(t_j | \neg t_j)) \qquad t_j \in T, v_i \in V \tag{1}$$

Accordingly, if the value of a test t is known, its utility over a specific vulnerability database V is expressed by a fitness function U defined as follows:

$$U(t, val, V) = \frac{\sum_{i=1}^{|V|} \left(\frac{testRed(v_i, t, val) * I(v_i)}{totalTests(v_i)} \right)}{|V|} \tag{2}$$

$$t \in T, val \in Boolean, v_i \in V$$

The $testRed$ function represents the number of tests whose truth values do not contribute to the final resolution of v_i when the value of t is val. The $totalTests$ function returns the number of tests involved in v_i. The I function returns a numerical value representing the impact security factor or criticality of v_i, e.g., its CVSS score [18]. Because the function represented by Equation 2 is used for selecting the next test to be executed, the evaluation values for those tests under selection are not known yet. Therefore, we define a weight function W for determining the average utility of a test t over a vulnerability database V as follows:

$$W(t,V) = \frac{U(t,true,V) + U(t,false,V)}{2} \quad t \in T \quad (3)$$

In order to select the next test to be evaluated, tests are sorted by descending utility values producing an ordered list $T_W = \{t_1, t_2, \ldots, t_n\}$. This list provides statistical-based ranking information for unevaluated tests that combined with a temporal factor supports the probabilist test selection process.

C. Probabilistic test selection process

As there is a threshold λ that limits the execution of the whole set of tests, not every test will be executed during a single security analysis. If only best tests were selected at each analysis and the device state remains the same, there would be tests that would never be evaluated. This effect is called test starvation meaning that some tests might never come up with the opportunity to be evaluated because of their low utility values. Therefore, some vulnerabilities might never be covered either. In order to avoid test starvation, we consider two factors that shape the overall behavior of our strategy across time. The first factor is a weighted probability ρ for each test that is directly proportional to its utility value. This means that even when a test has the highest utility value, another test with a lower utility value could be selected in its place for execution. Such approach is less elitist though still fair as it provides the opportunity for lower tests to substitute higher tests with probabilities according to their ranking. In order to specify the probability for a test to be chosen according to its positioning in the weighted list T_W, we define the ρ function as follows:

$$\rho(t,V,T_W) = \frac{W(t,V)}{\sum_{i=1}^{|T_W|} W(t_i,V)} \quad t, t_i \in T_W \quad (4)$$

The second factor to avoid test starvation is the elapsed time τ between each security analysis. The older the last evaluation of a test is, the higher is the chance for this test to be selected. This increase however must consider their ranking status indicated by the first factor ρ in order to respect the statistical analysis done for each test. In order to combine both factors in the selection process, we define the selectivity value for a test t in a given time x by the following equation:

$$S(t,x,V,T_W) = \rho(t,V,T_W) * \tau(t,x) \quad t \in T_W, x \in [0..\infty) \quad (5)$$

The main idea in Equation 5 is to prioritize high impact tests given their weighted probabilities but simultaneously promoting lower tests that turn more important as long as their last evaluations become older. The delta time τ for a test t is considered as the time elapsed between its last evaluation and a specific time x. τ is defined as follows:

$$\tau(t,x) = x - lastEvalTime(t) \quad t \in T_W, x \in [0..\infty) \quad (6)$$

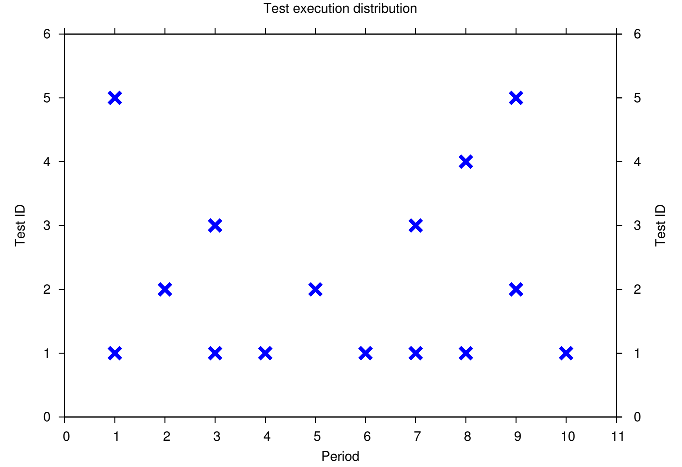

Fig. 2: Test execution distribution

The behavior of the selection process is illustrated in Fig. 2 where five tests constitute the body of known vulnerabilities V and they are assessed over ten periods of time ($\delta = 1$). Tests have been ordered according to their utility values over V, being the first test the most useful test. It can be observed how the test with the highest utility has been selected seven times, much more than the other tests with lower utility values. However, lower utility tests also have been selected though in a lower rate. It can be also noticed that the fourth test is stronger than the fifth test in terms of utility, but in this specific experiment however, the latter shows a higher selection frequency (periods 1 and 9) than the former (period 8). This is an interesting effect due to the probabilistic nature of the process though in the general case, as illustrated later in Section V, the test execution frequency tends to a coherent distribution according to test utility values. In the next section we present Ovaldroid, a probabilistic vulnerability assessment framework that integrates the proposed model in order to increase the overall security of Android devices.

IV. Ovaldroid, a probabilistic vulnerability assessment framework

Ovaldroid is a probabilistic-based framework designed for assessing configuration vulnerabilities over Android devices. We explain here its architecture as well as the underlying strategy that has been cautiously designed for outsourcing as much as possible the involved assessment activities and dealing with issues such as resource usage and ubiquity.

A. Architecture overview

The architecture of Ovaldroid, described in Fig. 3, has been designed as a centralized service-oriented infrastructure capable of analyzing vulnerabilities over Android-based devices. It is composed of two main building blocks, namely,

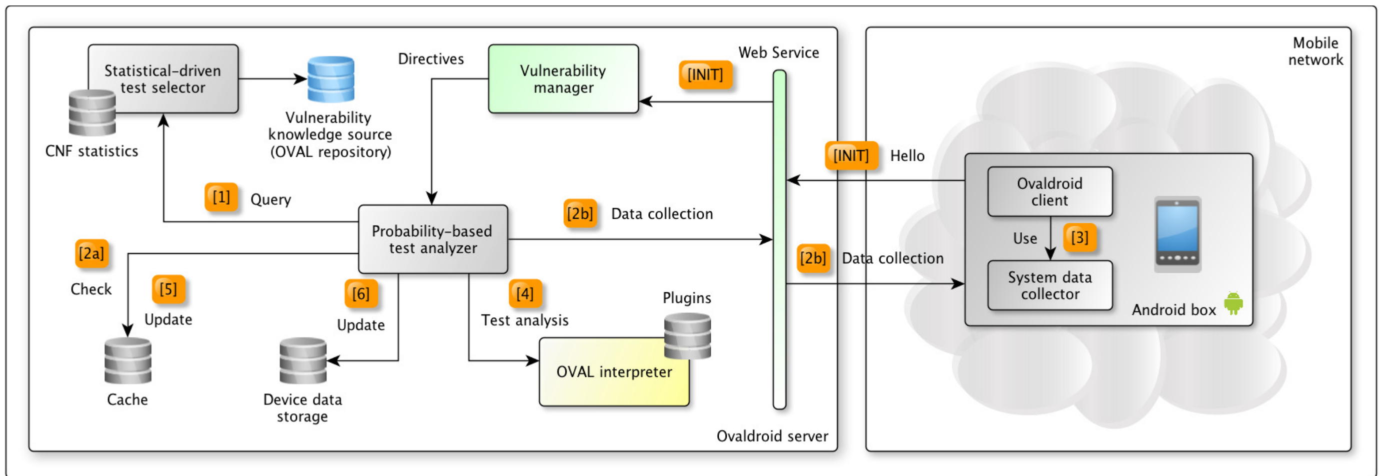

Fig. 3: Ovaldroid global architecture

a server that manages the whole assessment process and clients located on the mobile network that use the vulnerability assessment service. Mobile clients periodically communicate with the Ovaldroid server in order to inform about their assessment availability. This communication is started by the Ovaldroid client that sends an identified *Hello* message using the web-service provided by the server. Based on the historical evaluation registry, the server decides whether it is necessary to perform a new vulnerability assessment based on the pre-established assessment frequency (δ). If it does, the vulnerability manager subsystem located on the server side sends specific directives to the probabilistic-based test analyzer in charge of orchestrating the overall assessment activity. The probability-based test analyzer in turn, executes a sequence of OVAL tests until the specified percentage of vulnerabilities to be evaluated (λ) is reached.

In order to select which OVAL test must be evaluated at each iteration, the analyzer uses the services of the statistical-driven test selector (step 1). The latter builds, at the first call, a local CNF database representing the vulnerability descriptions available in the vulnerability knowledge source. Then, at each query sent by the analyzer, the statistical-driven test selector will produce an ordered list of tests suitable to be performed over the target device based on the impact that each unevaluated test has towards the desired vulnerability coverage. The analyzer then chooses the test to be executed from this list by considering its ranking combined with the elapsed time since its last evaluation as the probability to be selected. This means that high utility tests will be more likely to be selected because of their high ranking values. However, low utility tests still have the opportunity to be selected though in a minor rate.

Once a test has been selected for execution, the analyzer checks if a previous unexpired result for this test exists in the cache (step 2a). If it does, it is directly used thus saving computation resources on the client side. The cache also stores collected objects from previous tests due to sometimes the same object is used by different tests. Therefore, if no result for this test is found, the system looks for an unexpired version of the object previously collected from the device under analysis over which this test applies. If there is a hit, the object is used without interacting with the target device. Otherwise, the analyzer performs a data collection request on the target device (step 2b) in order to gather the required data and assess the corresponding OVAL test on the server side. Cache entries do not affect the test selection process itself because the oldness of these tests is already considered in the model. Therefore, the cache and its policy can be independently set to reduce the load even further on the target device.

Data collection is performed on the client side by running a small lightweight Android application (step 3). Once the required object is available, the services of an OVAL interpreter are used in order to evaluate the selected OVAL test (step 4). Depending on the nature of a vulnerability, different types of tests might be used when describing it, e.g., file tests, process tests, version tests. In that context, the OVAL interpreter uses plugins for each type of OVAL test where each plugin knows how to collect and analyze the information of the type of test it was created for. After the evaluation, the collected object and the test result are stored in the cache for future use (step 5). Finally, the test result is also placed in the results storage system on the server side (step 6). The process continues over steps 1 to 6 until the percentage of vulnerability coverage specified by the administrator is reached. Final assessment results are also saved in the results storage system.

B. Assessment strategy

The proposed methodology integrates a probabilistic component for selecting which tests must be evaluated at each security analysis. However, the spectrum of eligible tests is built following a statistical strategy. The steps followed by the combined assessment strategy are depicted in Algorithm 1. The general process consists in selecting and evaluating tests in the target device until the specified coverage threshold is reached (line 2). At each iteration, a test is selected as described in Section III by considering how much it contributes to achieve the specified coverage, the impact of the vulnerabilities this test participates in, and the elapsed time since its

last evaluation (line 3). The algorithm looks for a previous unexpired evaluation result of this test in the cache (line 4). If a result is found, it is directly used (line 5). If it is not, the object referenced by this test is searched in the cache (line 8). If the object is found (line 9), it is directly used. If it is not, the data collection process is launched over the target device (line 11). After a cache hit or the collection process itself, the evaluation process is performed (line 13) and the cache is updated with the collected object and the result (line 14). Current results are then updated in the general assessment results (line 16). Considering these results and the remaining tests to be assessed, the vulnerability list is reduced as explained in Section III-B by replacing known test values within the CNF formulas that represent such vulnerabilities (line 17). Finally, the vulnerability coverage obtained until this point is updated (line 18). The algorithm ends when the percentage of assessed vulnerabilities satisfies the specified threshold.

```
Input: CNFVulnList vulnList, Threshold threshold
Output: AssessmentResults results
1  coverage ← 0;
2  while coverage < threshold do
3      test ← computeBestUtilityTest(vulnList);
4      if test in cache then
5          testResult ← getResultFromCache(test);
6      else
7          object ← getObjectDescription(test);
8          if object in cache then
9              objectData ← getFromCache(object);
10         else
11             objectData ← collectFromDevice(object);
12         end
13         testResult ← evaluate(test, objectData);
14         updateCache(test, objectData, testResult);
15     end
16     updateAssessmentResults(results, testResult);
17     reduceCNFVulnList(vulnList, test, results);
18     updateCoverage(coverage, vulnList, test, results);
19 end
```

Algorithm 1: Probabilistic assessment algorithm

The proposed strategy is performed each time the Ovaldroid server considers that a security analysis needs to be made over a specific device. However, the event that potentially triggers such analysis is initiated by the client side. Indeed, a periodic *Hello* message is sent by the Ovaldroid client to the server in order to indicate its assessment availability as shown in Fig. 4. Communication messages are always sent by the client that analyses the response of the server. The responses of the server can be to start a new security analysis, to update the client policy and parameters, nothing to do at that moment (OK status) or an error such as busy error. If a new analysis is required based on the established frequency δ, the server will respond with the appropriate message and also the first OVAL object description to collect. The client will collect the items corresponding to the specified OVAL object and will send a new message to the server with the collected OVAL items. This mechanism is based on the piggybacking technique in order to reduce the amount of network messages transmitted during the process. The server will then respond with a new OVAL object request or a flag indicating the end of the assessment process. From the client point of view, it enters in a loop while the server keeps responding *ContinueAnalysis(OVAL object)* until it receives the assessment results. The collection of objects is quite simple and only uses two HTTP methods invoked from the client side. However, powerful network management protocols such as NETCONF [25] already exist and they could be envisioned in the future as soon as their linkage with OVAL and the SCAP protocol becomes more mature.

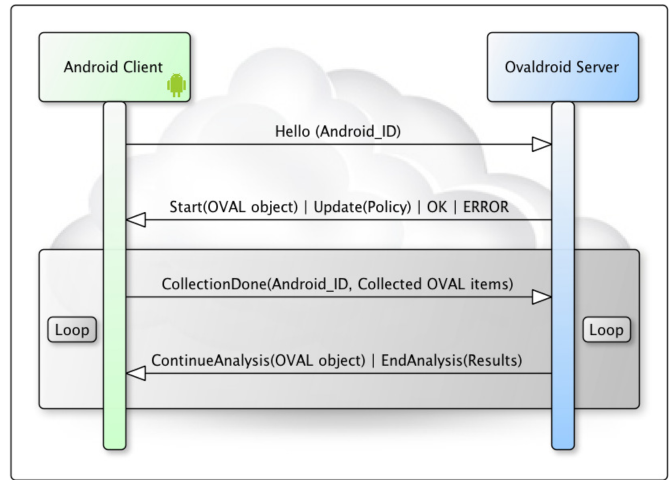

Fig. 4: Ovaldroid client-server interactions

V. Prototyping and performance evaluation

In order to provide a computable infrastructure to the proposed approach, we have developed an implementation prototype that integrates the building blocks presented in the Ovaldroid framework. We have also performed a deep behavioral analysis of the proposed framework through a comprehensive set of experiments. In this section we detail the implementation prototype, the experiments and the obtained results.

Ovaldroid has been designed using a client-server architecture. On the server side, a RESTful web service [26] enables mobile clients to communicate with the server and start new vulnerability evaluations. All the architectural components described in Fig. 3 have been purely implemented in Java 1.6 SE. Databases have been implemented using MySQL 5.1. OVAL-based vulnerabilities for Android are described using the OVAL Sandbox project [8] and they are translated to CNF representations using the CNF transformer provided by the *Aima* project [27]. XOvaldi, a multi-platform extensible OVAL analyzer, has been used as the OVAL interpreter [14]. On the client side, an extension to XOvaldi called XOvaldi4Android [7] conceived as a 94 KB size library has been used as the data collector subsystem. It is executed by the Ovaldroid client, implemented as a small Android service in charge of communicating the server according to its preconfigured frequency. The prototype has been developed to be compliant with Android versions starting at 2.3.3 thus supporting almost 80% of current operating versions.

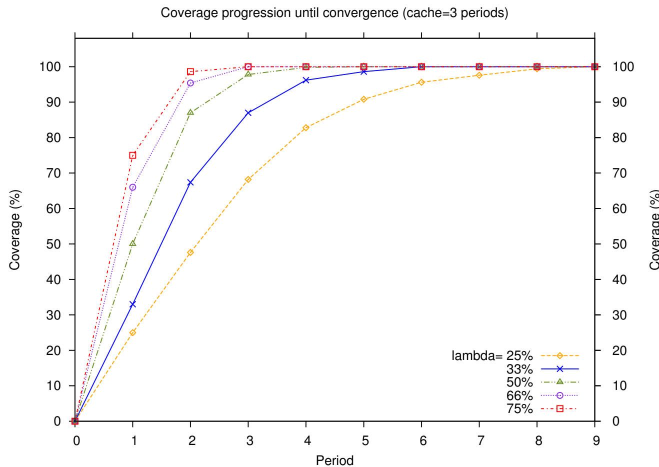

Fig. 5: Coverage convergence

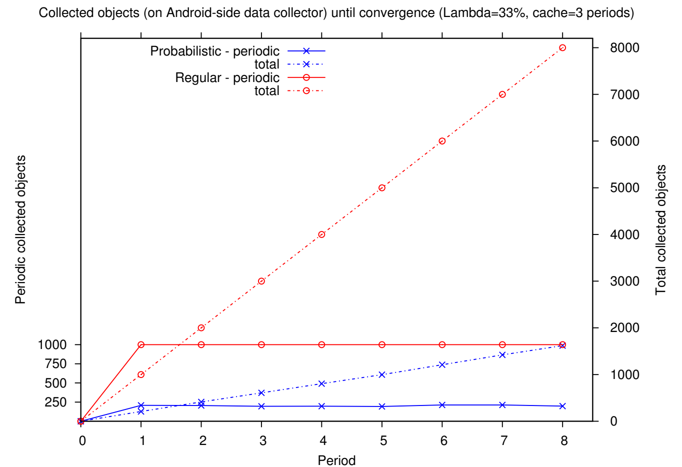

Fig. 6: Collected objects

In order to evaluate the behavior and performance of our framework, we have performed an extensive set of experiments using a regular laptop (Intel Core i7 2.20 Ghz, 8 GB of RAM, Linux kernel v.3.7.9) running the Ovaldroid server and a Samsung I9300 Galaxy S III smartphone (Quad-core 1.4 GHz, 4 GB of RAM, Android v.4.1.0) running the Ovaldroid client. The vulnerability database used within the experiments has been built taking real vulnerability descriptions for Android. In order to evaluate scalability aspects we have replicated their structure to construct more vulnerability descriptions involving two tests on average. Under a semantic perspective they represent the same vulnerability but under a technical perspective, these vulnerabilities and the involved tests, objects and states are different as they have different identifiers. Based on this methodology, we have constructed a database involving 500 vulnerability descriptions. Regarding Ovaldroid's parameters, we have experimented with several values for the vulnerability coverage λ while considering $\delta = 1$. As to the cache replacement policy, we have established an average of 3 periods before stored objects and results become expired.

We now present three different experiments that provide an insight of Ovaldroid's performance and show the feasibility of our solution. The first experiment shows how the proposed approach converges to a complete coverage of the vulnerability database across time. Indeed, one of the characteristics of Ovaldroid is the capability of distributing the load among different evaluation periods. By providing higher priority to those tests with higher utility as explained in Section III but simultaneously avoiding test starvation, the progression behaves as illustrated in Fig. 5. We can observe that only covering the 33% of vulnerabilities at each period (solid blue line with crossings), the whole vulnerability database can be 100% covered at the end of the sixth period. If the vulnerability database keeps the same, following periods will re-evaluate vulnerabilities according to their impact and importance, but always providing vulnerabilities with lower utility to be evaluated as well. If new vulnerabilities become available, they will have higher priority as they have never been evaluated. This re-evaluation process smooths the load impact on the target device, produces frequent and more accurate results, and also fits its potential changing nature. By augmenting the vulnerability coverage λ, our experiments have shown a faster convergence as expected.

In order to analyze the load activity variation on the Android side, we have performed a second experiment where we analyze the object collection behavior by measuring the standard approach evaluating all vulnerabilities at once, and Ovaldroid's approach distributing the assessment activity across time. Fig. 6 shows the observed behavior where two types of results are illustrated, namely, the number of collected objects per period (solid lines) and the total amount of collected objects (dashed lines). We can observe that while the standard approach collects 1000 objects per period (red solid line with circles), Ovaldroid's approach collects between 200 and 250 objects on average (blue solid line with crossings). This means that our approach only needs to collect approximately 25% of the objects required by the standard approach in this case, thus considerably reducing the load factor. Even though the proposed approach is slower than the standard one in terms of coverage speed, the load reduction achieved by Ovaldroid is really high and therefore, it positively contributes to the efficiency and responsiveness of the target device. The curves representing total accumulated objects show more clearly how the standard approach (dashed red line with circles) highly exceeds the interactions done by Ovaldroid with the mobile device (dashed blue line with crossings).

The experiments previously described consider a vulnerability dataset where each vulnerability has the same impact factor. In order to analyze the frequency with which each vulnerability is evaluated across time regarding their security impact, we have performed a third experiment depicted in Fig. 7 involving 14 evaluation time periods. Vulnerabilities identifiers are ordered by decreasing impact factor. We can observe as expected that vulnerabilities with a higher impact factor have been evaluated more frequently than those vulnerabilities with a lower impact factor. However, vulnerabilities with a lower impact factor have been analyzed several times meaning that the model also solves the starvation problem.

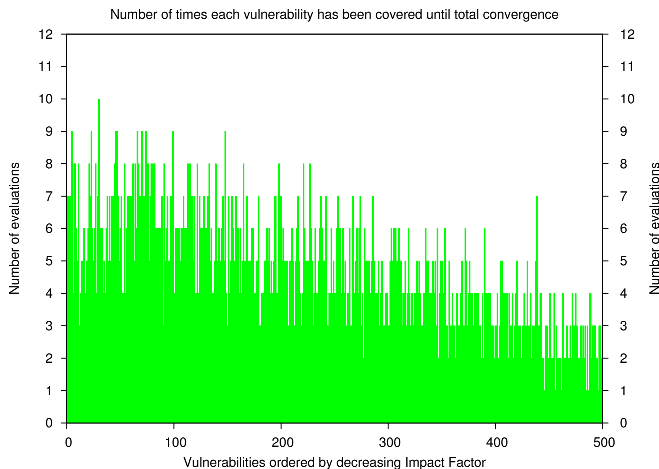

Fig. 7: Vulnerability evaluation rate

VI. CONCLUSIONS AND FUTURE WORK

Mobile devices, ubiquitous technologies and the services provided by them are revolutionizing the way we use and benefit from computing. However, end-users taking advantage of these unprecedented mobile technologies also face security problems that must be imperatively addressed. Vulnerabilities are a reality; they are present in applications, services and operating systems. In addition, current mobile devices still have limited resources that must be carefully managed in order to maximize the benefit obtained from them. In that context, we have proposed in this paper a novel approach for accurately detecting vulnerabilities on the Android platform and simultaneously outsourcing assessment activities thus minimizing the resource allocation required for this task. We have presented a statistical-based methodology for optimizing assessment activities and a probabilistic schema for ensuring complete and accurate vulnerability evaluations across time. We have also proposed a parametrizable OVAL-based assessment framework that highly reduces the resource consumption on mobile devices. Finally, we have presented an implementation prototype as well as a comprehensive set of experiments that shows the feasibility and benefits of our solution for performing vulnerability detection activities while keeping low load rates on mobile devices.

For future work we plan to analyze complementary algorithms in order to further optimize the proposed solution. Modeling a lookahead mechanism by effectively projecting which tests and OVAL objects would be required in subsequent evaluation steps would allow a single network request to carry more useful data at once thus speeding up the overall assessment protocol. Being a cloud oriented system, the Ovaldroid framework moves on a hostile environment. It must be able to defend itself against potential denial of service attacks and also to secure communication channels, e.g. using NETCONF over SSH. Finally, we consider that vulnerability awareness constitutes the first step towards more secure mobile solutions. However, remediating these vulnerabilities is a hard challenge that has been scheduled for future work as well.

ACKNOWLEDGEMENTS

This work was partially supported by the EU FP7 Univerself Project, FI-WARE PPP and the Flamingo Network of Excellence.

REFERENCES

[1] "Cisco Visual Networking Index." http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html. Last visited on August, 2013.
[2] "Android." http://www.android.com/. Last visited on August, 2013.
[3] "Gartner." http://www.gartner.com. Last visited on August, 2013.
[4] 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, USENIX Association, 2011.
[5] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," *Security Privacy, IEEE*, vol. 8, pp. 35–44, March-April 2010.
[6] "Lookout Mobile Security." https://www.mylookout.com/mobile-threat-report. Last visited on August, 2013.
[7] M. Barrère, G. Hurel, R. Badonnel, and O. Festor, "Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework," *In Proceedings of the 5th IEEE Symposium on Configuration Analytics and Automation (SafeConfig'12)*, Oct. 2012.
[8] "The OVAL Language." http://oval.mitre.org/. Last visited on August, 2013.
[9] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security Privacy, IEEE*, vol. 7, pp. 50–57, January-February 2009.
[10] P. Foreman, *Vulnerability Management*. Taylor & Francis Group, 2010.
[11] "VulnXML." http://www.oasis-open.org/committees/download.php/7145/AVDL%20Specification%20V1.pdf. Last visited on August, 2013.
[12] "MITRE Corporation." http://www.mitre.org/. Last visited on August, 2013.
[13] "CVE, Common Vulnerabilities and Exposures." http://cve.mitre.org/. Last visited on August, 2013.
[14] 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)*, pp. 32 –35, July 2011.
[15] "NIST, National Institute of Standards and Technology." http://www.nist.gov/. Last visited on August, 2013.
[16] J. Banghart and C. Johnson, "The Technical Specification for the Security Content Automation Protocol (SCAP)," *NIST*, 2009.
[17] N. Ziring and S. D. Quinn, "Specification for the Extensible Configuration Checklist Description Format (XCCDF)," *NIST*, March 2012.
[18] "CVSS, Common Vulnerability Scoring System." http://www.first.org/cvss/. Last visited on August, 2013.
[19] 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)*, pp. 722–725, Apr. 2008.
[20] 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)*, Oct. 2011.
[21] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A Logic-based Network Security Analyzer," *on USENIX Security*, 2005.
[22] Y. Diao, A. Keller, S. Parekh, and V. V. Marinov, "Predicting Labor Cost through IT Management Complexity Metrics," *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, pp. 274–283, May 2007.
[23] M. Chiarini and A. Couch, "Dynamic Dependencies and Performance Improvement," in *Proceedings of the 22nd Conference on Large Installation System Administration Conference*, pp. 9–21, USENIX, 2008.
[24] T. Xing, D. Huang, S. Ata, and D. Medhi, "MobiCloud: A Geo-distributed Mobile Cloud Computing Platform," in *Proceedings of the 8th IEEE International Conference on Network and Service Management (CNSM'12)*, pp. 164–168, IEEE, October 2012.
[25] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "RFC 6241, Network Configuration Protocol (NETCONF)," June 2011.
[26] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures, PhD. Dissertation, 2000." http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Last visited on August, 2013.
[27] "CNF Transformer." https://code.google.com/p/aima-java/. Last visited on August, 2013.