A Framework for Probabilistic Contract Compliance

G. Dini, F. Martinelli, A. Saracino, D. Sgandurra

IIT TR-03/2013

Technical report

Febbraio 2013

Istituto di Informatica e Telematica
A Framework for Probabilistic Contract Compliance

Gianluca Dini*, Fabio Martinelli†, Andrea Saracino*†, Daniele Sgandurra†
* Dipartimento di Ingegneria dell’Informazione
Università di Pisa, Italy
name.surname@iet.unipi.it
† Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
name.surname@iit.cnr.it

Abstract—We propose PICARD (Probabilistic Contract on AndRoID), a framework to detect repackaged applications for Android smartphones based upon probabilistic contract matching. A contract describes the sequences of actions that an application is allowed to perform at run-time, i.e., its legal behavior. In PICARD, contracts are generated from the set of traces that represent the usage profile of the application. Both the contract and the application’s run-time behavior are represented through clustered probabilistic automata. At run-time, a monitoring system verifies the compliance of the application trace with the contract. This approach is useful in detecting repackaged applications, whose behavior is strongly similar to the original application but it differs only from small paths in the traces.

In this paper, we discuss the framework of PICARD for describing and generating contracts through probabilistic automata and introduce the notion of ActionNode, a cluster of related system calls. Then, we present a first set of results using a prototype implementation of PICARD for Android smartphones to prove the efficacy of the framework in detecting two classes of applications, repackaged and trojanized ones.

Keywords—Probabilistic contract; Android; Malware; Repackaging;

I. INTRODUCTION

Mobile devices, such as smartphones and tablets, are becoming day-by-day more pervasive. Current operative systems for mobile devices are based upon the concept of apps. Apps are lightweight applications that are distributed through on-line marketplaces, such as the Apple AppStore or Android Google Play. Using this paradigm, users browse apps on markets and install them directly on their devices. This app-based model has been ported also on desktops and laptops, e.g. on the new Microsoft operative system Windows 8. However, this model is affected by some security and trust issues. As an example, some applications are produced by third-party developers, which can be malicious ones, and some markets do not ensure the security and quality of the published applications.

A large number of Trojan-like malware has been found hidden in applications distributed in the Android markets [1]. These applications are also known as repackaged applications, because they look and work as genuine applications, but they hide inside new code that misbehaves in the background, e.g. by contacting fake advertisement sites. A similar problem comes from Trojanized applications, a special class of repackaged applications that have malware inserted into them: nowadays, Trojanized applications are more than 80% of the total malware [2]. The detection of repackaged applications can be hard, since the observable behavior, from the user-side, of a good application is mostly the same as of a repackaged version.

In this paper, we propose PICARD (Probabilistic Contract on AndRoID), a framework aimed at detecting repackaged applications based upon probabilistic contracts. Contracts describe the sequences of actions that an application is allowed to perform when running. At run-time, the PICARD monitoring system verifies the compliance of the application trace with its contract using probabilistic contract matching.

The main contributions of this paper are:

- PICARD, a system that defines a contract of Android applications through probabilistic automata, which are computed from the application system call traces;
- the definition of ActionNode as a cluster of related system calls, represented through an oriented graph;
- the definition of probabilistic contracts and a method to verify the application compliance at run-time with the contract or to detect repackaged/trojanized applications;
- a first prototype of PICARD and a set of experiments on real repackaged and trojanized applications to prove the effectiveness of PICARD.

The remainder of the paper is organized as follows. In Sect. II we give some background information about application contracts and their generation. Section III discusses PICARD and the method to describe applications’ behavior through probabilistic automata and present the concept of ActionNode. Section IV describes the contract matching technique that is used to detect misbehaviors. In Sect. V we report some experiments to prove the effectiveness of our approach. Section VI reports some related works on system calls monitoring and probabilistic automata. Finally, Sect. VII concludes and proposes some future extensions.
II. CONTRACT

In this section, we give some background notions on contracts and application certification.

A contract fully describes the expected behavior of an application. A contract can be defined using information that can be computed either statically or dynamically. In the first case, the contract can be built by learning some properties from the source code or from an intermediate-level language (e.g., byte-code) or directly from the binary code. With this approach, it may be impossible to know in advance some properties of the code, e.g. behavior of the application that depends about some inputs, such as configuration files read at run-time. Usually, with this approach, when matching the contract with the application run-time behavior, there are no false positives (since the static analysis over-approximates the set of legal actions), but more false negatives may arise.

Alternatively, a contract can be defined by exploiting the information learnt from the application’s executions, e.g. by monitoring some executions of the program to extract its behavior. In this scenario, to faithfully represent the application behavior, two conditions must be met: (i) all the possible actions should be generated so they can be monitored, otherwise false positives may arise at run-time; (ii) the device is not attacked during this phase, otherwise false negatives may arise at run-time.

When matching the application run-time behavior with the contract, the advantages of the dynamic approach are, usually, a reduction of the false negative occurrences. In fact, a contract that is defined using information dynamically generated should better represent the real executions of the application. In this case, a contract clusters the possible actions of an application into two sets: “likely actions” and “unlikely actions”. In this scenario, an unlikely action that is performed several times should be considered a misbehavior. The disadvantages of the dynamic generation is the higher number of false positives, since some good actions may not be included in the contract if, during the learning phase, some good actions have not been performed.

A contract can be formally certified, i.e. it is coupled with a proof that certifies that the contract matches the application behavior (e.g., proof-carrying-code). Otherwise, a contract can be digitally signed (without a formal proof) and, in this case, the only form of guarantee is the trustworthiness of the signer. A contract can be self-generated, i.e. there is no proof that it matches the application behavior. Finally, a contract can be missing: in fact, in some case, a contract is not necessary, e.g. when a policy is enforced regardless of the contract which can be used only to save computational time on the final node and the final node checks if the contract matches the policy.

The contract can be generated by (i) the developers; (ii) the distributor (e.g., marketplace); (iii) third-party certification authority; (iv) by the user. If a contract is missing, it maybe useful to generate the behavior from the code to see if it matches the policy before running it. Any combinations of the previous points is possible and, in any case, the contract can be generated either statically or dynamically.

Let us consider \( A \) the set of possible actions for an application \( \alpha \) and \( C \) the set of actions described in a contract \( \gamma \) for \( \alpha \). Considering the relationship between \( A \) and \( C \), the contract \( \gamma \) can be: (i) redundant: \( A \subset C \), the contract \( \gamma \) over-approximates the set of legal actions of \( \alpha \); when matching the run-time behavior with the contract, no false positives are generated but false negatives may arise. As previously said, this is usually the kind of contracts that are generated statically, since a static analysis (e.g., data-flow analysis) always returns safe assumptions; (ii) under-specified: the contract under-approximates the set of legal action (false positives may be generated at run-time and also false negatives, but usually less than in the first case). This is usually the kind of contract that are dynamically generated, since not all the legal behaviors may have been generated; (iii) exact: this would theoretically perfectly match the application’s behavior.

If a formal proof of the contract-application matching application is missing, then the compliance of an application with its contract has to be verified. An application \( \alpha \) is compliant with the contract \( \gamma \) when all the actions of the application are included in the contract, i.e.:

\[
\alpha \models \gamma
\]

We say that an application \( \alpha \) is compliant with a probability \( \xi \) to the contract \( \gamma \) if any action performed by \( \alpha \) is in the contract and happens with a probability greater or equal to \( \theta = 1 - \xi \), i.e.:

\[
\alpha \models_\xi \gamma
\]

The definition of probabilistic compliance suits better to the dynamic generation of contracts. In fact, contracts defined using dynamically-generated behaviors may be probabilistic as well and, since they are built upon execution traces, it is possible to compute the probability that each action is performed, including a quantitative information in the contract. Probabilistic contracts are easily described with Markov Chain models [3].

III. PICARD: A FRAMEWORK FOR PROBABILISTIC CONTRACT ON ANDROID

In this section, we describe PICARD (Probabilistic Contract on AndRoID), which is a framework that exploited the dynamic approach for contract generation discussed in the previous Section to define a contract through probabilistic automata.

In the following, we consider that any application trace is a sequence of system calls and that multiple traces can then be combined together.
A. Contract Generation

During its lifetime, a process issues several system calls, which form a trace, i.e., an ordered sequence of system calls. If we consider each system call as a node of an oriented graph, whose edges represent the transition from a system call to the next one in the original trace, then it is possible to provide a signature of the application behavior by transforming the original trace in an oriented graph. Hence, since during a trace execution a system call can be issued several times consecutively, the graph also includes ears, which are edges outgoing and ingoing into the same node. Moreover, each edge is labeled with a sequence number that represents the position in the trace of a transition, allowing a complete description of the execution trace itself. This graph type is known in graph theory as multi-graph. PICARD revises this well-known representation for application behavior by introducing the concept of node clustering and applying elements of probabilistic contract theory.

We introduce the notion of ActionNode, which is a cluster of related system calls, i.e., the graph of system calls that are consecutively issued in the trace and that are bound by some relation and that form an action (a high-level operation). In general, any relation can be used, e.g., any partition of the set of system calls in several subsets. In our scenario, the relation should produce a meaningful action: as an example, an ActionNode can be composed by the multi-graph of the system calls performed consecutively on the same file, where the relation is the fact that all these system call work on the same file descriptor to produce a relevant action.

As an example of ActionNode, consider the sequence of system calls: open(A) – read(A) – close(A), where A is the filename. This ActionNode represents at high level the action of reading data from file A: this action requires that, firstly, the file has to be opened, then data is read and, finally, the file is closed. Some examples of ActionNodes are depicted in Fig. 1.

Several advantages stem for this representation: if we transform the original execution graph using ActionNodes, then an application trace can be seen as a graph whose nodes are actions, and this representation is more meaningful and expressive than a trace whose nodes are just system calls. In fact, generally a program executes several system calls that, taken as standalone in the trace, only give limited information about the application behavior. This is due to the fact that, usually, there hundreds of system calls on a mobile OS, but generally a program executes only few of them, repeatedly. If we represent a trace with a graph where each node is a different system call, then the program behavior is represented through a graph with few nodes and a lot of edges that, usually, form a full mesh. Using ActionNodes, the framework can define a larger number of nodes, hence more detailed signatures.

When defining contracts, if we focus on n system calls (the critical ones), the maximum theoretical number of ActionNodes that can be achieved is:

$$max_{bn} = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} k^2$$

An ActionNode can contain at most n distinct nodes and the number of possible ActionNodes is given by the sum of possible dispositions, with an increasing number k of nodes. Moreover, each node may have a number of outgoing arcs that may vary from 0 to k and, hence, each number of dispositions should be multiplied for k^2. Notice that max_{bn} is a theoretical upper bound, which should never be reached due to the semantics of the action. Although it is strongly unlikely that a program performs a close system call on a file and then start to read it with the read system call. Hence, several ActionNodes should not exist actually. As we describe in Sect. V, the number of ActionNodes generated in a normal execution trace is much lower than max_{bn}. Hence, just the presence, or the absence, of one or more ActionNodes in an application trace constitutes a signature of the application.

The nodes that compose an ActionNode, i.e., system calls, will be called henceforth SysCallNodes. In the following, we consider as SysCallNodes system calls that act on files, namely the open, read, write, close, ioctl, system calls.

B. Algorithm to Define the Probabilistic Contract

To define the contract, the PICARD’s algorithm takes as input all the generated application traces and outputs a probabilistic automaton. When traversing the aggregated traces, from the first to the last system call, the algorithm checks, for each node, if the argument is the same of the previous one. Since the monitored system calls act on files, then the argument is the filename (or file descriptor). Then, a new ActionNode is created each time a system call is issued.
with an argument that differs from the previous system call (see Fig. 2).

Different and subsequent system calls with the same argument are inserted in the same ActionNode (Figure 3). After an ActionNode is generated, an edge is added from the previous ActionNode to the current one. Hence, the system call traces are represented by a multi-graph of ActionNodes. This multi-graph contains an ActionNode for each high-level operation. The ActionNodes are oblivious of the file-name, meaning that the same cluster of operations, performed on two different files, generate the same ActionNode. The multi-graph may contain also several edges insisting on the same node, if an ActionNode is visited more than once.

Finally, from the multi-graph, an automaton is built that contains the same nodes of the multi-graph, but the edges ending on the same node are collapsed into a single edge. Each edge is labelled with the probability that it is traversed by taking into account the number of occurrences of each edge. More specifically, each edge \((i,j)\) reports the probability of reaching directly (in a single step) the node \(j\) from the node \(i\). It is worth noticing that this automaton can be described by means of a Markov-Chain, whose states are the ActionNodes of the application, and the transition matrix reports the probabilities of the automaton edges. This probabilistic automaton is used to define the probabilistic contract of the application behavior. It should be noted that, in a real-world scenario, the contract should be defined by the developers of the applications, so that all the possible states and transitions are explored.

IV. CONTRACT COMPLIANCE

Once an application contract has been defined, through the algorithm described in the previous Section, PICARD exploits the contract to verify the compliance of the application at run-time, e.g. to verify that an application has not been maliciously repackaged. Repackaged applications are apps that have been modified by developers adding to the original code some code that performs malicious actions. The repackaged application is then distributed as it was the original one. The rationale behind the PICARD approach is that these misbehaviors, even if small, should be actions, or sequences of actions, that are not part of the contract. Hence, the misbehavior should be recognized and eventually stopped before it can harm the user or the device.

At run-time, PICARD incrementally builds the monitored application behavior: a misbehavior is performed when an application is in a state of the contract’s automaton that is strongly unlikely.

Markov Chain is used to model stochastic processes with the following property:

\[
Pr\{X_{t+1} = i_{t+1} | X_{t} = i_{t}, X_{t-1} = i_{t-1}, \ldots, X_{0} = i_{0}\} = Pr\{X_{t+1} = i_{t+1} | X_{t} = i_{t}\}
\]

As explained in the former Section, probabilities are computed by counting the number of outgoing edges from a node, ignoring the sequence. Hence, it can be easily verified that for the PICARD contracts the Markov property holds. For any \(t \in T\), where \(T\) is the set of time steps, and for any \(i \in S\) where \(S\) is the set of possible states. Hence, Markov Chain are used to model stochastic processes where the probability of being in a state \(i\) only depends on the former one. The probability of transition from one state \(i\) to another state \(j\) is described by the numbers \(p_{i,j}\). These numbers form a matrix of size \(m \times m\), \(m = \#(S)\) called transition matrix \(P\).

In PICARD, the probabilistic contract is composed of ActionNodes where edges are labelled with transition probabilities. Hence, each ActionNode is a state in a Markov Chain transition matrix, and \(m\) is the number of ActionNodes in the contract. In a Discrete Markov Chain, the probability of being in each state \(i\) at the step \(n\) is computed as the \(i\)-th element of the vector:

\[
p^{(n)} = p^{(0)} p^n
\]

where \(p^{(0)}\) is the initial state vector, representing the probability that the system starts in a specific state.

PICARD defines three levels of contract violation. In order of severity we have:

- **Probability Misbehavior**: The probability of being in the current state is very low but non null. Given a threshold \(0 < \theta < 1\), if \(p^{(n)} < \theta\), then a misbehavior is detected. The value of \(\theta\) is parametric with respect to the length of the considered subtrace and
the application is performing several times an action that is strongly unlikely. For example, consider an application that has to send an SMS after the installation for registration purposes. In the probabilistic contract of such an application the “send SMS” operation should be considered strongly unlikely, since it happens only once. If a repackaged version of this app, starts to send several SMS, the framework should detect such a misbehavior.

- **Missing-Edge Misbehavior**: the probability of being in the current state \( i \) is zero, because the edge from \( j \) to \( i \) does not exist in the contract, where \( j \) is the state at the step \( n - 1 \). This is the case of a mimicry attack. The application performs normal high-level operations, not considered dangerous as standalone, but that may harm the device if performed in a specific malicious sequence. The malicious sequence should never appear in the contract and the misbehavior should be detected.

- **Missing-ActionNode Misbehavior**: the probability of being in the current state \( i \) is zero, because the current state (ActionNode) does non exist in the contract. This means that an unknown high-level operation has been performed. This is the strongest misbehavior. In the case of a repackaged application, this ActionNode constitutes a high-level operation such as sending an SMS, or accessing to a system file, which is never performed on the original version.

Anytime PICARD detects a contract violation, it is up to the system policy to decide which rule to apply, e.g. stopping the application, send an alert to the user, and so on.

V. Tests

To test the effectiveness of our approach, we have built a first prototype implementation of PICARD to (i) define the contracts from the application traces and (ii) check the compliance of the applications’ run-time behavior with respect to their contract. To test the efficacy of PICARD, we have analyzed two case studies: (i) proof-of-concept of repackaged applications, by updating all the sample applications provided with the Android SDK; (ii) real applications and their repackaged version, found on official market and on malware archive site.

In the following we focus on one example for each class.

**Repackaged Application**: One proof-of-concept application that we tested is a Tic Tac Toe sample application provided with the Android SDK. We modified the application code to perform a malicious unwanted behavior. The modified application sends an SMS message to a specific number each time the user opens the menu to change the game skin, slowly leaking the money credit. The SMS is sent stealthily, since Android does not advise the user when an SMS is sent by an application. Moreover, the messages sent by an application are not stored in the message outbox, hence there is no trace left of the occurred event, except for the leaked money. It is worth noticing that the unwanted SMS sending is a typical misbehavior of several malware that can be found in repackaged applications [4].

Firstly, PICARD records several application traces, which are modeled using a multi-graph representation, and then it generates the probabilistic contract of the original Tic Tac Toe game (i.e. the not malicious application). Then, PICARD explores the multi-graph representation to build the ActionNodes and to compute the probabilities of each transition among ActionNodes and, finally, it outputs the contract. To build a representative graph, the traces have been recorded trying to explore all of the game features. Then, PICARD recorded further game traces of both the original game and of the malicious one. In each trace the graphic skin was changed at least once to trigger the SMS sending in the malicious application.

In Figure 4a we report the PICARD probabilistic automaton extracted from the traces of the original application. This automaton constitutes the probabilistic contract that has to be compared to other traces, whose compliance has to be proved. Each node represents a specific file high-level operation (or action), which is a sequence of related system calls on the same file. Each edge is labelled with the probability of transition from the previous ActionNode to the next one. Since the Tic Tac Toe game is a very simple application, the number of states is relatively small. The automaton in Figure 4b represents the execution of the malicious version of the game. We set the value of the threshold \( \theta \) has a function of the length of the considered subtrace and the minimum of the probabilities of the edges from the first node of the current subtrace to the current node.

At run-time, PICARD incrementally builds the execution graph of the monitored applications through ActionNodes, any time an action is performed. As explained in Sect. IV, when checking the run-time behavior of an application, the case of a ActionNode that is missing in the contract is the most suspicious one. A missing ActionNode represents, in fact, a new high-level operation performed by the monitored application that is not included in the contract. The automaton of the malicious trace of the Tic Tac Toe game is much more complex than the one of the original version. ActionNodes of the original version are composed of no more than two SysCallNodes, due to the simplicity of the application. Instead, in the malicious trace, several new high-level operations have been detected. An example of missing ActionNode in the contract is number 11 of Figure 4b, which is shown in Fig. 5. In this case, the non-compliance of the trace with the contract is verified since this operation is not included in the contract. We would like to point out that, by using a representation without ActionNodes, then this difference between the two traces may have gone been undetected. In fact, both the original and the malicious
trace use all of the monitored system calls, but only the malicious one use them to perform malicious actions that are easily identified through the ActionNode representation. Furthermore, several times a probability misbehavior has been detected as well as missing-edge misbehaviors.

On the other hand, all the traces of the original application have been reported as compliant with the contract, i.e. no false positives have been raised.

Figure 5: ActionNode 11 in the malicious trace.

Trojanized Applications: The effectiveness of PICARD in detecting repackaged applications has been proved on real applications downloaded from the official market. We report here the results on the game Baseball Superstars 2010, which is a complex baseball simulation game. We found a repackaged version of the same game, which has an observable behavior identical to the one of the original application. However, the repackaged version is infected by a Trojan malware called Geinimi. The malware steals private user data, like the IMEI code, and tries to send them to an attacker via SMS or Internet.

To build a representative probabilistic contract, several traces have been collected by PICARD while playing the game for several hours, exploring the various play modes. The probabilistic automaton extracted from these executions graph has 13 states, hence, the type of operations concerning the monitored system calls is comparable with the one of the simpler Tic Tac Toe game. However, the number of edges is much higher and the probabilities on each edge have a higher variance than in the Tic Tac Toe automaton. This correctly depicts the higher complexity of the real application with respect to the proof of concept formerly analyzed. Detecting non-compliant behavior may result more difficult with the increase of the application complexity. However, PICARD proved to be effective also in this case. PICARD has monitored several traces of the genuine application and several traces of the repackaged version.

The traces of the repackaged application resulted in the generation of four ActionNodes that are not contained in the contract. These misbehaviors are related to the various attempt to access the IMEI code of the phone. The IMEI code can be exploited by an attacker to clone the USIM of the device. Due to the insurgence of this unknown, malicious operations, the application is not considered compliant with the contract and is recognized as repackaged. As in the previous experiment, several probability misbehaviors and missing-edge misbehaviors have been detected.
To check the specificity of PICARD, some traces of the original application have been extracted. None of these traces have been used to build the contract. PICARD classified all of these traces as compliant with the contract. In all the experiments, no false positives have been raised. Finally, the run-time performance overhead of PICARD is negligible.

VI. RELATED WORK

[5] proposes Crowdroid, an IDS that is based on the number of system calls issued by an application. Misbehaviors are identified by applying computational intelligence techniques. However, such a system may be deceived by low-profile attacks and attacks brought by cooperating malicious apps. Another system that exploits system calls and computational intelligence is presented in [6], which is an anomaly-based intrusion detection system that, differently from Crowdroid, monitors the system globally, but it may not be able to detect some trojanized application if their behavior faithfully represents the good ones.

A further system that exploits machine learning to detect intrusion on Android is Andromaly. Differently from the previous works that mainly focus on low-level events, e.g., system calls, Andromaly considers the occurrences of higher level events and use them to detect intrusion but has been tested only on proof-of-concept malware. Some Android security frameworks try to protect the system by monitoring the communication level and defining security policies. One of these systems is presented in [7], which allows the definition of context based security policies. Analysis of system calls with Markov Models have formerly been performed on other operating systems. In [8] a scheme for intrusion detection is proposed. This system exploits system calls and hidden Markov models and is able to detect efficiently denial of service attacks. [9] presents another system based upon system calls and Markov models to detect intrusions. This system analyzes the arguments of the system calls but is oblivious of the system call sequence. System call sequence and deterministic automata have been used in [10] to detect anomalies, which are detected when system call sequences differ from an execution trace known to be good.

VII. CONCLUSIONS

Repackaged applications are one of the main security issues for mobile devices. In this paper we have presented PICARD, a framework to detect repackaged applications on Android systems through probabilistic contracts. The contract is represented using a clustered multi-graph, where each node represents a specific high-level operation, which is converted into a probabilistic automaton. At run-time, the application behavior is checked against the contract and a misbehavior is detected when the application is in a state of the automaton that has a low or null probability according to the probabilistic contract. Using a probabilistic approach, the effectiveness in detecting misbehavior is increased. We have proved the effectiveness of PICARD by testing the framework both on repackaged and Trojanized applications.

We are planning to extend this work by increasing the number of monitored system calls and extending the set of tested applications and using new relation among system calls to create ActionNodes based upon a different notion of relation rather than file descriptor. A further extension is the extraction of behavioral patterns from the applications, which can be used to perform black-list-based intrusion detection of malicious applications.

REFERENCES


