CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Detection of Repackaged Mobile Applications through a Collaborative Approach

Alessandro Aldini[1], Fabio Martinelli[2], Andrea Saracino[2], Daniele Sgandurra[3]

[1]*Dipartimento di Scienze di Base e Fondamenti, Università di Urbino, Italy*
[2]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*
[3]*Department of Computing, Imperial College London, London, UK*

SUMMARY

Repackaged applications are based on genuine applications, but they subtlety include some modifications. In particular, trojanized applications are one of the most dangerous threats for smartphones. Malware code may be hidden inside applications to access private data, or to leak user credit. In this paper, we propose a contract-based approach to detect such repackaged applications, where a contract specifies the set of legal actions that can be performed by an application. Current methods to generate contracts lack information from real usage scenarios, thus being inaccurate and too coarse-grained. This may result either in generating too many false positives or in missing misbehaviors when verifying the compliance between the application and the contract. In the proposed framework, application contracts are generated dynamically by a central server merging execution traces collected and shared continuously by collaborative users executing the application. More precisely, quantitative information extracted from execution traces is used to define a contract describing the expected application behavior, which is deployed to the cooperating users. Then, every user can use the received contract to check whether the related application is either genuine or repackaged. Such a verification is based on an enforcement mechanism that monitors the application execution at run-time and compares it against the contract through statistical tests. Copyright © 2014 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: probabilistic contract; Android; malware; app repackaging

1. INTRODUCTION

The large diffusion of smartphones and tablets experienced in the last years has drastically changed the paradigm of application distribution. Currently, applications for mobile devices (apps, for short) are distributed through online marketplaces, such as *Google Play* or *App Store*. These marketplaces act as an hub where app developers publish their own products, which can be bought or downloaded for free by users. Usually, official market charge users for these apps, while several unofficial marketplaces distribute their own apps free of charge. In particular, with unofficial markets users can simply download the app file and install it later without adhering to the official installation steps. In this last case, trust is at risk, since there is no centralized control and it may happen that untrusted developers distribute malicious apps. This issue is particularly serious for Android – which represents the framework of reference for our study – as it is both the most popular operating system for mobile devices and the system with the greatest share of malware in the last years [1]. In particular, Android accounts for 97% of all mobile malware in 2013. The number of new malware is pretty scary: on average, every day more than 160,000 new specimens are reported [2]. Moreover, recently, researchers [3] have found that a quarter of all Google Play free apps are clones, i.e., repackaged apps of popular apps such as WhatsApp and Angry Birds.

The vast majority of mobile malware is Trojan, which comes in the form of *trojanized app* [4]. These apps look like genuine ones, but they run malicious code in the background. Typical misbehaviors of these apps include private data leakage, user position tracking, stealthy outgoing of SMS messages and forced subscription to premium services. Trojan related to SMS and subscription to premium services are extremely common and constitute a Trojan class named *SMS Trojan*, which accounts for almost 83% of the total Trojan malware. These malicious apps send SMS messages, which by itself is a cost for the user, who will not even notice this misbehavior, since in Android devices SMS sent by apps are not stored in the outbox [5]. Moreover, SMS trojans often send SMS messages used to subscribe the user to premium services. These services charge the user periodically for services, generally multimedia contents, sent via SMS or MMS. However, SMS trojans intercept the action of incoming messages and if they are from a white-list of premium numbers, then they are dropped without notifying the user. Only after a substantial amount of money has been leaked in this way, it is likely that the user notices the misbehavior.

Diffusion of malicious apps is a serious issue. In general, it is well-known that they can be easily distributed through unofficial markets, which are known to be non-secure. However, malicious apps have been also found in the Google Play store. Furthermore, extremely popular apps have been distributed also through uncontrolled channels, like simple Web pages, which thus represent the best target for developers of malicious apps. An example is given by the case of the `Flappy Bird` app. This extremely widespread app has been recently removed from the official market for reasons that are not related to security, but it is still possible to get it through other channels. Currently a considerable share of the `Flappy Bird` app available online are trojanized [6].

Since it is hard for a user to detect misbehaviors due to hidden malicious code, it is worth adopting ad-hoc, automatic, control mechanisms. Android includes security mechanisms to protect the device and its user from malicious apps. In particular, the *Android permission* system provides an access control mechanism for all the resources and critical operations on the device. However, the effectiveness of this system is limited against trojanized apps. The main reason is that permissions are too coarse-grained to express effective security policies [7], while users may find hard to understand the security threat brought by apps by simply reading its permission list [8].

In this paper, which is a revised and extended version of a work presented at CTS 2013 [9], we present PICARD (ProbabIlistic Contracts on AndRoiD), a probabilistic contract-based intrusion detection system to recognize and block the misbehaviors performed by trojanized apps on Android devices. PICARD is a collaborative framework based on probabilistic contracts generated from the execution traces collected by a network of collaborative users. A *contract* is a document that describes the expected behavior of an app. A version α of an app executed by the user is *compliant* with the contract γ related to the app, denoted $\alpha \models \gamma$, when all the sequences of actions effectively performed by the app are included in the contract.

A contract can be defined using information that can be computed either statically or dynamically. In the static approach, the contract can be built by learning some properties from, e.g., the source code. However, it may be difficult to know in advance some properties of the code, such as the behavior of the app depending on specific inputs or interactions. Moreover, static approaches are based on the availability of the app source code, which is seldom distributed by the app developers. In the dynamic approach, a contract can be defined by exploiting the information learned from app's execution, which is monitored at run-time to extract its behavior. Our methodology focuses on the dynamic approach, as it is more suitable to represent apps whose behavior depends on user inputs. Contracts defined by using dynamically-generated execution traces can also include quantitative information deriving from the direct observation of the app behavior. In particular, such information can be obtained by analyzing the occurrences of any execution trace. As we will see, by enriching the contract with specific information about the frequency with which every behavior is expected to be observed, it is possible to approximate probabilistically the relation $\alpha \models \gamma$. More specifically, since both γ and α include quantitative information, intuitively the idea is to verify whether α is approximately compliant with γ up to some tolerance threshold ξ, written $\alpha \models_\xi \gamma$.

Finally, it is worth observing that in the proposed approach the contract is based on execution traces provided by collaborative but possibly untrusted users. Hence, to keep track of the trustworthiness of each user participating in the contract generation, we integrate our framework with a centralized reputation system employed to favor user collaboration and exactness of the contract.

Contributions of the Paper The main contributions of the paper can be summarized as follows:

- We present PICARD as a collaborative framework in which users share, through a central server, execution traces of the apps running on their mobile devices. This is done through the PICARD app, which is the component running on the mobile device used to collect execution traces of the apps at the system call level and to protect the device itself from app misbehaviors.
- We introduce the concept of ActionNode to describe app behaviors and app contracts through clustered graphs of system calls. Then, based on such a notion, we present (i) an approach to the formal representation of app behaviors and contracts through probabilistic automata, and (ii) a method to collaboratively build a contract by merging several execution traces from different users, by taking into account their reputation.
- We describe a method to match the behavior of an app with a probabilistic contract through statistical tests, to discern between genuine and repackaged versions of a mobile application.

Structure of the Paper The rest of the paper is organized as follows. In Section 2, we describe the design of the PICARD framework, by detailing the algorithm used to acquire traces from collaborative users. In Section 3, we describe the methodology used to generate probabilistic contracts. In Section 4, we define the statistical methods used to match a monitored behavior with the contract. In Section 5, we show the viability of the approach through experiments on some trojanized apps. Section 6 reports on related work about contract-based approaches and probabilistic techniques to monitor the app behavior. Finally, Section 7 concludes the paper by discussing some future works.

2. PICARD FRAMEWORK DESCRIPTION

In this section, we describe in detail the framework of PICARD, which is a distributed, collaborative client-server platform (see Figure 1). In this framework, different users share dynamically their experience in the usage of every specific app – in the form of monitored execution traces – through a centralized server. The objective of the server is to employ and combine the behavior of the app, as monitored by every participating user, in order to build incrementally an app contract, which is then broadcast to all the involved users. Basically, no trust assumptions are made *a priori* on users. Hence, to enable a virtuous circle ensuring user collaboration and reliability of the collected information, user reputation is continuously updated and considered during the whole contract lifetime.

As far as the client side is concerned, we assume that each user has a unique identifier assigned by the PICARD server when the PICARD app is installed on the user device. The identifier is based on a fingerprint of user's phone IMEI (International Mobile Equipment Identifier) in order to ensure its uniqueness. Whenever the user downloads a new app, the PICARD app notifies the central server in order to obtain, if available, the app contract, which is signed by the private key (of an asymmetric cryptography key pair) of the server. Then, at the client side the PICARD app implements the algorithm depicted in Figure 2. Execution traces of the monitored app are collected directly and constantly on the user device. If the PICARD app has a contract for the app, then the monitored behavior is used to detect possible misbehaviors with respect to the contract, otherwise it is simply forwarded to the server. If a misbehavior is detected, then PICARD blocks the current action and notifies the user that the app behavior does not match the contract. The user can then choose to stop the misbehaving app, block and uninstall it, or simply do nothing. Notice that if no misbehavior is detected, then the monitored behavior is sent to the server to contribute to the contract maintenance.

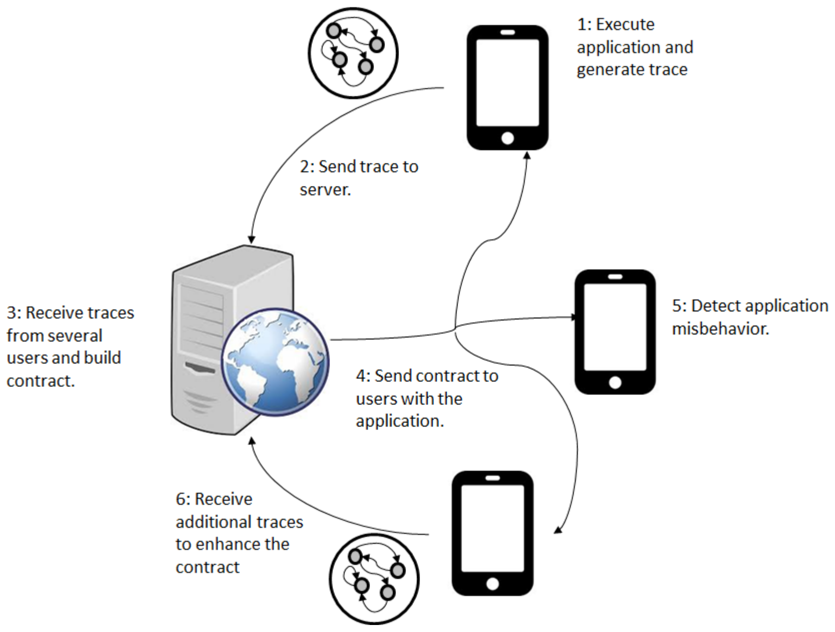

1: Execute application and generate trace
2: Send trace to server.
3: Receive traces from several users and build contract.
4: Send contract to users with the application.
5: Detect application misbehavior.
6: Receive additional traces to enhance the contract

Figure 1. PICARD High-Level Architecture

PICARD exploits system call analysis to capture and monitor the app behavior at the kernel level. The advantages of performing analysis at this level is that performing a code hijack at such a low level is harder than doing it at an higher level, e.g., API or application level. Hence, the obtained behavior representation should be exhaustive and comprehensive of every action performed by any app. The sequences of system calls resulting from the monitoring activity derive from the app usage during its lifetime. Typically, an execution trace is recorded from the time when the app is started and ends when the app is closed, or after a sufficiently long time of the app execution. When PICARD stops monitoring an app, i.e., the monitored app is terminated or the trace length overcomes a specific threshold, the user is asked to send the execution trace as a report to the PICARD central server. To reduce the required level of interaction, which can be considered annoying by users, it is possible to set the app to automatically send the execution traces when they are available.

On the PICARD server side, a database of users, apps, and contracts is managed. All the operations dedicated to user reputation and contract management are handled by the central server. When a user sends the report concerning an observed execution trace, the PICARD server verifies if the related app is already in the database. If the app is missing, a new record is added. Then, the PICARD server follows the algorithm depicted in Figure 3 to manage the app contract on the basis of the received execution traces. As can be noticed, the first operation upon trace reception consists of checking contract's completeness, which is an index describing the reliability of the contract. Intuitively, a contract is not considered complete (and not made available to the users) until enough traces have been received from a sufficiently high number of users. As we will see in Section 3, the completeness level is determined through statistical methods contributing to validate the contract.

If the contract is not complete yet and if the reputation of the user exceeds a given threshold θ, i.e., the feedback reported by the user is trusted enough, then the execution trace and the reputation of the user are used to update the partial contract of the app, as detailed in Section 3, otherwise the trace is discarded and does not contribute to the contract. Merging the contributions provided by several different users is fundamental to achieve contract completeness. In fact, if we only consider the traces recorded by a single user (or a small group of users), then the generated contract would be partial since a user rarely explores all of the possible app behaviors.

As soon as the contract is complete, the fitness of any execution trace received is tested against the contract. If the test is passed, then the reputation of the user is increased to reward the collaborative behavior. Moreover, notice that the contract, even if complete, is continuously updated to keep track

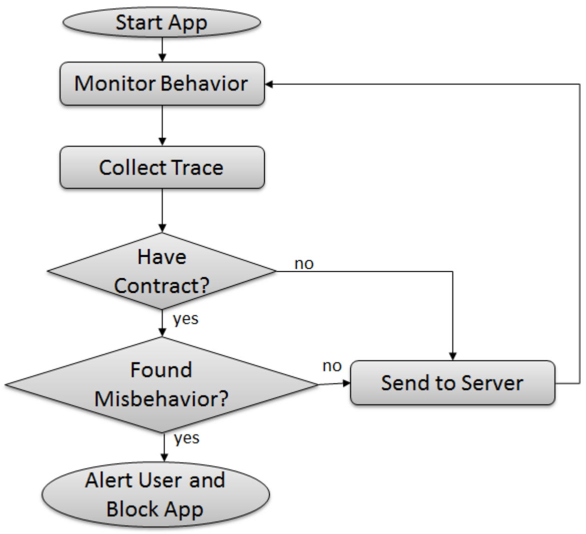


Figure 2. Description of the Behavior of the PICARD App

of additional contributions provided by new traces. If the test is not passed, then it may reveal a potential malicious behavior of the user, whose reputation is decreased. Then, the trace is recorded (and added to the contract) only if the user is trusted enough. In other words, the behavior of users reporting their usage experience is rewarded (or punished) in terms of reputation, provided that such a feedback is (or is not) consistent with respect to the contract. In fact, it is likely that cheating users provide false feedback that, however, does not fit the contract behavior. Hence, the fitness test and the negative reputation variation related to unsuccessful tests represent a way for isolating such behaviors.

The complete version of the contract is sent by the PICARD server to each involved user, while new versions of the contract – resulting from the combination of additional execution traces continuously sent by collaborative users – are released when necessary. These operations are clustered and the updated contract is resent to users periodically. To strengthen the role of reputation as an incentive to promote collaboration, the server may decide to privilege trustworthy users by sending to them the updated versions of the contract more frequently with respect to less trusted users.

Finally, we point out that the dynamic nature of the approach used to define the contract is such that even a complete contract, which is essentially generated through tests, cannot be completely specified in a formal sense. A fully specified contract requires full state space exploration, which, however, could be impractical in real scenarios.

3. CONTRACT GENERATION

In the PICARD framework, generation of an app contract is based on the quantitative analysis of a certain amount of different execution traces that represent the usage of the app by several users. Each execution trace observed by a trusted user is transformed into a clustered execution graph, in which system calls are grouped to form graph nodes and each edge between nodes represents the transition from a node to the next one. From a collection of these graphs, we derive a probabilistic contract that describes quantitatively the expected behavior of the app. In the following, we present step-by-step the generation of a probabilistic contract starting from the acquisition of the execution traces.

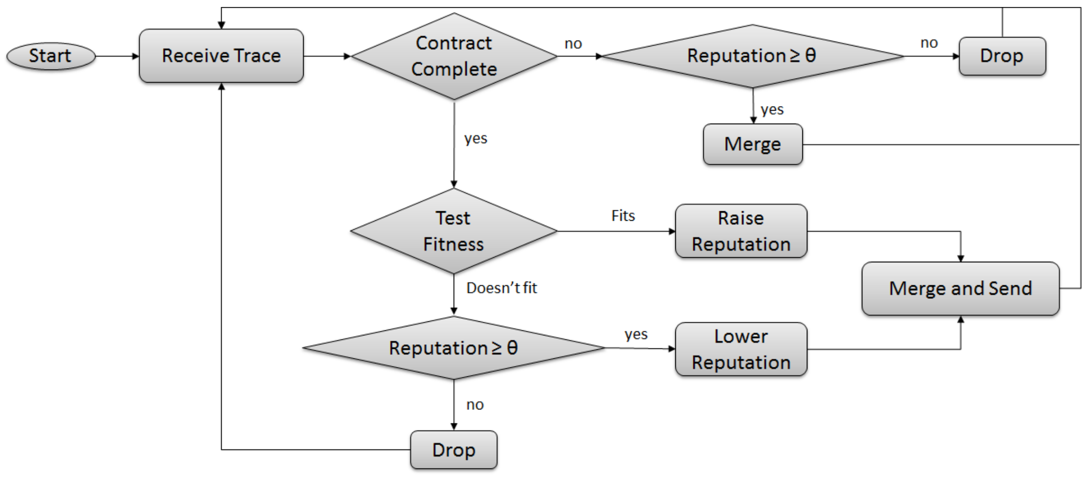


Figure 3. Behavior of PICARD Server upon the Reception of a New Execution Trace

3.1. *ActionNode*

To properly represent the contract, we introduce the notion of *ActionNode* [10], which is a cluster of related system calls that (i) are consecutively issued and (ii) are bound by some relation to represent an *action*, i.e., a high-level operation. In general, any relation among system calls can be used, e.g., any partition of all the subsets of system calls. However, in PICARD we only consider those relations that produce a meaningful action. For instance, an ActionNode can be composed by the system calls performed consecutively on the same file, where the relation expresses the fact that all these system calls work on the same file descriptor to produce a relevant action. As we will show in detail, all the system calls forming an ActionNode represent a sub-graph of system calls, while the collection of ActionNodes modeling the app behavior is in turn a graph.

As an example of ActionNode, see Figure 4, consider the sequence of system calls: `open(A) - read(A) - read(A) - close(A)`, where A is the filename. This ActionNode, called `Read_File`, represents at high level the action of reading consecutively data from file `A`: this action requires that, firstly, the file has to be opened, then data is read in a loop and, finally, the file is closed. Some other examples of ActionNodes are depicted in Figure 5. Notice that each ActionNode is actually a graph of system calls. The three examples report, ordered left-to-right, the actions of opening, writing, and closing a file (where the same list of operations is performed at least twice), the action of reading a file and then manipulating the underlying device parameters (probably a socket), and finally the action of reading from a previously opened file.

Several advantages stem when using ActionNodes. In fact, by representing the execution graphs through ActionNodes, the app traces can be seen as a graph whose nodes are high-level actions. This representation is more meaningful and compact than a graph whose nodes are just system calls. In fact, generally a program executes several system calls that, taken as standalone in a trace, only give limited information about the app behavior. This happens because only few system call types are issued by apps repeatedly. In fact, by representing the traces through a graph where each node

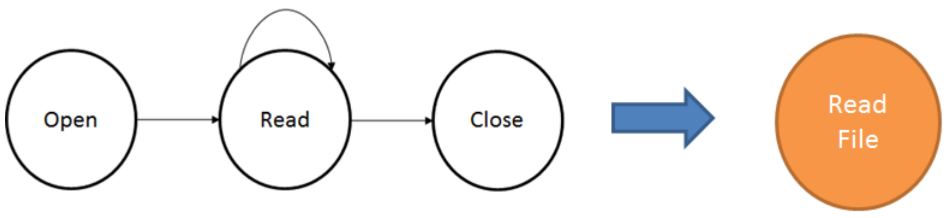


Figure 4. Definition of an ActionNode

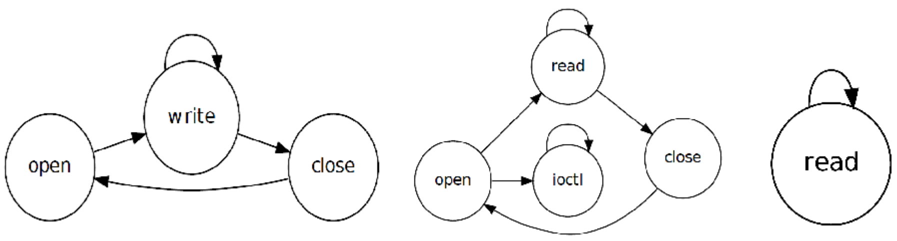


Figure 5. Three Examples of ActionNodes

is a different system call, then the program behavior would be represented by a graph with few nodes and a large amount of edges, which form a full mesh. However, from this kind of graph it is not possible to extract a significant contract capable of representing the probabilistic high-level behavior of the app.

The internal nodes that compose an ActionNode, i.e., system calls, are called *SysCallNodes*. In the following, we consider as SysCallNodes system calls that act on files, namely the `open`, `read`, `write`, `close`, `ioctl` system calls. Algorithm 1 reports the pseudocode for the generation of the graph of ActionNodes from the system call traces. When traversing the trace of system calls generated by a monitored app, the algorithm checks, for each SysCallNode, if the argument is the same as the previous one. Since the monitored system calls act on files, 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 that of the previous system call (see Figure 6), which can be seen as the parameter of the whole ActionNode. Different and subsequent system calls with the same argument are inserted into the same ActionNode (Figure 7). If the system call is the same as the previous one, with the same argument, then an edge (between SysCallNodes) is generated that goes back to the same SysCallNode. If the argument is the same, but the system call is different, a new SysCallNode is added to the current ActionNode, provided it has not been already created. In this case, an arc (between SysCallNodes) is added from the current SysCallNode to this existing SysCallNode. Then, each ActionNode is considered as a node representing the high-level operation.

After a new ActionNode is generated, an edge (between ActionNodes) is added from the previous ActionNode to the current one. Self loops may arise because there might be two consecutive actions that are represented by the same ActionNode. Furthermore, a new ActionNode is added only if a similar one does not exist already. To check if an ActionNode already exists, a comparison is made among the internal structure of the new ActionNode and of the existing ones (i.e., the structure of the graphs of the internal SysCallNodes must be the same). Notice that the ActionNodes are oblivious of the filename, meaning that the same cluster of operations performed on two different files generates the same ActionNode.

In the end, a graph of ActionNodes is generated. In the next step, to define the contract, PICARD takes as input the graph of ActionNodes and outputs a probabilistic automaton. As an illustrating

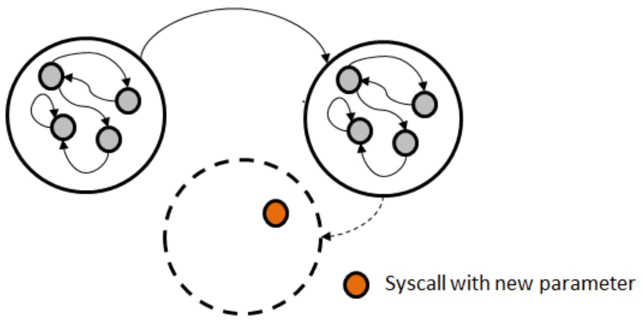


Figure 6. Generation of a New ActionNode

Algorithm 1: Algorithm for Generating ActionNodes

```
Input:
Trace: the trace of System calls.
Output: ActionNodeGraph: the graph of ActionNodes.
createActionNodeGraph(Trace)
begin
    actionNodeGraph = empty graph of ActionNode;
    actionNodeSet := empty set of ActionNode;
    currentActionNode := new empty ActionNode(“start”);
    currentActionNode.filename := “-1”;
    currentActionNode.syscallSet := empty set of sysCallNode;
    currentActionNode.syscallGraph := empty graph of sysCallNode;
    prevActionNode := undefined;
    prevSyscallNode := undefined;
    foreach syscall in Trace do
        if currentActionNode.filename is undefined then
            currentActionNode.filename := syscall.filename;
        end
        if syscall.filename $\neq$ currentNode.filename then
            if actionNodeSet contains a node AN with the same syscall graph then
                currentActionNode := AN;
            else
                add currentActionNode to actionNodeSet;
            end
            if prevActionNode is not undefined then
                actionNodeGraph := add a new edge from prevActionNode to
                currentActionNode
            end
            prevActionNode := currentActionNode;
            currentActionNode := new empty ActionNode;
            currentActionNode.filename := undefined;
            currentActionNode.syscallSet := empty set of sysCallNode;
            currentActionNode.syscallGraph := empty graph of sysCallNode;
            prevSyscallNode := undefined;
        else

        end
        if currentActionNode.syscallSet contains a sycall node SN labelled with syscall then
            currentSyscallNode := SN;
        else
            currentSyscallNode := new syscall node;
            set the label of currentSyscallNode to the name of syscall;
            add currentSyscallNode to the currentActionNode.syscallSet;
        end
        if prevSyscallNode is not undefined then
            currentActionNode.syscallGraph := add an edge from prevSyscallNode to
            currentSyscallNode;
        end
        prevSyscallNode := currentSyscallNode;
    end
    return ActionNodeGraph;
end
```

```
start, open(10), read(10), read(10), close(10), open(11), read(11), read(11), ↩
    close(11), open(10), read(10), read(10), close(10), ioctl(20), ioctl(20), ↩
    open(10), read(10), read(10), read(10), close(10), open(12), read(12), ↩
    read(12), read(12) , read(12) , close(12) , open(14) , write(14), write↩
    (14), close(14), ioctl(20), ioctl(20), ioctl(20), open(11), write(11), ↩
    write(11), close(11), open(14), write(14), write(14), close(14)
```

Table I. Original Trace of System Calls

example, Table I shows a simplified trace of system calls issued by an app where, for the sake of conciseness, only the system call name and the file descriptor are shown.

Figure 8(a) reports the graph of system calls representing the trace reported in Table I. The system calls are then clustered on the base of their parameter, in order to extract the ActionNodes presented in Figure 8(b). Finally, Figure 8(c) depicts the graph of ActionNodes obtained through the algorithm of Figure 6 and 7.

3.2. *Traces Analysis and Contract Generation*

In the previous section we have seen that by using the notion of ActionNode an execution trace of system calls can be represented as a graph of ActionNodes. In this section we give the formal representation of this type of graph, which is called *labeled multidigraph of ActionNodes* (LMA, for short). We then show how to derive a probabilistic contract from the LMA representing the observed app behavior.

Definition 3.1
A *labeled multidigraph of ActionNodes* (LMA) is a tuple (V, I, E, s, t, L), where V is the finite set of ActionNodes that describe the high-level operations performed by a specific app, $I \subseteq V$ is the set of initial vertices from which traces of observations can start, E is the finite set of edges, $s : E \to V$ is a mapping indicating the source vertex of each edge, $t : E \to V$ is a mapping indicating the target vertex of each edge, $L : E \to \mathtt{T}$ is a labeling function from edges to the trust domain $\mathtt{T}$ such that $L(e)$ denotes the trust level of the user who observed the execution associated to the edge e.

Notice that there can be both multiple occurrences of edges with the same source and target vertices and edges insisting on the same vertex. While the structure of the LMA is given by the user executing the specific app, the trust related labeling is defined by the central server – when receiving the LMA – on the basis of user's reputation. Then, from the union of the LMAs collected by the central server from all the collaborative users, which in turn is a LMA, a probabilistic contract is built. It is worth noticing that any new multidigraph that is submitted to the central server by a certain user is joined to the current one stored in the central database, and from the union of the two a novel probabilistic automaton is recomputed.

Formally, the probabilistic contract is defined in terms of a structure called *labeled probabilistic automaton* (LPA, for short) which is built from the LMA.

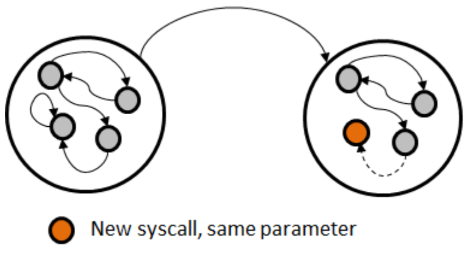
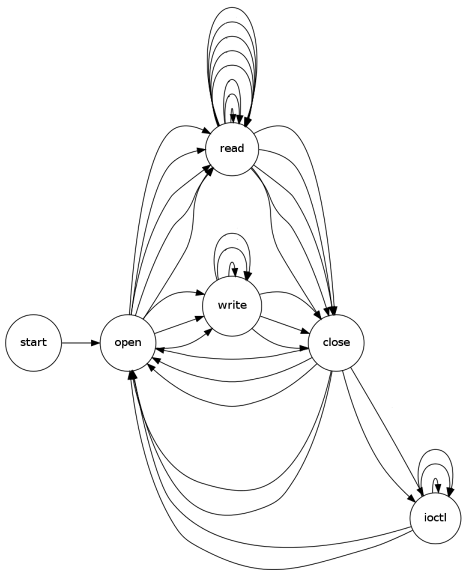
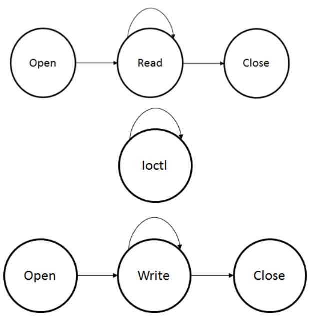


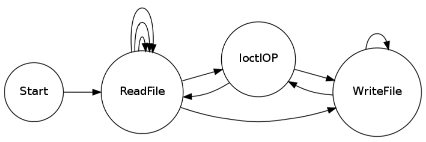
Figure 7. Insertion of a New Node in an ActionNode



(a) System call graph



(b) ActionNodes



(c) ActionNode graph

Figure 8. Transformation from System Call Graph to ActionNode Graph

Definition 3.2
A *labeled probabilistic automaton* (LPA) is a tuple (V, I, P, T) where V is the finite set of states, $I \subseteq V$ is the set of initial states, $P : V \times V \rightarrow [0, 1]$ is the transition probability function satisfying $\forall v \in V : \sum_{v' \in V} P(v, v') = 1$, and $T : V \times V \rightarrow \mathtt{T}$ is the transition trust function.

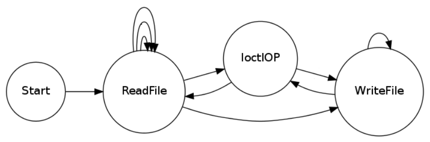
From the LMA (V, I, E, s, t, L) we derive the corresponding LPA (V, I, P, T), where P and T are defined as follows:

- $P(v,w) = \begin{cases} p & \text{if } \exists e \in E : s(e) = v \wedge t(e) = w \wedge p = \frac{mul(v,w)}{mul(v)} \\ 0 & \text{otherwise} \end{cases}$
  where $mul(v,w)$ is the multiplicity of the edges from v to w in E and $mul(v)$ is the number of outgoing edges from v in E.
- $T(v,w) = f\{\!|\, L(e) \mid s(e) = v \wedge t(e) = w \,|\!\}$ where f is any function that, applied to a multiset of trust values, returns a trust value.

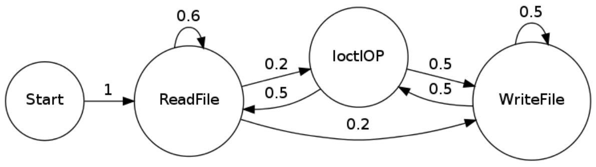
On one hand, notice that edges of the LMA with the same source and target vertices collapse into a unique transition in the LPA. Multiplicities of these edges in the LMA are used to compute the transition probability in the LPA. On the other hand, the trust level associated to a transition of the LPA derives from a combination of the trust levels of the edges of the LMA contributing to the transition. For instance, provided that the trust domain is totally ordered and numeric, the combination can be formalized through a mathematical function like, e.g., *min*, *max*, and *avg*. In particular, by using function *max* we assume that a transition is trusted as the most trustworthy user who observed its execution.

For instance, consider the transformation depicted in Figure 9, related to the same example of Figure 8. For the sake of simplicity, we abstract away from the information related to trust. Let us concentrate on the `ReadFile` vertex of the LMA, which has five outgoing edges, i.e., three self-loops, one edge directed to the `IoctlOP` vertex, and one edge directed to the `WriteFile` vertex. The corresponding state in the LPA has three outgoing transitions, i.e., one self-loop with probability $\frac{3}{5}$, expressing that three out of five edges departing from the vertex are self-loops, one transition towards state `IoctlOP` and one transition towards state `WriteFile`, both with probability $\frac{1}{5}$. Notice that the self-loop in state `ReadFile` summarizes three edges that may derive from the multidigraphs of different users. Hence, the trust level associated to such a transition results from a combination of the reputations of these users.

At run-time, the quantitative compliance of the app behavior with the contract is evaluated by comparing the frequency of the observed execution traces with respect to the probability distributions of the expected behaviors extracted from the LPA (V, I, P, T) associated to the contract. These distributions refer to the probability, when starting from any state $v \in I$, of observing distinct finite executions, which we call *longest distinct paths*. More precisely, a longest distinct path starting from v is a finite path that traverses every state in the path at most once, except possibly for the last one, which is either an absorbing state with no outgoing transitions or the unique state of the path visited twice. Intuitively, a longest distinct path describes a maximal finite observation of non-repeated behaviors. The motivation behind the choice of considering the longest distinct paths is that any path including two occurrences of the same state (different from the last one) can be actually viewed as the concatenation of two distinct observations. Hence, any execution trace that is



(a) ActionNode graph



(b) Contract

Figure 9. From LMA to LPA

observed at run-time represents a sequence of longest distinct paths, each one associated to its own frequency. As mentioned before and as we will see in the next section, such frequencies have to be compared with the probability distribution associated to the set of longest distinct paths of the LPA, which are calculated formally as follows.

Definition 3.3
Given a LPA (V, I, P, T), a *longest distinct path* starting from state $v_1 \in I$ is any finite sequence of states $v_1 \dots v_n$, with $n \geq 2$, such that:

- $P(v_j, v_{j+1}) > 0$ with $1 \leq j < n$;
- $\forall v_j, 1 \leq j < n$, there does not exist $k \neq j$, $1 \leq k < n$, such that $v_j = v_k$;
- either $\exists v_j, 1 \leq j < n$, such that $v_j = v_n$, or v_n has no outgoing transitions.

As usual, the probability of a path $v_1 \dots v_n$ is computed as the product of the probabilities of the transitions forming the path:

$$Prob(v_1 \dots v_n) = \Pi_{i=1}^{n-1} P(v_j, v_{j+1})$$

By following classical results related to probability distributions and measures for probabilistic systems [11], it is natural to construct a probability space over the set of distinct paths starting from any state $v \in I$ (in this set neither of the given paths is a prefix of another in the set). In particular, we have a unique probability distribution over the set of distinct paths of the same length starting from v, whose overall probability sums up to 1. Analogously, the finite set of longest distinct paths starting from v is measurable as well and defines a probability distribution.

As an example, reconsider the LPA of Figure 9 and examine the possible observations starting from state `ReadFile`. The related set of all the longest distinct paths is reported in the first column of Table II, which is constructed by exploring every possible sequence of transitions starting from state `ReadFile` and satisfying the three conditions of Def. 3.3. By calculating the related probabilities as discussed above, we obtain the probability distribution reported in the second column of Table II (RF stands for `ReadFile`, IO for `IoctlOP`, and WF for `WriteFile`).

Then, by exploiting the trust information labeling the transitions of the LPA, we associate a trust level to each longest distinct path. Intuitively, such a trust level shall be a combination of the trust values associated to the transitions forming the path. Similarly as previously argued for the definition of function T in Def. 3.2, several functions are candidates to characterize such a combination. However, in order to favor a conservative and cautious approach, we argue that the trust level of a path should not be higher than the trust level of its weakest transition. For instance, we advocate the use of function *min*, i.e., the trust level of a path depends on the least trustworthy transition forming the path.

To summarize, the probabilistic behavior specified by the contract is given by the set of probability distributions associated to the longest distinct paths starting from the potential initial states of the LPA, each of these paths equipped with the related trust level. A fundamental issue is related to determining whether such a set represents a complete, exhaustive contract for the app, i.e., it describes all the possible expected behaviors of the app together with the related probabilities of being observed. This is not a trivial issue that can be solved by comparing the obtained

Path	Probability
RF-RF	0.6
RF-IO-RF	0.1
RF-IO-WF-IO	0.05
RF-IO-WF-WF	0.05
RF-WF-IO-RF	0.05
RF-WF-IO-WF	0.05
RF-WF-WF	0.1

Table II. Longest Distinct Paths Starting From State `ReadFile` (RF) of Figure 9

contract with the set of potential behaviors determined, e.g., statically, and that characterize the app functionalities, because the contract includes quantitative, probabilistic information requiring some form of validation. In other words, we need an objective condition establishing that the quantitative information extracted from the LPA is sufficient to characterize the app behavior and, therefore, the obtained contract is ready for distribution to the collaborative users. As anticipated formerly, this level of acquired knowledge shall be stated by a *completeness* index, which is based on the accuracy of the quantitative information characterizing the contract. To this aim, we notice that initially the construction of the probability distributions of the longest distinct paths passes a transient phase during which they change by virtue of the LMAs added by users. Ideally, after such a transient phase, these distributions shall reach a steady state, which would represent the completeness proof. In order to approximate such a theoretical result, we assume that the contract is complete and ready for distribution whenever the last n consecutive trace executions received by the central server contribute to alter the probability distributions up to a tolerance threshold ϵ. The choice of these two parameters defines the tradeoff between accuracy of the obtained contract and time waited prior deployment of the contract. We point out that reducing the delay between app release and contract release is important from user's standpoint, who needs to check the compliance of the own installed app with respect to the contract as soon as possible. On the other hand, it is also worth recalling that even a complete contract continues to be adjourned by further contributions provided by users. These observations suggest to reduce the waiting time and augment the level of approximation, by acting on parameters n and ϵ, in order to anticipate as much as possible the contract release by sacrificing its correctness as little as possible.

4. CONTRACT COMPLIANCE

Once the contract has been built, it can be used to verify the compliance of different versions of the same app on distinct smartphones where these versions have been installed. A non-compliant app is an app exhibiting a behavior different from the one declared in the contract. More specifically, an app is non-compliant when it performs one or more operations not included in the contract, i.e., *functional misbehavior*, or when a sequence of operations is observed with a probability distribution appreciably different from the one associated to the same sequence as defined in the contract, i.e., *non-functional misbehavior*. Functional misbehaviors are captured by solving a subgraph isomorphism problem [12] between the functional projection of the LPA underlying the contract (obtained by removing probabilities from the transitions) and an analogous version of the LPA that is extracted from the observed behavior of the app at run time. On the other hand, non-functional misbehaviors require the analysis of quantitative behaviors. In the following, we consider such a case by showing how to compare quantitatively the probabilistic behaviors of the contract and of the app at run-time.

Let C be the probabilistic contract of an app A. We want to verify the compliance of A', a possibly different version of A, against C. To this end, we monitor the behavior of A' by progressively building the ActionNodes resulting from the observations and then extracting both functional and nonfunctional characteristics of the longest distinct paths that are obtained through the method discussed in the previous section. Then, the resulting probability distributions are compared with those forming the signed contract C. The comparison among probability distributions is typically estimated by means of similarity or distance measures [13]. In particular, we propose the usage of two of these metrics.

The first metric we consider is based on the *Pearson's Chi Squared Test* [14] for estimating the consistency of the behavior of A' with respect to the probability distributions characterizing C. The Pearson's Chi Squared Test belongs to a family of distance measures containing the Squared Euclidean distance $\sum_i (x_i - y_i)^2$, where x_i and y_i denote the probability values of the i-th pair of elements under comparison, and is used in statistics to verify if a sample is statistically consistent with respect to a known probability distribution. In our setting, the events generated by A' represent the statistical sample, whilst the contract describes the known distribution associated to the observable longest distinct paths.

At run-time, by monitoring the execution of A' we incrementally build the probabilistic behavior of the resulting longest distinct paths. Given a certain state v, to which the contract C associates n possible longest distinct paths, let us denote with O_i the probability associated to the observation of the i-th longest distinct path during the execution of A', and with E_i the expected probability of the same path as stated by the contract C. Then, the chi-squared χ^2 is computed according to the following formula:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

To verify the chi-squared null hypothesis, i.e., the behavior of A' has a distribution consistent with the one described in the contract C, the test statistic is drawn from the chi-squared distribution. If the computed probability is higher than conventional criteria for statistical significance the null hypothesis is not rejected, i.e., the behavior is compliant with the contract and the app should not be considered repackaged.

By following such an approach, however, trust is not taken into account. In order to consider this additional dimension, we propose to discount the result of the i-th comparison by a factor proportional to the trust associated to E_i. Formally, we normalize the trust values to obtain a value in the interval $[0, 1]$ and then we multiply the normalized trust associated to the i-th longest distinct path by the term $\frac{(O_i - E_i)^2}{E_i}$. In this way, we emphasize that differences about untrusted observations are more tolerated with respect to analogous differences about trusted observations.

We point out that, by virtue of the chosen measure, a behavior with non-zero probability in A' that, instead, does not occur in C, cannot involve any comparison, to emphasize that in this case the observed behavior is non-compliant with the expected behavior specified by the contract. Notice that such a misbehavior would be revealed also in a purely functional setting. However, while in the functional setting such a result is simply binary – a new, unexpected behavior occurs that is not allowed by the contract – in the quantitative setting we can estimate the probability of observing such a misbehavior, which, if negligible, could be tolerated to some extent.

To this end, we need a distance measure that allows PICARD to compare distributions over sets of elements some of which could be associated to null probabilities. A candidate metric is the Lorentzian measure, which combines the absolute difference with the natural logarithm:

$$Lor = \sum_{i=1}^{m} ln(1 + |O_i - E_i|)$$

where the term 1 is added to ensure non-negativity and to eschew the log of zero, while m refers to the cardinality of the union of the sets of longest distinct paths observed in A' and those defined in C.

Finally, it is worth noticing that the statistical analysis for contract compliance is employed to verify *a posteriori* the statistical similarity between the probabilistic behavior expected by the contract and the probabilistic behavior observed by every collaborative user who has sent a proper execution trace to the central server in order to generate the contract. Obviously, the same test is executed also for each further execution trace received by the central server even after the generation of the probabilistic contract. The objective of such a similarity analysis is to adjust the reputation of every collaborative user proportionally to the fitness of the execution trace provided by the user with respect to the contract. In particular, strong dissimilarities identify possibly malicious execution traces provided by users who, deliberately or not, generated false or inaccurate multidigraphs. Hence, to keep track of the result of such a similarity analysis, the reputation of the users involved is increased (resp., decreased) if the distance measure is below (resp., beyond) a given trust (resp., untrust) threshold, by an amount proportional to the difference.

Application	Traces	Length (min)	ActionNodes	Edges	Misbehavior
`TicTacToe`	100	10	7	30	Send SMS
`LunarLander`	50	10	11	37	Send SMS
`BaseballSuperstar`	100	15	11	46	Geinimi
`AngryBirds`	100	10	21	103	Geinimi
`K-Launcher`	50	15	13	52	KMIN
`Jewels`	50	15	18	78	PJAPPS
`Hamster Super`	50	10	9	35	YZHC
`Tower Defense`	50	10	18	49	Geinimi

Table III. Data About Traces Used to Build the Contracts

5. EXPERIMENTAL RESULTS

PICARD has been implemented for Android devices, mainly because of the wide distribution of this mobile OS and its openness. Android is in fact an open source project, based on a custom Linux kernel, which also allows for building customized versions of the OS, called custom ROMs. The execution traces of system calls are captured by using a kernel module, which hijacks the traced system calls and writes on a file the system calls along with the relevant parameters (e.g., the file descriptor). In the current version, the tested smartphones need to be rooted, since command `insmod`, which is used to run the tracing kernel module, can be issued only by a super user in the Android kernel. After hijacking the called system calls, PICARD passes the collected information to the application level through a file buffer shared between the two levels. At the application level, PICARD checks continuously the shared buffer and stores the monitored system call traces.

In order to avoid that an app can interfere, possibly maliciously with other apps running on the mobile device, the Android Linux Kernel enforces isolation cleverly exploiting the multi-user feature. In fact, each app runs in a Dalvik Virtual Machine (DVM), which is an optimized version of the Java Virtual Machine (JVM) acting as a sandbox for the app. Every DVM (i.e., app) receives a Linux User-ID and is treated as a Linux user by the kernel. Hence, each DVM has an `home` directory and its own memory space. Moreover, every app can launch Linux processes and threads whose owner is the associated DVM, identified by the User-ID. The User-ID is extracted from the package of the specific app and we exploit this feature to retrieve the actions performed by a specific app at the kernel level.

To prove the effectiveness of PICARD in discerning between genuine and trojanized apps, we tested our approach on a set of Android apps. The aim of PICARD is to recognize if an app is repackaged, i.e., not compliant with a probabilistic contract. To this end, we have analyzed a set of apps of which we were able to analyze both the genuine and the repackaged version. In the following, we report details about eight representative apps that we consider meaningful. The app contracts have been built running the genuine apps, downloaded from the official Google Play market, on real devices (Samsung Galaxy Nexus with Android 4.0) collecting traces of real usage of different users.

We report in Table III the list of tested apps including the number of collected traces, their length, the amount of action nodes and edges of the app contract and the kind of misbehavior or malware name found in their repackaged version. The first two applications are distributed as sample, together with the Android SDK. Thus, modifying the source code to introduce a misbehavior is simple. The other apps are real repackaged apps found in the wild. In particular, they have been downloaded either from a repository of malicious applications[*], or from an unofficial app market[†]. In all the tests, PICARD has been effective in recognizing traces coming from all the malicious apps as non-compliant with the contract. For the last seven apps of Table III, functional misbehaviors have been

[*] http://contagiominidump.blogspot.com/
[†] appbrain.com.

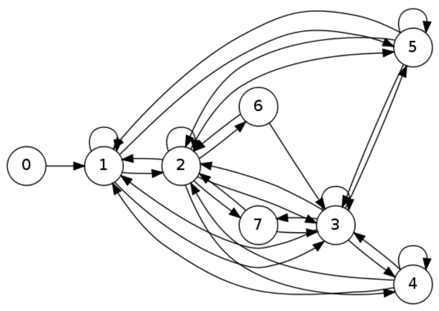


Figure 10. Probabilistic Contract of TicTacToe

detected through the comparison between the contract resulting from the generated ActionNodes and the behavior resulting from several execution traces of the repackaged versions. Only for the `TicTacToe` app the functional analysis was not enough, while statistical analysis was necessary to reveal the malware.

In Figure 10 we report the contract of the Android app `TicTacToe`, where transition probabilities are omitted for the sake of readability. We emphasize that the set of execution traces used to generate this contract is representative of the app behavior and, therefore, adequate to generate a complete contract. In particular, with respect to the completeness criteria discussed in Section 3, we point out that the last of these traces contributes to changes in the probability distributions specifying the contract below the tolerance threshold $\epsilon = 10^{-5}$. It is also worth noticing that traces longer than the ones collected are not representative of the real usage of the app. In fact, it is unlikely that a user uses continuously one of the reported apps for more than 10-15 minutes.

Afterward, we extracted from a trojanized version of `TicTacToe` 10 traces of a time span of 15 minutes. This trojanized version of TicTacToe sends an SMS message to a phone number each time the user opens the activity to change the graphic style of the game. This action has been triggered at least once for every monitored trace. As an example, we report in Table IV a list of some longest distinct paths starting from state 5 and the related probability distribution both in the contract and in one of the monitored traces.

To verify if the monitored traces are compliant with the app contract, the statistical tests of χ^2 and Lorentzian measure have been used as described in the previous section. The contract of the app includes $m = 7$ probability distributions, as every state in the LPA is a potential initial state, except for node 0 that represents a fictitious initial state. The two statistical tests are performed for all these probability distributions for each monitored trace. The observed behavior of the

Path	E_i	O_i
5-3-1-1	0.00005476	0
5-3-1-3	0.0002	0
5-3-1-5	0.000001	0
5-3-2-2	0.00188	0
5-3-2-3	0.0034	0.0003
5-3-2-5	0.00002	0.000026
5-3-3	0.0004	0.12
5-3-4-1	0.0000012	0.0012
5-3-4-3	0.00014	0.0054
5-3-4-4	0.000189	0.0024
5-3-5	0.0031	0.0015
5-3-7-3	0.0000156	0.00075

Table IV. Comparison Related to Some Longest Distinct Paths From state 5

app is compliant with the contract if the null hypothesis of the test is verified. For the χ^2 test, the null hypothesis h_0 that has to be checked for each probability distribution $i \in \{1, \ldots, m\}$ is $\chi_i^2 \leq \gamma$, where γ is the critical value of the χ^2 distribution for one degree of freedom (as we are considering one monitored trace) and a significance level α representing the desired tolerance (which is a configurable parameter of the framework). As an example, by computing the χ^2 test on the distribution of longest distinct paths outgoing from state 5, which represents the action node `read(A)-read(A)`, we obtain a non-compliance result. For instance, the value returned by the test for a representative monitored trace is $\chi_5^2 = 37.22$. Notice that the critical value corresponding to tolerance $\alpha = 0.999$ (which is a very high tolerance) is $\gamma = 10.828$, which is much lower than the value returned by the test. Thus, the null hypothesis h_0 is not verified and the monitored trace is correctly considered non-compliant with respect to the contract.

By performing the Lorentzian test on the same data, the obtained result is $Lor_5 = 0.13$. In order to favor a correct interpretation, we compare this value with the ones obtained from monitored traces belonging to the genuine version. The highest value returned from a genuine trace is $Lor = 0.0012$, which is two orders of magnitude lower than Lor_5.

As far as performance aspects are concerned, we report the overhead of the testing of PICARD executed on a Samsung Galaxy Nexus with Android 4.0. The measured overhead of the PICARD app is 7% of the CPU usage and 3% of RAM occupation when performing compliance check with the contract. The PICARD app causes less overhead when simply collecting traces and sending them to the PICARD server. The overhead of the kernel module for system call monitoring is 5% for CPU usage while RAM usage is negligible.

6. RELATED WORK

In the literature, system call analysis has been proposed in several systems to monitor and detect malicious behaviors. One of the first network intrusion detection systems (IDS) based on state transition graphs analysis is NetSTAT [15]. Analysis of system calls with Markov models have formerly been performed on other operating systems. For instance, Hoang and Hu [16] propose a scheme for intrusion detection. This model is based on system calls and hidden Markov models and is able to detect efficiently denial of service attacks. Maggi et al. [17] present another model relying on system calls and Markov models to detect intrusions. Their system analyzes the arguments of the system calls but is oblivious of the system call sequence. System call sequences and deterministic automata have been used by Koresow [18] to detect anomalies, which are revealed when system call sequences differ from an execution trace known to be secure. *Crowdroid* [19] is an Android-based IDS that is based on the number of system calls issued by an application. Misbehaviors are identified by applying computational intelligence techniques.

A behavioral analysis of Android applications at the system call level is presented in [20]. The authors propose a framework called CopperDroid that discerns good behaviors from bad ones. Copperdroid tries to automatically stimulate malicious applications to misbehave through instrumentation. The analysis of behaviors is automatic, which means that the behavior of the application stimulated by user interaction is not considered. Another approach that aims at classifying malicious behaviors is presented in [21], which aims at finding similar sub-graphs describing common behaviors of malicious apps. A classifier is then used to determine whether an app is malicious or not. The approach requires each app to be run in a sandbox, which may alter the execution of some actions that are usually performed on real devices. A framework for analysis of Android apps is presented in [22]. The proposed framework emulates in a virtual sandbox all the components of the Android framework and can be used to analyze the behavior of Android apps by collecting traces at the Dalvik level. This framework can be complementary to the PICARD approach and as future work it would be worth considering the possibility of using the Dalvik level traces for higher level ActionNodes. The framework proposed in [23] analyzes the behavior of Android applications by running them on emulators in desktop environments. Each application is forced to execute with as many input as possible, in order to discover as many hidden behaviors as possible. The security analysis is then performed offline on the functions called at the *smali* (bytecode) level. This approach is different from the one of PICARD, which aims at finding discrepancies between the application behavior and the contract. Thus, the concept of "misbehavior" in PICARD is more general, which should be more effective in detecting zero-day attacks.

Some Android security frameworks try to protect the system by monitoring the communication level and defining security policies. One of these systems is CRePE [24], which allows the definition of context based security policies. Another typical approach to monitor Android app behavior is called tainting. In this approach sensitive data flow is tracked to check the apps that are able to access a specific piece of information. Examples of this approach on Android are [25] and [26]. The tainting approach is more aimed at detecting privacy leakage, whilst the PICARD approach addresses a more general concept of misbehavior. In [27] a framework based on SELinux is proposed to enforce security policies on Android devices and to tackle common misbehaviors performed by malware. The approach is more focused on the application of security policies than on detection of malicious apps. Moreover, a custom version of the operating system is required. *Aurasium* [28] is another security framework for Android devices, which is able to enforce security policies through repackaging of all the installed apps. Aurasium forces the apps running on the device to call modified APIs, which are used to perform security checks. Aurasium is aimed at enforcing security policies instead of the detection of repackaged apps.

Another approach that exploits application contracts to monitor the behavior of mobile applications is given by the Security-by-Contract framework [29]. This framework has been extended [30] and applied in several ways and in different scenarios [31, 32]. Differently from PICARD, the security by contract framework does not perform intrusion and malware detection, but matches the application behavior with a security policy. Thus, malicious behavior that is not specified in the security policy is not considered. Referring to quantitative models, probabilistic contracts have been firstly introduced by Delahaye et al. [33, 34, 35] for analyzing reliability and availability aspects of systems. Their approach to contract definition and compliance check is static and is not related to practical applications in the field of intrusion detection. On the other hand, we have shown that PICARD is based on a dynamic approach using statistical analysis that turns out to be effective in detecting trojanized apps. Bielova and Massacci [36] present a notion of distance among traces characterizing enforcement strategies by the distance from the original trace. This approach is generic and, differently from PICARD, it does not consider low level actions, nor proposes real applications. The chi-squared test has been used to detect anomalies in several different fields. For instance, an application to network traffic analysis is presented by Ye and Chen [37].

The work presented in [38] describes a system to classify malicious apps on the base of the API calls they perform. The approach uses a classifier that statically analyzes features related to the called methods, with a particular attention to intercommunication constructs. This approach is effective in foreseeing malicious misbehaviors which are functional, albeit it is less likely to discover non-functional misbehaviors.

7. CONCLUSION AND FUTURE WORK

In this paper we have proposed PICARD, a collaborative framework for generating and checking Android apps' probabilistic contracts. PICARD performs analysis of the app behavior at run-time, by building the contract dynamically. To this aim, we have introduced the concept of ActionNode in order to describe the behavior of apps through clustered graphs and probabilistic automata. PICARD discerns between genuine and trojanized apps, by revealing both functional and non-functional misbehaviors.

The dynamic approach used by PICARD is more specific with respect to an approach relying on the integrity check based on the checksum of the `apks`. In fact, if an app is updated, e.g., by changing only some of its data, such as an utility app that changes a background picture, the app still performs the same actions as the former version and, hence, has the same behavioral contract but, however, the checksum is different. Moreover, in PICARD the analysis is conducted through statistical tests performed on the app execution traces described in terms of probabilistic automata.

Hence, the PICARD approach does not differentiate the behaviors in "known" and "unknown" only, but it is also able to distinguish between likely and unlikely behaviors. The dynamic analysis of PICARD does not require the program to be decompiled. Moreover, since the contract is built by monitoring real user behaviors, it is possible to detect misbehaviors that may not be noticed through static analysis. Thus, PICARD can be viewed as a framework complementary to static analysis systems.

The effectiveness of PICARD depends strongly on the cooperation of several users, thus motivating the use of a reputation system stimulating collaborative, honest behaviors. In particular, the proposed approach is based on a computational notion of trust inspired by the Jøsang model [39], which keeps track of the history of the user behavior, by increasing reputation for each correct report and by decreasing it otherwise. A similar approach is presented in [40]. Moreover, in our approach the quantitative behavior of the app reported by users is weighted by the reputation of the user, while the notion of contract compliance is based on the comparison between probability distributions through a distance metric. Employing reputation-based weights and similarity tests to rate the credibility of the reported feedback is not a completely naive approach and is used, e.g., in the TrustGuard framework [41], which defines heuristics to mitigate dishonest feedback. In order to discourage false or inaccurate feedback, more sophisticated cooperation incentives can be used (see, e.g., [42, 43]), possibly based on some form of remuneration (see, e.g., [44]).

The validation of the trust configuration policies and parameters discussed in Sections 3 and 4 is left to sensitivity analysis in future work. We also plan to extend the experimental studies by verifying whether the reputation system of PICARD is effective to isolate selfish behaviors due to non-collaborative users and to detect and punish (possibly orchestrated) malicious behaviors due to collaborative users who deliberately cheat by providing false feedback during the contract generation.

Further extensions are related to the notion of ActionNodes, which, in the current implementation, are built on system call graphs. In particular, the concept of ActionNode can be extended including complex nodes that describe higher level actions, such as "Send text message". Finally, we also plan to extend the experiments on a larger set of apps, with the possibility of including an automatic analysis approach, as in [45] and [46].

REFERENCES

1. Fsecure mobile threat report q1 2014 2014. URL `http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2014.pdf`.
2. Pandalabs quarterly report january-march 2014 2014. URL `http://press.pandasecurity.com/wp-content/uploads/2014/05/Quaterly-PandaLabs-Report_Q1.pdf`.
3. Viennot N, Garcia E, Nieh J. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.* Jun 2014; **42**(1):221–233, doi:10.1145/2637364.2592003. URL `http://doi.acm.org/10.1145/2637364.2592003`.
4. La Polla M, Martinelli F, Sgandurra D. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE* quarter 2013; **15**(1):446 –471, doi:10.1109/SURV.2012.013012.00028.
5. Dini G, Martinelli F, Saracino A, Sgandurra D. MADAM: A Mult-Level Anomaly Detector for Android Malware. *6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2012, St. Petersburg, Russia*, vol. 7531 - 2012, Springer Verlag, 2012.
6. Trojans are flipping over flappy bird 2014. URL `http://www.trojansnews.com/2014/01/trojans-are-flipping-over-flappy-bird`.
7. AP Felt, E Chin, S Hanna, D Song, D Wagner. Android Permissions Demystified. *8th ACM conference on Computer and Communications Security (CCS'11)*, ACM (ed.), 2011; 627–638.
8. AP Felt, E Ha, S Egelman, A Haney, E Chin, D Wagner. Android permissions: User attention, comprehension, and behavior. *Technical Report*, Electrical Engineering and Computer SciencesUniversity of California at Berkeley 2012. Http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.html.
9. Aldini A, Martinelli F, Saracino A, Sgandurra D. A collaborative framework for generating probabilistic contracts. *International Conference on Collaboration Technologies and Systems (CTS 2013)*, IEEE, 2013; 139–142.
10. Dini G, Martinelli F, Saracino A, Sgandurra D. Probabilistic contract compliance for mobile applications. *ARES*, IEEE Computer Society, 2013; 599–606.
11. Forejt V, Kwiatkowska M, Norman G, Parker D. Automated verification techniques for probabilistic systems. *Formal Methods for Eternal Networked Software Systems*, *LNCS*, vol. 6659, Bernardo M, Issarny V (eds.), Springer, 2011; 53–113.
12. Kuramochi M, Karypis G. Frequent subgraph discovery. *1st IEEE International Conference on Data Mining*, IEEE, 2001; 313.

13. Cha SH. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences* 2007; **1**(4):300–307.
14. Plackett RL. Karl Pearson and the Chi-Squared Test. *International Statistical Review / Revue Internationale de Statistique* 1983; **51**(1):59–72.
15. Vigna G, Kemmerer RA. Netstat: A network-based intrusion detection system. *J. Comput. Secur.* Jan 1999; **7**(1):37–71.
16. Hoang X, Hu J. An efficient hidden Markov model training scheme for anomaly intrusion detection of server applications based on system calls . *12th IEEE International Conferecence On Networks, ICON 2004*, vol. 2, IEEE, 2004; 470–474.
17. Maggi F, Matteucci M, Zanero S. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing* October-December 2010; **7**(4).
18. Koresow A. Intrusion detection via system call traces. *Software* 1997; **14**(5).
19. Burguera I, Zurutuza U, Nadijm-Tehrani S. Crowdroid: Behavior-based malware detection system for android. *SPSM'11*, ACM, 2011.
20. Reina A, Fattori A, Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *Proceedings of the 6^{th} European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, 2013.
21. Martinelli F, Saracino A, Sgandurra D. Classifying android malware through subgraph mining. *DPM/SETOP*, 2013; 268–283.
22. Yan LK, Yin H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USENIX Association: Berkeley, CA, USA, 2012; 29–29. URL http://dl.acm.org/citation.cfm?id=2362793.2362822.
23. Johnson R, Stavrou A. Forced-path execution for android applications on x86 platforms. *Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, SERE-C '13, IEEE Computer Society: Washington, DC, USA, 2013; 188–197, doi:10.1109/SERE-C.2013.36. URL http://dx.doi.org/10.1109/SERE-C.2013.36.
24. Conti M, Nguyen V, Crispo B. CRePE: context-related policy enforcement for android. *ISC'10 Proceedings of the 13th international conference on Information security* , Springer-Verlag, 2010; 331–345.
25. Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM: New York, NY, USA, 2014; 259–269, doi:10.1145/2594291.2594299. URL http://doi.acm.org/10.1145/2594291.2594299.
26. Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM* Mar 2014; **57**(3):99–106.
27. Bugiel S, Heuser S, Sadeghi AR. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, USENIX Association: Berkeley, CA, USA, 2013; 131–146. URL http://dl.acm.org/citation.cfm?id=2534766.2534778.
28. Xu R, Saïdi H, Anderson R. Aurasium: Practical policy enforcement for android applications. *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USENIX Association: Berkeley, CA, USA, 2012; 27–27. URL http://dl.acm.org/citation.cfm?id=2362793.2362820.
29. Dragoni N, Martinelli F, Massacci F, Mori P, Schaefer C, Walter T, Vetillard E. Security-by-contract (SxC) for software and services of mobile systems. *At your service - Service-Oriented Computing from an EU Perspective.*, MIT Press, 2008.
30. Costa G, Dragoni N, Issarny V, Lazouski A, Martinelli F, Massacci F, Matteucci I, Saadi R. Security-by-Contract-with-Trust for mobile devices. *JOWUA* Dec 2010; **1**(4):75–91.
31. Dragoni N, Massacci F. Security-by-contract for web services. *SWS*, 2007; 90–98.
32. Gadyatskaya O, Massacci F, Philippov A. Security-by-Contract for the OSGi Platform. *IFIP TC 11 Information Security and Privacy Conference*, 2012; 364–375.
33. Delahaye B, Caillaud B, Legay A. Probabilistic contracts: A compositional reasoning methodology for the design of stochastic systems. *10th International Conference on Application of Concurrency to System Design (ACSD), 2010*, IEEE, 2010.
34. Delahaye B, Caillaud B, Legay A. Probabilistic contracts: a compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Formal Methods in System Design* 2011; **38**(1):1–32.
35. Delahaye B, Caillaud B. A model for probabilistic reasoning on assume/guarantee contracts. arXiv preprint arXiv:0811.1151 2008.
36. Bielova N, Massacci F. Predictability of enforcement. *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, vol. 6542, Springer, 2011; 73–86.
37. Ye N, Chen Q. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Quality and Reliability Engineering International* 2001; **17**(2):105–112.
38. Aafer Y, Du W, Yin H. Droidapiminer: Mining api-level features for robust malware detection in android. *Security and Privacy in Communication Networks*, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 127, Zia T, Zomaya A, Varadharajan V, Mao M (eds.). Springer International Publishing, 2013; 86–103, doi:10.1007/978-3-319-04283-1_6. URL http://dx.doi.org/10.1007/978-3-319-04283-1_6.
39. Josang A. Trust-based decision making for electronic transactions. *Proceedings of the Fourth Nordic Workshop on Secure Computer Systems (NORDSEC99)*, 1999; 496–502.
40. Bhattacharya R, Devinney TM, Pillutla MM. A formal model of trust based on outcomes. *Academy of management review* 1998; **23**(3):459–472.

41. Srivatsa M, Xiong L, Liu L. Trustguard: countering vulnerabilities in reputation management for decentralized overlay networks. *Proc. of WWW'05*, ACM Press, 2005; 422–431.
42. Fernandes A, Kotsovinos E, Ostring S, Dragovic B. Pinocchio: Incentives for honest participation in distributed trust management. *Proc. of iTrust'04*, *LNCS*, vol. 2995, Springer, 2004; 63–77.
43. Yang M, Feng Q, Dai Y, Zhang Z. A multi-dimensional reputation system combined with trust and incentive mechanisms in p2p file sharing systems. *Proc. of IEEE Distributed Computing Systems Workshops*, 2007.
44. Bogliolo A, Polidori P, Aldini A, Moreira W, Mendes P, Yildiz M, Ballester C, Seigneur JM. Virtual currency and reputation-based cooperation incentives in user-centric networks. *Proc. of Wireless Communications and Mobile Computing Conference (IWCMC-2012)*, IEEE, 2012; 895–900.
45. Mirzaei N, Malek S, Păsăreanu CS, Esfahani N, Mahmood R. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes* Nov 2012; **37**(6):1–5.
46. Liu Y, Xu C. Veridroid: Automating android application verification. *Proceedings of the 2013 Middleware Doctoral Symposium*, MDS '13, ACM: New York, NY, USA, 2013; 5:1–5:6.