Noname manuscript No.
(will be inserted by the editor)

Ask A(n)Droid to Tell You the Odds: Probabilistic Security-by-Contract for Mobile Devices

Alessandro Aldini · Antonio La Marra · Fabio Martinelli · Andrea Saracino

Received: date / Accepted: date

Abstract Security-by-contract is a paradigm proposed for the secure installation, usage, and monitoring of apps into mobile devices, with the aim of establishing, controlling, and, if necessary, enforcing, security-critical behaviors. In this paper, we extend this paradigm with new functionalities allowing for a quantitative estimation of such behaviors, in order to reveal in real-time the more and more challenging subtleties of new generation malware and repackaged apps. The novel paradigm is based on formal means and techniques ranging from statistical analysis to probabilistic model checking. The framework, deployed in the Android environment, is evaluated by examining both its effectiveness with respect to a benchmark of real-world malware and its effect on the execution of genuine, secure apps.

Keywords security-by-contract · Android · statistical analysis · probabilistic model checking

1 Introduction

The exponential increase in popularity and computational power of mobile devices such as smartphones, tablets, and wearables, has rapidly driven the attention of application developers. The number of mobile applications (apps) available in digital markets has already passed the one million threshold [12] and currently they cover almost all the applicative categories. The two main target Operating Systems (OS) for mobile apps are Android and iOS. In particular, starting from 2012, Android is the most popular OS [12] for mobile devices and thus, it has both the largest number of available apps and the highest number of users to be reached by them.

Security represents a critical issue for the market of mobile apps, which are exposed to many types of cyber-attacks. Malicious developers strive to design mobile apps able to damage both users and devices, by using, e.g., Trojan horses, which may cause corporate (or personal) data (or money) theft and leakage [40]. Moreover, genuine applications often expose vulnerabilities, due to programming mistakes, permission overdeclaration and unprotected interfaces [19,24]. A recent study sponsored by IBM [37] reveals that companies test less than 50% of the mobile apps they build, while a company out of three does not conduct security tests before the app deployment at all. On the other hand, paradigms like the Bring-Your-Own-Device are adopted by an increasing number of companies that allow employees to use business apps on their personal devices, thus challenging the traditional security perimeters and risk management systems [7]. Similarly, nowadays a large amount of users feel confident in downloading and using apps that manage their credentials and personal information, like financial data or patient records, without paying attention to the risks deriving from the use of potential repackaged apps [24].

In this setting, it is standard to use security models based on the specification of *contracts* establishing the kind of actions the app can execute [34]. The underlying semantics is either based upon trust relationships or upon statements of purpose. In the former case, the users run a mobile app because they trust the app provider, essentially leading to a security model based on the "all or nothing" policy, that is either the app is trusted and, therefore, allowed to do anything, otherwise it is not installed [24]. In the latter case, the app provider declares the security relevant actions that could be performed by the app, so that the users can decide whether to run the app or not, by possibly restricting the app access rights. However, in such a case typically the semantics is too coarse-grained and scarcely user-friendly, as it happens in Android, where rights to perform security relevant actions are declared through permissions [18].

For these reasons, fully automated, contract-based intrusion detection systems are proposed to facilitate and make flexible the task of controlling the behavior

Antonio La Marra, Fabio Martinelli, Andrea Saracino are at:
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
E-mail: name.surname@iit.cnr.it

Alessandro Aldini is at:
Dipartimento di Scienze Pure e Applicate,
Università di Urbino
Urbino, Italy E-mail: alessandro.aldini@uniurb.it

of installed mobile apps. In particular, the Security-By-Contract (S×C) approach [21] is based on a mobile, formal notion of contract that accompanies the app and has a twofold objective. On one hand, it is used to establish whether the app behavior is compliant with the declared scope as targeted by the app builder. On the other hand, it represents the base for checking whether the app is secure with respect to specified policies. The validation of such a twofold compliance check is strictly related to the completeness and exactness of all the parameters at hand: the actual observations stating the app behavior, the statements describing the respected contract, and the specification of the security policies. However, in real-world scenarios, it can be difficult to estimate precisely some of these parameters while, at the same time, fluctuations of non-functional features may have an effect upon the compliance results, thus requiring approximation techniques.

In this paper, we present an improvement to the Security-By-Contract paradigm, called Security by Contract with Probability (S×C×P), which extends conservatively the logical architecture of the standard S×C model. The extension is based on a quantitative model to specify probabilistic behaviors of the applications and to define and enforce probabilistic policies. In particular, two alternative probabilistic models are considered for the verification of *action-based* policies and of *history-based* policies, respectively, where the latter turns out to be more expressive than the former. On one hand, action-based conditions refer to relative probabilities of observed events, while, on the other hand, history-based conditions are related to the probability distributions of sequences of actions, thus reasoning at the level of event consequentiality. The verification conducted through this twofold approach exploits a mechanism to intercept at run-time critical security-relevant actions, by evaluating application compliance in order to ensure policy matching for the protection of the execution environment.

Even if the framework is general enough to be virtually used in any application environment, in this paper we present an implementation for Android devices, discussing a set of policies for tackling actions of popular malware and limiting unwanted behavior of both malicious and genuine apps. The resulting policies prove to be expressive enough to effectively stop the action of two malware classes, namely *Spyware* and *SMS trojan*, without limiting critical operations of genuine apps.

This paper extends previous work [33] with the following new contributions:

– an approach, called history-based, to the probabilistic modeling and verification of policies, which is compared to the standard action-based one;
– the details of the implementation of the S×C×P framework about the methods and tools used for policy evaluation and enforcement;
– a new set of experiments to validate and compare from the expressiveness standpoint the two probabilistic approaches;
– a comprehensive evaluation and discussion of performance overhead for both approaches;
– an extended and up-to-date review of the state of the art.

The rest of the paper is organized as follows. Section 2 reviews the state of the art. Section 3 introduces the logical model of the proposed framework, including the related components and workflow. At first, we describe the basic S×C model and then introduce the extension for action-based and history-based probabilistic policy and contract definition. Then, Section 4 describes formally the methodology underlying the probabilistic compliance operations, both for action-based and history-based approaches. Hence, the implementation of the S×C×P framework as an app for Android devices is detailed in Section 5, which also introduces the security relevant actions considered in this specific implementation. Section 6 describes the experiments conducted on malicious and genuine applications, also reporting a performance analysis. More precisely, both action-based and history-based policies are designed to counteract the misbehavior of samples belonging to several well-known Android malware families. The effect of such policies on a set of genuine apps is tested, and the expressiveness comparison between the two approaches is considered. The section includes also a set of experiments to measure the effectiveness of run-time policy enforcement and to measure the system performance. Finally, Section 7 briefly concludes by proposing some future extensions.

2 Related Work

From the quantitative standpoint, the problem of finding an optimal control strategy is considered firstly in [22] in the context of software monitoring. In this setting, the system is represented as a Directed Acyclic Graph, while rewards and penalties with correcting actions are employed dynamically to find the optimal solution. Similarly, an encoding of access control mechanisms using Probabilistic Decision Processes is proposed in [36], where the optimal policy can be derived by solving the corresponding optimization problem.

From a different perspective, [11] proposes a notion of distance among traces representing the system behavior. If a trace is not secure, then it should be edited to a secure trace close to the non-secure one, where closeness is estimated in terms of the distance metric, thus characterizing an enforcement strategy. An approach for dynamic building of probabilistic contracts based on the observation of executable traces for Android applications is presented in [6]. The authors in this work focus on a different concept where frequent sequences of system calls are grouped in macro-nodes of a labeled graph of actions. Differently from our approach, the framework presented in this work performs analysis at system call level and does not offer enforcement tools. An approach for detection of non-compliant applications through the analysis of the system calls is proposed in [20] by using machine learning techniques to distinguish between standard behaviors and malicious ones. This work does not consider enforcement mechanisms, and the analysis is based on statistical concepts instead of probabilistic ones. In [28], a scheme for intrusion detection using probabilistic automata is proposed. This system exploits system calls and hidden Markov models and is able to detect efficiently denial of service attacks. In [32], a system based upon system calls and Markov models is proposed to detect intrusions. This 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 in [29] to detect anomalies whenever the system call sequences differ from an execution trace known to be acceptable. This approach might suffer from high false alarm rates, since any trace different from a known one is considered as malicious. Our approach relaxes this condition, allowing the definition of more complex and flexible policies. Alterdroid [41] is a tool that compares the behavioral differences between an original app and an automatically generated version that contains modifications (faults) to detect hidden malware. The method of [26] proposes malware detection based on embeddings of function call graphs in a vector space capturing structural relationships. This representation is used to detect Android malware using machine learning techniques. The present work is not necessarily focused on malicious apps, and potentially any kind of policy can be applied. Similarly, [43] classifies Android malware via dependency graphs by extracting a weighted contextual API dependency graph as program semantics to construct feature sets. A framework that allows the definition of security policies mainly intended to control the flow of information in the Android OS is TaintDroid, presented in [23]. Similarly as the S×C×P framework, TaintDroid performs the enforcement at runtime, still does not consider probabilistic conditions, nor enforces action related policies. Moreover, it requires a modification of the operating system. In [39], a framework is proposed for Android to enforce dynamically policies aimed at reducing the amount of traffic generated by Android apps. Differently from this work, the policies are very specific and do not consider probabilistic aspects. Another tool for the definition of policies and the analysis of Android applications is presented in [8]. The proposed framework, named R-Droid, performs static analysis, not at runtime, and does not consider probabilistic conditions. The authors of [42] present a framework named SPOKE for runtime analysis of security policies in Android applications, in order to find vulnerabilities and possible data disclosures. The SPOKE framework is not focused on enforcement aspects, mainly targeting the app vulnerabilities, instead of malicious and unwanted behaviors, which are instead the intended targets for the S×C×P framework. A framework that performs hybrid/static dynamic security policy enforcement on Android devices is presented in [35]. The framework has been specifically designed to tackle Android malware, targeting API calls related to specific misbehaviors. Differently from our approach, this framework mainly operates on classification and heuristics, and does not allow the definition of complex policies.

Referring to probabilistic models, probabilistic contracts have been firstly introduced in [16], where the contract generation is based on the analysis of the occurrences of system calls. The basic model relies on the Assume/Guarantee paradigm for stochastic systems and the goal of verification is to analyze both reliability and availability aspects of such systems. The approach we propose is intended to extend the Security-by-Contract model in the probabilistic framework, in order to assess quantitatively the constraints behind the adoption of enforcement strategies at execution time.

3 A Formal Approach to Security By Contract

In this section, we first recall the nondeterministic security-by-contract approach, and then we present the logical models of the proposed probabilistic framework.

3.1 Security By Contract

The Security-by-Contract (S×C) model [21] is based on three cornerstone elements: the app *code* A, the app *contract* C, and the client *policy* P. Given an app, its *contract* is a formal specification of the security relevant behavior that the app can exhibit during its execution. Such a behavior is related, e.g., to the security virtual machine API calls or the critical system calls

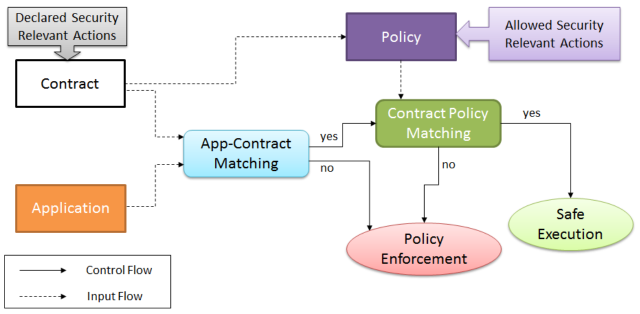

Fig. 1 Security-by-Contract components and workflow.

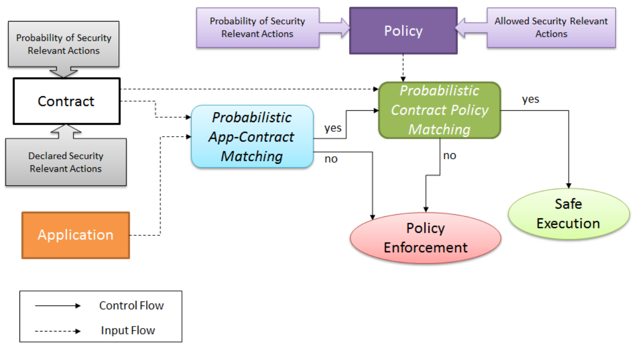
Probability of Security Relevant Actions
Probability of Security Relevant Actions
Policy
Allowed Security Relevant Actions
Contract
Probabilistic Contract Policy Matching
yes
Probabilistic App-Contract Matching
yes
Declared Security Relevant Actions
no
no
Safe Execution
Policy Enforcement
Application
Control Flow
Input Flow

Fig. 2 Security-by-Contract-with-Probability components and workflow.

performed by the app. A *policy* is a formal specification of the acceptable security relevant behavior that the app is allowed to execute on the device in which it is installed. The basic idea of the contract-based approach consists in employing the contract to verify that the security conditions imposed by the policy are actually satisfied by the app. More precisely, denoted with $\preceq$ the *compliance relation* between any pair of elements of the S×C model, the contract-based approach is based on the satisfaction of the following transitive relation:

$$A \preceq C \preceq P \Rightarrow A \preceq P \tag{1}$$

Figure 1 reports the S×C components and workflow, which can be described as follows. As a first step, it is verified whether the contract and the app match, i.e., if the contract is really representative of the app behavior ($A \preceq C$). This operation is named *App-Contract Matching* and can be based on several methodologies, depending on the adopted contract model, and spanning from proof-carrying code to the application of trust relations towards either the application developer or the certification authority issuing the contract. Assumed that app and contract are matched, S×C verifies if contract and security policy defined by the system administrator match as well ($C \preceq P$). If both contract and policy are expressed formally through compatible models, this operation, named *Contract-Policy Matching*, can employ automatic formal verification techniques, like model checking or equivalence checking. If the match is verified, the app can be executed safely, because, based on the previous checks, it is demonstrated that the app behavior respects the policy ($A \preceq P$). If either the App-Contract Matching or the Contract-Policy Matching fail, the desired relation (1) cannot be verified. In such a case, the app could be still executed. However, a *monitor* is attached to the app with the aim of checking that the app execution is matching step-by-step the specified policy. In particular, if the app is violating a policy by proposing the execution of a specific action, then the monitor *enforces* the policy by stopping the execution of such an action. On the implementation side, this enforcement brings overhead, which is completely avoided in the case of safe execution.

3.2 Probabilistic Security by Contract

The basic idea of the probabilistic security by contract (S×C×P) is to loosen the rough constraints of the standard S×C model, where a policy may only express if an action *should* or *should not* be executed. The S×C×P model extends the app contract with probabilistic information and allows for the definition of more flexible policies, where security relevant behaviors are integrated with conditions concerning the execution probability of actions. Figure 2 depicts the architecture and workflow of the S×C×P model, which extend conservatively those described in Figure 1.

On one hand, it is well-known that enriching the system verification with quantitative information allows for capturing behaviors that cannot be analyzed in a purely nondeterministic setting, thus increasing the expressive power of the detection mechanism. On the other hand, estimations of the difference between the monitored, detected behavior and the ideal one allow for refining and relaxing the classical binary approach of the possibilistic framework. These general considerations apply also to the specific setting of mobile apps, whose complex execution may require policies that are not as plain as those modeled in terms of boolean conditions only, which turn out to be too simplistic – i.e., unable to detect potential malicious behaviors – and, at the same time, too restrictive – i.e., unable to approximate in order to recognize that behaviors that are close enough to the ideal one can be permitted.

Therefore, the goal of the S×C×P framework is to provide a quantitative, conservative extension of the standard S×C approach enabling approximate security analysis. Obviously, such an extension requires the semantic redefinition of the main S×C control operations.

To this aim, we introduce two alternative behavioral models.

3.2.1 Action-based Contract

In the first approach, the model describes the probability distribution associated to the execution of the security relevant actions. By extending the nondeterministic approach, in which actions are either permitted or not by the policy, in the probabilistic setting we ideally consider any intermediate scenario between these two limiting cases. The intuitive motivation is that actions that are considered unlikely (resp., likely) from a security perspective shall be executed with low (resp., high) frequency by a compliant app. The quantitative formalization of these checks depends on factors like, e.g., the probability distribution associated to security relevant events and the tolerance thresholds that parameterize the analysis at execution time.

Informally, a probabilistic contract is a document specifying all the security relevant actions that the app can perform together with the related, expected execution probability. Thus, the probabilistic contract describes quantitatively the expected behavior for the app and represents a generalization of a standard contract. In fact, an action with null probability corresponds to an action that cannot be executed in the standard S×C model, whilst every action with a probability greater than 0 represents an enabled action for the S×C model.

With respect to the S×C model, it may be not sufficient to examine statically the control flow graph of the app to build a probabilistic contract. The execution probabilities can be either manually estimated by the entity building the contract (certification authorities or the developer), or they can be generated dynamically, i.e., by observing a sufficiently large number of execution runs of the app, e.g. in a safe environment, to obtain estimations of its probabilistic behavior at the desired level of precision. For instance, an automatic, tool-supported methodology to build probabilistic contracts from observed execution traces is presented in [6].

On the architectural side, the *Probabilistic App-Contract Matching* checks whether the real, observed behavior of the app effectively matches the contract. This is done by characterizing quantitatively the run-time app behavior and then comparing the obtained estimation with the probabilistic contract. Because of the unpredictable, nondeterministic effect of behaviors that are not dependent from the app, such a characterization represents an approximation of the quantitative behavior of the app expected by the contract, so that tolerance thresholds, determined through classical statistical inference analysis, are used to govern the comparison.

The *Probabilistic Contract-Policy Matching* checks whether the probability distribution expressed in the contract does not violate any condition specified in the policy, e.g., if no action is supposed to be more/less frequent than specific policy values. The requirements can be either strict, i.e., the probability values specified in the policies must be matched exactly, or can be approximated, by allowing a deviation depending on configurable thresholds. The rationale behind this relaxed compliance verification is again to favour approximate, flexible analysis dealing with the unpredictability of certain app executions, without compromizing the result of the security check.

Analogously, if the policy enforcement is needed, the related mechanism takes into account probabilistic information and tolerance thresholds to decide whether an action is to be stopped or not, at the desired level of approximation.

3.2.2 History-based Contract

The previous approach extends in a natural way the S×C model by adding information and conditions about the probability distribution of the actions representing the app security relevant behavior. Such an action-based extension relies on an absolute criterium abstracting from the evolution of the app execution and, therefore, the history of its behavior. However, in certain cases, the behavior of the app may change from time to time, depending on the past history of events. To take into account such a dependence, we propose also a more expressive behavioral model encoding the history of activities performed by the app. To this aim, the history-based model is not simply represented by an absolute probability distribution associated to the domain of security-relevant actions. Instead, it is given by a probabilistic labeled transition system used to model both the probabilistic behavior of the app and its contract. Analogously, a policy is not simply an expected, theoretical distribution, but it is described in terms of a logic formula expressing a property that is to be satisfied by the model. Hence, as we will see, the control operations of Eq. (1) are conducted accordingly. More precisely, the *Probabilistic App-Contract Matching* performs a similarity check between the probabilistic models describing the app behavior and the app contract, while the *Probabilistic Contract-Policy Matching* performs model checking to verify whether the contract model satisfies the formula representing the policy. Analogously, in the case of enforcement, the model checking based control is applied to the model of the app behavior against the policy formula. As in the previous approach, approximation techniques are considered to relax the conditions of each control operation.

To illustrate the difference between the two behavioral models, we show their graphical interpretation in Figure 3. On the left side, we observe the action-based model of a contract including four security relevant events, each one equipped with the related execution probability. A policy in such a model represents a constraint over these probability values, as shown in the illustrating example, describing a condition stating that the (absolute) execution probability of the event d must be less than 0.1. Notice that the action-based model abstracts from the history of events characterizing the app behavior. On the right side, we observe the history-based specification of a contract, which is given by a probabilistic state-transition model where the same security relevant events of the previous example are represented by certain transitions. Notice that the model describes the (branching) structure of the potential app behaviors and is probabilistic, as the choice among alternative transitions is probabilistic and governed by the normalized weights labeling the transitions. In such a model, a policy is a probabilistic temporal logic formula. For instance, the policy shown in the example states that the probability of reaching a state enabling the event d must be less than 0.1.

4 Probabilistic compliance

This section provides a formal description of the three operations of the S×C×P framework, namely $A \preceq C$, $C \preceq P$ and $A \preceq P$, in the probabilistic setting, both for the action-based and history-based approaches.

4.1 Action-based Approach

We start by presenting a simple, action-based model for the S×C×P framework with the aim of formalizing the notion of (approximated) probabilistic compliance and, as a consequence, extending in a quantitative setting the purely functional relation $\preceq$.

As explained in the previous section, the app contract C is equipped with a model of the quantitative expected behavior of the app A. Formally, such a model is given in the form of a theoretical estimated (sub-)probability distribution π associated to the domain of (relevant) actions Act. More precisely, function π, defined from a nonempty, at most countable set Act to $\mathbb{R}_{[0,1]}$, is a discrete probability distribution over Act if $\sum_{a \in Act} \pi(a) = 1$. Intuitively, the contract C specifies that at each step of the app run the action $a \in Act$ is chosen with probability $\pi(a)$. Notice that, considering sub-probability distributions (such that, i.e., the summation above is ≤ 1) permits to ignore irrelevant actions, which are excluded from Act but have non-zero probability. Even in such a relaxed setting, the following theory can be applied with no restrictions.

The policy P is described in terms of quantitative conditions over the execution frequency associated to the allowed security relevant actions. For instance, "the probability of action a must be between 0 and 0.1" could be one such requirements, which, generally speaking, are defined to restrict the values that certain execution frequencies can assume at run time. Formally, the specification of P is represented by a constraint satisfaction problem, see, e.g., [38] for the mathematical details.

In the following, we show how to employ the quantitative models surveyed above to relate app behavior, contract, and policy. Such relations are used to establish the probabilistic application compliance.

4.1.1 Estimating $A \preceq C$

Firstly, the compliance of the app with respect to the contract ($A \preceq C$ in the non-probabilistic case) cannot be checked statically through the application of some metric. In fact, we need to evaluate the quantitative run-time behavior of A. To this aim, notice that the sequence of actions observed during execution defines an actual discrete probability distribution. Such a quantitative characterization of the observed sequence is expected to represent an experimental approximation of π, with a precision that depends on the run length and other parameters, like, e.g., the estimated level of confidence. Hence, the aim is to compare, at each trial of the execution run, the actual distribution observed at run time against the theoretical distribution π, in order to verify whether the app behavior is compliant with the contract from a quantitative perspective.

In the following, we show how to apply classical statistical inference analysis to the estimation of such a compliance check directly at run time. The reader interested in the mathematical background can refer to [27, 31]. We first recall that, by the Central Limit Theorem, in the long run the actual monitored behavior of the app shall estimate correctly the quantitative expected behavior described by π. However, if the app does not respect the contract, such a convergence will not be achieved and the challenge is to realize as soon as possible that the compliance cannot be satisfied.

The core idea is that for each action $a \in Act$, we consider a Bernoulli trials process with estimated probability $\pi(a)$ for success on each trial, and such that

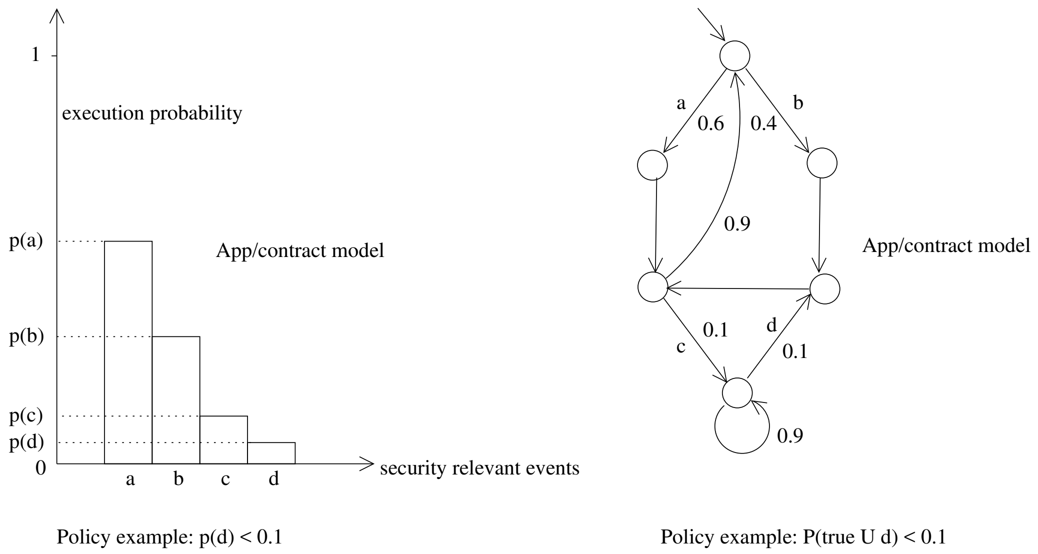

Fig. 3 Behavioral models comparison.

$X_i = 1$ if the i-th outcome is actually a success, and $X_i = 0$ otherwise. Therefore, the actual estimation for action a after a trials run of length n is given by:

$$\rho_n(a) = \frac{\sum_{i=1}^{n} X_i}{n}.$$

By virtue of the Central Limit Theorem, $\rho_n(a)$ is approximately normally distributed with mean $\pi(a)$ and standard deviation:

$$sd_n(a) = \sqrt{\frac{\pi(a)(1-\pi(a))}{n}}.$$

Hence, it is possible to employ the classical confidence levels estimated for the standard normal distribution. In particular, we recall that the standard scores (called Z-scores) of the standard normal distribution are defined as:

$$Z = (X - \mu)/\sigma \qquad (2)$$

where X is the approximating random variable, μ is its mean, and σ is its standard deviation. In our framework, they are represented by $\rho_n(a)$, $\pi(a)$, and $sd_n(a)$, respectively. Therefore, since the Z-scores are known from the standard normal distribution table, it is worth expressing (2) as $E = Z \cdot \sigma$, where the error E represents $\mid X - \mu \mid$. In our framework, the error, which depends on the action a, the confidence level l, and the length n of the execution run, can be expressed as: $E_n^l(a) = Z \cdot sd_n(a)$. Then, since the aim is to verify whether the actual estimate $\rho_n(a)$ measured at run time is compliant with the estimated probability $\pi(a)$, we evaluate the disequality:

$$|\,\rho_n(a) - \pi(a)\,| \leq E_n^l(a). \qquad (3)$$

Notice that the tolerance factor is inversely proportional to the length of the execution run and depends on the desired confidence level.

Example 1 Let us compute the maximum expected error E for $\pi(a) = 0.2$ and level of confidence equal to 95% (i.e., $Z = 1.9599$). Hence, for n trials we have the following error estimations:

n	E
1	0.7839
5	0.3505
10	0.2479
100	0.0783

Now assume, for instance, to observe an experimental run of length 100, during which the action a has been observed 11 times. Then $0.2 - 0.11 = 0.09 > 0.0783$, from which we can deduce with a 95% confidence level that the probabilistic observed behavior is not fair with respect to the theoretical estimated probability. In fact, the 95% based confidence interval means that if we run the experiment several times, in 95% of them the obtained confidence interval shall contain the theoretical estimated probability.

Definition 1 Given the confidence level l, A matches C after n execution steps ($A \preceq_n^l C$) if for all $a \in Act$ it holds that (3) is satisfied.

The above definition can be extended to apply the compliance check at each step of an execution run of length m.

Definition 2 Given the confidence level l and the execution run ν of length m, A matches C on ν if for all $1 \leq n \leq m$ it holds that $A \preceq_n^l C$.

Hence, contract compliance of the observed behavior of the app (represented by a sequence ν of actions) can be verified by checking whether A matches C on ν.

With respect to the proposed approach, there is some analogy with formal methods applied to the comparison of the quantitative behavior of systems, like, e.g., in the formal setting of process algebra and behavioral equivalences [10]. The system behaviors associated to two probability distributions π and ρ_n can be described, e.g., by the probabilistic process algebraic terms:

$$S \triangleq \sum_{a \in Act} a[\pi(a)].S \qquad \text{and} \qquad O_n \triangleq \sum_{a \in Act} a[\rho_n(a)].O_n$$

representing the theoretical estimated *specification* of the system and the experimental run-time *observation*, respectively. In both cases, the summations express the external choice among the several different events, which is governed by a probability distribution. The notation $a[p].P$ intuitively means that a is executed with probability p after which the process behaves as P. The semantics of the two process terms are given by probabilistic labeled transition systems with a unique state and a self-loop for each $a \in Act$ labeled with the action name a and the probability $\pi(a)$ (resp., $\rho_n(a)$). These systems obey the generative model of probabilities and can be compared with each other through an approximated variant of the classical bisimulation based equivalence check, which we refer as ϵ-bisimulation. By following such an approach, see, e.g., [3, 4, 17], it turns out that the two systems behave approximately the same up to a tolerance threshold computed as $\epsilon = \max_{a \in Act}(\,|\,\pi(a) - \rho_n(a)\,|\,)$, similarly as demonstrated through the statistical inference check. However, differently from these approaches, which limit their contribution to the computation of such an ϵ, in the setting of statistical inference analysis ϵ is given a reliability interpretation with respect to the desired confidence level. This interpretation is necessary to trade n (the length of the experiment), the error estimation ϵ, and the risk-related probability (provided by the confidence level) of guessing a (statistically significant) difference between the estimated behavior and the monitored behavior.

4.1.2 Estimating $C \preceq P$

The probabilistic compliance between contract C and policy P can be easily checked by observing that π represents an evaluation for the constraint satisfaction problem modeling the policy. We first notice that, by definition, π is complete as it includes all the variables of the constraint satisfaction problem. Hence, it is sufficient to check whether π is also consistent, i.e., no constraints are violated by π. Indeed, we recall that an evaluation represents a solution for the problem if it is both consistent and complete [38].

Definition 3 We say that C matches P (written $C \preceq P$) if π is a solution of the constraint satisfaction problem modeling P.

In order to relax the definition above, we add a tolerance threshold to the compliance check. Informally, C approximates P if π is similar to a distribution that solves the constraint satisfaction problem modeling P. In analogy with the similarity checks of the previous section, we use the Chebyshev distance metric [13] to compare two distributions π and π':

$$d(\pi, \pi') = \max_{a \in Act}(\,|\,\pi(a) - \pi'(a)\,|\,)$$

based on which we provide the following approximated version of Def. 3.

Definition 4 C matches P up to ε ($C \preceq_\varepsilon P$) if there exists a solution π' of the constraint satisfaction problem modeling P such that $d(\pi, \pi') \leq \varepsilon$.

4.1.3 Estimating $A \preceq P$

Relation $\preceq_\varepsilon$ can be applied also to relate A and P, in which case the distribution to consider is ρ_n instead of π. With this consideration in view, the following compliance result between A and P can be derived by combining linearly the two checks based on relations $\preceq_n^l$ and $\preceq$ previously discussed.

Theorem 1 *If $A \preceq_n^l C$, with $\varepsilon = \max_{a \in Act} E_n^l(a)$, and $C \preceq P$, then it holds that $A \preceq_\varepsilon P$ with confidence level l.*

The proof is a straightforward consequence of Defs. 1 and 3. The interpretation is as follows. Assume that the monitored behavior of the app statistically matches the contract with respect to the desired confidence level, and that the contract matches the quantitative requirements of the policy. Then, the app satisfies the policy with a margin dependent on the statistical error.

By replacing $C \preceq P$ with $C \preceq_\varepsilon P$ we obtain an extension of the previous result showing that the level of approximation relating A and P is the sum of the margin errors introduced by each of the two compliance checks.

Theorem 2 *If $A \preceq_n^l C$, with $\varepsilon_1 = \max_{a \in Act} E_n^l(a)$, and $C \preceq_{\varepsilon_2} P$, then $A \preceq_{\varepsilon_1 + \varepsilon_2} P$ with confidence level l.*

4.1.4 Enforcing $A \preceq P$

It may be that the contract C does not match the policy P neither precisely nor approximately (see Defs. 3 and 4). Therefore, it would not be meaningful to check the compliance of the app A with respect to the contract in order to estimate $A \preceq P$. Analogously, it is not possible to derive transitively the compliance of A with respect to P whenever A does not match the contract (i.e., the hypothesis $A \preceq_n^l C$ of the previous theorems is not satisfied). In such cases, A can be still executed and checked step-by-step against the policy P, possibly enabling the enforcement mechanism every time the app does not respect the conditions of the policy, up to the desired tolerance. Formally, the compliance of A with respect to P is defined as a variant of Def. 1 in which we replace the distribution modeling the contract with a distribution solving the constraint satisfaction problem modeling P.

Definition 5 Given the confidence level l, A matches P after n execution steps ($A \preceq_n^l P$) if there exists a solution π of the constraint satisfaction problem modeling P such that for all $a \in Act$ it holds that:

$$|\,\rho_n(a) - \pi(a)\,| \leq E_n^l(a).$$

We can argue similarly to define the corresponding variant of Def. 2. In practice, the check above is to be satisfied at each step n of the execution run, otherwise the enforcement mechanism is activated.

4.2 History-based Approach

As discussed above, in order to refine the action-based approach, it is worth considering the history of events and security conditions based on such histories. To this aim, the history-based contract and policies must be able to model complex behaviors and conditions, which consider sequences of actions, their branching structure, and the probability of each sequence. To reach such a level of expressiveness, we rely on probabilistic labeled transition systems to represent contract and app behavior and on probabilistic temporal logics to specify policies. We start by defining the following functional model.

Definition 6 Let AP be a set of atomic propositions. A *labeled multidigraph of states* (LMS) is a tuple (V, I, E, s, t, L), where V is the finite set of states, $I \subseteq V$ is the set of initial states, E is the finite set of edges, $s : E \to V$ is a mapping indicating the source state of each edge, $t : E \to V$ is a mapping indicating the target state of each edge, and $L : V \to 2^{AP}$ is a function relating each state to the set of atomic propositions that are true in the state.

In our framework, AP represents a set of boolean conditions identifying the status of specific device's dynamic features, like, e.g., the fact that the device screen is either ON (true, 1) or OFF (false, 0). Assuming that the status of the device is defined by n boolean variables, $L(i)$ for a state i can be given as a n-length vector of boolean values. Hence, a transition from state i to another state j, defined as a directed edge on the LMS from i to j, represents an event (caused either by app operations, or system and user behaviors) that implies the flipping of some boolean value. The atomic propositions can be global, i.e., related to the user, device or OS activities, and app-specific, i.e., related to the behavior of the specifically monitored app. Hence, a LMS with the same set of states can be instantiated for every monitored app.

The LMS modeling the behavior of an app is generated by observing the sequence of events at execution time, as shown in Figure 4. On the left, we have a list of updates of the vector of boolean values expressing the device status ($n = 3$) and caused by the monitored activities. On the right, we have the corresponding LMS, where each state is associated to a different vector and each edge refers to a vector update. For the sake of readability, the edges are numbered to emphasize the execution order of the events. As shown, it is possible to have multiple edges insisting on the same state transition (3,5 and 4,6), and self-loops, i.e., edges having the same state as source and target (2).

Multiplicities of edges are exploited to turn the LMS into a probabilistic labeled transition system containing the same states of the LMS and transitions deriving from the edges of the LMS.

Definition 7 Let AP be a set of atomic propositions. A *probabilistic labeled transition system* (PLTS) is a tuple (V, I, P, L), where V is the finite set of states, $I \subseteq V$ is the set of initial states, $P : V \times V \to [0, 1]$ is the probabilistic transition function satisfying $\forall i \in V : \sum_{j \in V} P(i, j) = 1$, and $L : V \to 2^{AP}$ is a function relating each state to the set of atomic propositions that are true in the state.

From the LMS (V, I, E, s, t, L), the corresponding PLTS (V, I, P, L) is derived by defining P as follows:

$$P(i,j) = \begin{cases} \frac{mul(i,j)}{mul(i)} & \text{if } \exists e \in E : s(e) = i \wedge t(e) = j \\ 0 & \text{otherwise} \end{cases}$$

where $mul(i, j)$ is the multiplicity of the edges from i to j in E and $mul(i)$ is the number of outgoing edges from i in E. Notice that such a transformation induces

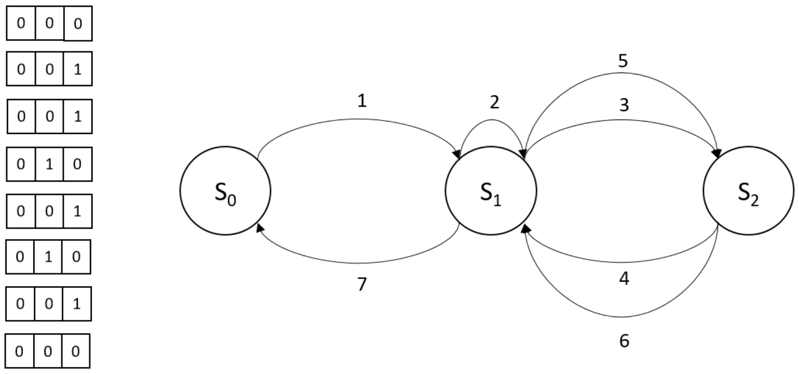

Fig. 4 Example of an execution trace and the related LMS.

a probability distribution for each state, as required by Def. 7.

As an example, consider Figure 5. On the up side, we have a LMS similar to that described in Figure 4 and, on the down side, we have the corresponding PLTS, which is computed by applying the transformation above.

For the verification of PLTSs, we rely on model checking of probabilistic temporal logic formulas [14, 30]. To this aim, since we are interested in modeling (probabilistic) history based policies, we use the logic PCTL to express policy conditions. The syntax of PCTL is as follows:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{P}_{\bowtie p}(\psi) \\ \psi &::= \Phi\, U\, \Phi \mid \Phi\, U^{\leq k}\, \Phi \end{aligned}$$

where $a \in AP$, $p \in [0,1]$, $\bowtie$ stands for usual comparison arithmetic operators, and k is a natural number. Informally, a state i satisfies the atomic formula a if $a \in L(i)$. The semantics of the propositional logic fragment of PCTL derives directly from CTL semantics, while the probabilistic operator $\mathcal{P}$ expresses that the probability that paths fulfill the path formula ψ is $\bowtie p$. The classical until path formula $\Phi\, U\, \Phi'$, which can be indexed by k, expresses that Φ holds along the path until Φ' holds (within k steps in the step-bounded variant). For a comprehensive explanation of the formal semantics and of the model checking algorithms, the interested reader is referred to, e.g., [9]. In our framework, it is worth specifying that a PLTS (V, I, P, L) satisfies a formula Φ if every state in I satisfies Φ.

In the following, we show how to employ these models to relate app behavior, contract, and policy similarly as done for the action-based approach.

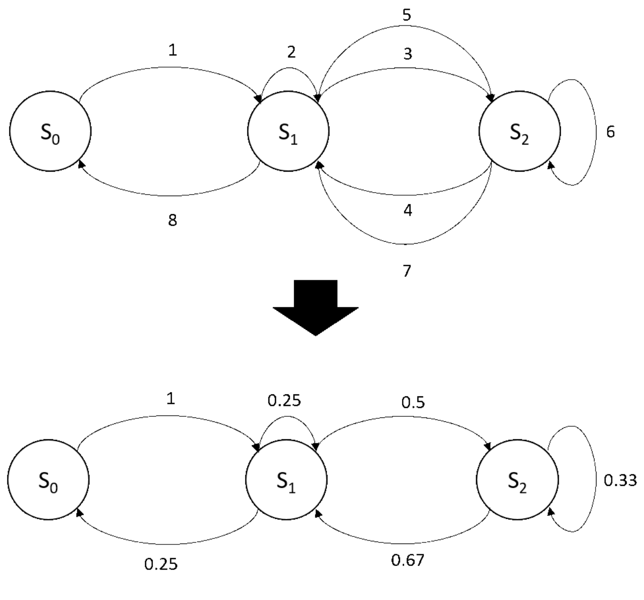

Fig. 5 Example of generation of a PLTS from a LMS.

4.2.1 Estimating $A \preceq C$

Checking the compliance of the behavior of an app A with respect to the related contract C, formally described as labeled transition systems, is not a novelty in the literature. For instance, in the probabilistic setting, an approach based on probabilistic labeled transition systems and on statistical tests is proposed in [5]. The underlying idea is similar to that exposed in the action-based setting and consists in comparing a certain probability distribution of the execution traces extracted from the probabilistic labeled transition system modeling the contract and an analogous probability distribution extracted from the probabilistic labeled transition system that is generated on-the-fly at the app run-time. The comparison is estimated by means of similarity or distance metrics [13].

By employing classical formal methods, the compliance between A and C can be also evaluated through bisimulation-based equivalence checking, which can be generalized to the use of ϵ-bisimulation in the case of approximate verification.

These approaches, which can be applied in our framework as they are, turn out to be feasible provided that whenever the app behavior is monitored, a formal specification of the contract is available. On the other hand, the compliance is ensured by construction if the contract is generated through direct observations of the app execution.

4.2.2 Estimating $C \preceq P$

The probabilistic compliance between a contract C modeled as a probabilistic labeled transition system and a policy P modeled as a PCTL formula is verified by model checking C with respect to P. More precisely, $C \preceq P$ holds if the PLTS expressing C satisfies the PCTL formula expressing P.

In order to relax quantitatively the analysis whenever the check is negative, it is possible to introduce a tolerance threshold to the verification of conditions of the form $\bowtie p$ occurring in the probabilistic operator of PCTL.

Definition 8 Given the policy P given as the PCTL formula $\mathcal{P}_{\bowtie p}\psi$ and the tolerance threshold $\epsilon \in [0, 1]$, it holds that C matches P up to ϵ ($C \preceq_\epsilon P$) if C satisfies $\mathcal{P}_{\bowtie r}\psi$, where $|r - p| \leq \epsilon$.

4.2.3 Estimating $A \preceq P$

The transitive compliance of A with respect to P via C in the non-trivial case in which A and C are different, can be inferred depending on the techniques used to check $A \preceq C$ and $C \preceq P$. In particular, since $C \preceq P$ is estimated through PCTL based model checking, it is necessary to base the verification of $A \preceq C$ on comparison techniques preserving satisfiability of PCTL formulas. This is the case if the compliance between A and C is conducted by using the probabilistic bisimulation semantics, which coincides with PCTL based equivalence [9]. Analogous considerations hold in the approximate setting of ϵ-bisimulation and related pseudometrics.

On the other hand, if $A \preceq C$ is based on the statistical tests of [5], depending on the distance metric adopted, the measure of the similarity between A and C provides just a rough estimation of the tolerance threshold that allows A to approximately satisfy P. In order to overcome such a limitation, in the next section we discuss an alternative approach based on model checking and on the same statistical analysis of Section 4.1 that, moreover, turns out to be more efficient than the equivalence-based methods.

4.2.4 Enforcing $A \preceq P$

The enforcement of $A \preceq P$ may be necessary for several reasons, either by virtue of the considerations of the previous section, or, e.g., because the relation $A \preceq P$ cannot be deduced transitively through C for the same motivations surveyed in Section 4.1.4. In these situations, in order to ensure a secure execution of the app, it is worth enforcing $A \preceq P$ at run-time. To this aim, in this section we propose a naive approach inspired by the statistical check presented in Section 4.1.

As discussed in Section 4.2.1, the probabilistic labeled transition system describing the app behavior can be generated and updated on-the-fly during the app execution. Then, the goal is to verify at each step if such a model meets the policy, up to some tolerance in the case of probabilistic conditions. More precisely, we assume to deal with probabilistic policies represented by formulas of the form $\mathcal{P}_{=p}\psi$, where ψ is a path condition over the interval $[t_s; t_e]$. As an example, in the trivial case in which the unique occurrence of the until operator in ψ is of the form $\Phi\, U^{\leq k}\, \Phi$, then the considered interval is $[0; k]$.

Starting from the beginning of the app execution, the PLTS is updated step by step, and in the interval $[t_s; t_e]$ it is model checked against $\mathcal{P}_{=?}\psi$ in order to estimate the actual probability r associated with paths satisfying the property ψ [30]. By rephrasing the statistical approach of Section 4.1, the theoretical parameter p assumed by the policy and the actual parameter r obtained through model checking are compared, similarly as done in the case of Eq. 3 and related definitions. In particular, assuming that r is estimated at the step $t_i \in [t_s; t_e]$, and given $n = t_i - t_s + 1$ and the confidence level l, the error expressing the tolerated difference between p and r is given by:

$$E_n^l = Z \cdot \sqrt{\frac{p(1-p)}{n}}$$

and, therefore, the expected condition to meet is:

$$|r - p| \leq E_n^l.$$

Hence, as the execution proceeds the tolerance diminishes and r must converge to p. Based on such an estimation of the tolerance threshold, we have the following notion of compliance.

Definition 9 Given the confidence level l and the policy P represented by the PCTL formula $\mathcal{P}_{=p}\psi$, with ψ defined over the interval $[t_s; t_e]$, A matches P after $t_i \in [t_s; t_e]$ execution steps, with $n = t_i - t_s + 1$, denoted $A \preceq_n^l P$, if the PLTS generated after t_i observations of the behavior of A satisfies $\mathcal{P}_{=r}\psi$ and:

$$|r - p| \leq E_n^l.$$

The check above is tested at each step of the interval of interest, and whenever it is not met by the app the enforcement mechanism is activated. An analogous verification is straightforwardly applied to the general case of probabilistic formulas of the form $\mathcal{P}_{\bowtie p}\psi$. In such a case, the check $\mid r - p \mid \leq E_n^l$ is to be verified only whenever $r \bowtie p$ is not satisfied.

5 Enforcement Architecture

In this section, we describe the implemented architecture that can be exploited for both dynamic contract generation and policy enforcement. The proposed implementation has been designed for Android systems. Thus, the S×C×P framework comes as an Android app, which is able to perform the operations previously discussed, directly on the device. Henceforth, the app will be referred as S×C×P-App.

The S×C×P-App is able to control actions performed by any other app on the device, together with a set of global actions performed by the operating system, such as controlling the light of the screen. The control of an action is performed by means of the *hooking* operation, i.e., putting a callback on the method invocation related to the action that has to be controlled. Once hooked, it is possible to control a method, by performing actions *before* it is executed or immediately *after*. To this end, the proposed system exploits the Xposed-Framework [1], a toolkit for methods hooking available for Android. The Xposed Framework (or simply Xposed) is an advanced custom developer tool designed to give a much greater control on the Android system and on the running apps, compared to the one granted by the Android available APIs. The Xposed Framework can be installed on any Android device and release, however it requires the target device to be *rooted* (jailbroken), which may constitute a limitation for distribution. However, we argue that the S×C×P framework is not designed, for now, to be largely distributed among average users. Instead, it has to be considered as an advanced research tool for policy enforcement, which is ready to be integrated in specific environments where a single entity controls several devices (e.g., as in the case of a company that gives a business mobile phone to the employees).

The S×C×P-App can be configured to work in two modes: (i) *Logging Mode*, which records every time an hooked action is invoked, and (ii) *Enforcement Mode*, which verifies if the hooked action is compliant with the policy when it is invoked, by stopping the behavior (*enforcement-by-truncation*) or modifying the outcome (*enforcement-by-obligation*) if the policy is not matched.

The enforcement of history-based policies puts additional challenges, since the probabilities of the current execution traces have to be matched continuously with the policy. This task is demanded, as for the contract-policy verification, to the evaluation engine of the model checker PRISM, which is part of the S×C×P-App. However, the execution to be checked cannot be recorded directly by the S×C×P-App, since the Xposed framework hooks actions directly from the hijacked app, being thus able to see only its memory space, not shared with the one of the S×C×P-App. To reduce the performance overhead that would be caused by writing in the external memory, and to overcome the issues related to missing permissions, the S×C×P-App and the hijacked apps communicate by means of the *shared-preferences* buffer. A schematic representation of the implementation is presented in Figure 6.

Whenever a new app is installed on the device, the S×C×P-App intercepts the event and checks if a contract is available for it. The contract can be provided as an extension of the original `AndroidManifest.xml` file, or can be made available through an external repository. In the following, for the sake of simplicity we assume that the probabilistic App-Contract Matching is the only task that is not performed directly on the device at execution time. Such a validity check is, in fact, proven before the app execution on the mobile device. To this end, it is possible to assume that all the related operations, which must be conducted dynamically, are performed by running the app in a protected environment (such as a sandbox). As an alternative, contract and compliance check can be provided directly by the developer in such a way that the App-Contract Matching reduces to be an operation based on trust, using for example the same model proposed in [15].

The actions controlled by the S×C×P-App are:

- **Outgoing SMS Messages**: functions related to sending text messages, controlled to verify if the message is directed toward a number that is present in the contact list, or toward an unknown number.
- **Activity.onResume/Activity.onPause**: functions related to the execution of the application either in foreground or in background.
- **Opening Contact List**: functions related to the opening and selection of contacts from the contact list.
- **HTTP connections**: functions related to the opening of connections toward external servers.
- **WebView opening**: functions related to the creation of a `WebView` to show an online content.
- **User Present and Screen On/Off**: functions related to the interactions of the user with the device.

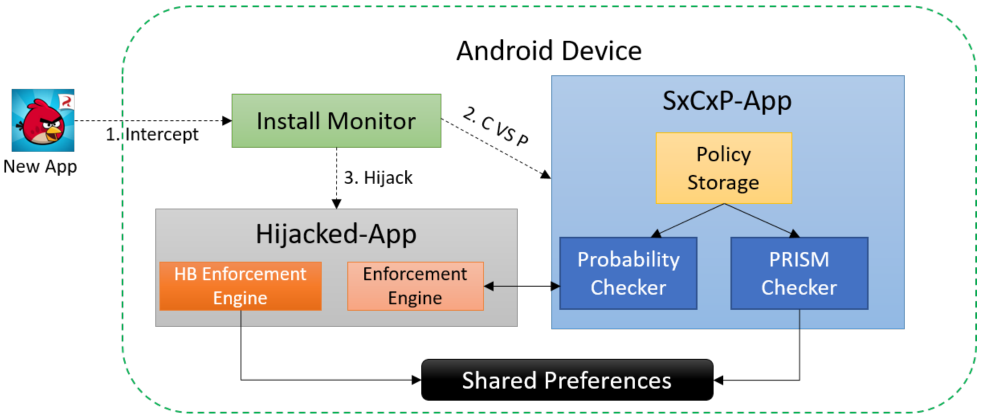

Fig. 6 Implementation of the SxCxP in real devices.

6 Experimental Results

In this section, we show the application of the S×C×P framework to the analysis of the security critical functionalities of several Android apps. In particular, we consider its effectiveness against 270 malware samples divided into 8 malware families belonging to the Spyware and SMS-Trojan classes. These two malware classes include the most dangerous malware samples, since their malicious behavior affects both user money and privacy. We check also the transparency of the S×C×P framework with respect to the execution of genuine apps that do not include any malicious behavior. The verification is conducted through the action-based and the history-based approaches in order to illustrate their different expressiveness and overhead.

6.1 Action-based verification

6.1.1 Contract generation and $A \preceq C$ compliance

The contract for each malware sample is extracted dynamically, by executing the software in a protected sandbox, i.e., a controlled Android emulator, where the app could not cause any real damage. To this end, the logging mode of the S×C×P-App has been used, by collecting 10 traces of variable length for each sample, which are then used to generate the app contract. The used principle is that, with respect to the estimation of $A \preceq C$ (see Section 4.1.1), contract compliance of the observed app behavior derives by construction of the contract and is guaranteed with a 95% confidence level.

The same approach is used to generate the contract for a set of genuine apps downloaded from Google Play, the official Android market.

6.1.2 Checking $C \preceq P$

By following Def. 3, the contract is matched against a set of probabilistic policies with configurable probabilities to control malicious text messages and undesired data connections. The policies are as follows:

- **Policy 1** The app is allowed to send a SMS message with probability not greater than p.
- **Policy 2** The app is allowed to perform a http connection with probability not greater than p.
- **Policy 3** The app is allowed to send a SMS message toward a number that is not in the contact list with a probability not greater than p.
- **Policy 4** The app is allowed to perform a http connection toward a numerical IP address (i.e., not toward a domain name) with probability not greater than p.

Policies 1 and 3 are used to control outgoing text messages with different specifications about the recipient number. By comparing the two policies, notice that Policy 1 is more adequate to control apps that are not supposed to send text messages to any contact, while Policy 3 is more suitable to control the behavior of apps that can use text messages, such as messaging apps, or SMS managers. However, in the normal usage of such a kind of apps it is much more likely that a text message is sent to a known number, i.e., in the contact list, rather than to an unknown one. Still, through the usage of probabilistic policies it is possible to allow unlikely operations, by tuning the related probability execution. Analogously, Policies 2 and 4 are designed to control the outgoing data traffic, by monitoring the connections opened by specific apps. In particular, notice that opening a connection toward an IP address in the inet format, i.e., not toward a domain name, is typical of malware sending information to servers controlled by attackers (spyware). Thus, with respect to Policy 2, Policy 4 performs a fine-grained control on outgoing traffic, filtering only the suspicious one.

The results of the $C \preceq P$ check against the 270 samples belonging to the 8 malware families are reported in Table 1. All malware samples violate one or more of the four security policies. Each SMS Trojan, but `DogoWar`, violates both Policies 1 and 3. Spyware apps, instead, do not satisfy Policy 2 since they do not perform any connection to IP-based addresses. Hence, all analyzed spyware samples send traffic directly to domains controlled by the attackers. The severity of the violation is estimated by tuning the policy configuration parameter p. The threshold column of Table 1 reports the highest value of p that causes the contract-policy mismatch. The trace length column of Table 1 reports the average global number of critical operations collected per trace in order to build the sample contract. As shown, some apps have a limited trace length, in particular `FakePlayer` and `Kmin`. This is representative of the reduced interactions of some malicious apps, which do not trigger any of the critical actions monitored, except for those revealing the malicious behavior.

Similarly as done for the previous experiment, Table 2 reports the results for $C \preceq P$ compliance for the set of genuine apps. The majority of the tested apps have a contract that does not match Policy 2, which is related to data usage. However, the use of thresholds here clarifies the additional contribution of the quantitative approach with respect to the nondeterministic one. Indeed, by tuning the parameters it is possible to relax (or to restrict) the conditions of the compliance check, thus enabling the desired level of flexibility.

By analyzing the threshold p, it is worth noticing the strong network interactions of games like `CrossyRoad`, which is supposed to be a offline videogame (i.e., working even if the device is not connected to the network), but, in spite of this, spends about 50% of the monitored actions in performing network activities. Even if it is not by itself a security criticality, the S×C×P framework could also be exploited to enforce usage policies that are able to improve the user experience, reducing the overhead generated by apps. Another interesting case is represented by `Whatsapp`, which sends a single text message at the very first start-up to register the user. Then, the overall relative probability of such a security relevant event is negligible, thus making the app contract compliant with respect to any of the considered probabilistic policies, except for the case in which no SMS is allowed at all. In a purely functional setting, `Whatsapp` does not pass the compliance test of the standard S×C control if outgoing SMS messages are not allowed by the policy. On the other hand, the S×C check is not meaningful at all if outgoing messages are allowed by the policy.

Finally, observing the trace length column, it is possible to notice the difference in the amount of actions with respect to Table 1. Genuine and popular apps are highly interactive, especially for games. Still, controlling the number of actions is not enough to differentiate between malicious apps and genuine ones, since malicious code can also hide behind popular apps that are repackaged [6].

6.1.3 Enforcing $A \preceq P$

The apps that do not satisfy a policy can be either removed or executed under the control of the enforcer described in the previous section. Table 3 reports the results concerning the enforcement of the four policies on the monitored apps, by applying the method of Section 4.1.4 (app-policy matching is based on 95% confidence level) on a set of traces different from those used to generate the contracts. Both malware and genuine apps are considered, in order to estimate the effectiveness of the enforcement in the former case and its transparency with respect to the app execution in the latter case. As in the previous tables, the probabilistic parameter of Table 3 represents the highest value causing the enforcement of the policy (N.E. stands for not enforced).

For the malicious apps, the policies that are enforced are representatives of the malware type and are fully consistent with the numerical results of Table 1, thus revealing the adequacy of the statistical approach of Section 4.1 followed to estimate the compliance check. Once more, notice that the malware `OpFake` violates three policies: the two SMS related policies and Policy 2 about http connections. In fact, `OpFake` shows an hybrid behavior, by sending at first SMS messages for registration to premium services, and then by sending device information, such as the IMEI code, to a domain controlled by the attacker. Both malicious behaviors are detected and trigger the enforcement.

Analogous comments hold in the case of the genuine apps, for which we point out that the enforcement does not negatively affect the user experience, still preserving all the desired app functionalities. The reason is that several genuine apps mainly use http connections to provide in-app advertisement, which is not effectively necessary to the correct execution of the app, providing instead an undesired overhead for the user. Hence, in such cases, the policy enforcement represents a way to keep under control such an overhead.

Notice that the major degree of severity of `WhatsApp` enforcement with respect to the $C \preceq P$ verification de-

Table 1 Action-based verification: Contract-Policy matching for malware families.

Family Name	Class	Samples	Violated Policy	Threshold	Trace Length
`DogoWar`	SMS-Trojan	7	1	50%	20
`FakePlayer`	SMS-Trojan	17	1,3	50%	17
`HippoSMS`	SMS-Trojan	15	1,3	10%	285
`Kmin`	Spyware	52	2	20%	18
`Mania`	SMS-Trojan	12	1,3	30%	27
`OpFake`	SMSTrojan/Spyware	14	1,3 - 2	30% - 10%	20
`Raden`	SMS-Trojan	19	1,3	10%	41
`Vidro`	Spwyare	39	2	35%	41

Table 2 Action-based verification: Contract-Policy matching for genuine apps.

App Name	Type	Violated Policy	Threshold	Trace Length
`AngryBirds`	Game	2	30%	322
`Browser`	Web Browser	2,4	40%	316
`CrossyRoad`	Game	2	50%	251
`CustomSMSManager`	Communication	1,3	30%	156
`Firefox`	Web Browser	2,4	40%	327
`Flow`	Game	2	20%	158
`FruitNinja`	Game	2	10%	537
`Google Earth`	Navigation	2	50%	417
`Google Maps`	Navigation	2	50%	409
`KingCalculator`	Calculator	2	10%	144
`SMS Manager`	Communication	1,3	20%	179
`WhatsApp`	Communication	1,3 - 2	< 1% - 40%	242
`YouTube`	Streaming	2	50%	273

pends on the fact that the behavior violating the policies occurs immediately at the beginning of the app execution, thus altering significantly the probabilistic security critical behavior of the app. However, the result is still acceptable and allows `WhatsApp` to be executed without restrictions by applying threshold values that, on the other hand, are sufficient to detect any kind of malware. In general, the enforcement of genuine apps is less restrictive than the $C \preceq P$ verification and can be kept under control by acting on the parameters.

6.2 History-based verification

We now investigate policies that cannot be expressed through the action-based approach. Formally, by following Def. 4.2.2, the PLTS expressing the contract is matched against a set of parameterized PCTL formulas of the form $\mathcal{P}_{\bowtie p}\psi$, where ψ formalizes the following reachability conditions:

- **Policy SMS-H** The app is allowed to send a SMS message in an interval $[t_s, t_e]$ without having selected the message recipient from the contact list in the last d steps, with a probability not greater than p.
- **Policy HTTP-H** The app is allowed to perform a http connection in an interval $[t_s, t_e]$ without having opened a `webView` in the last d steps, with a probability not greater than p.

The aim of these history-based policies is to refine Policies 1-4 of the action-based verification. Indeed, as shown in Figure 7, these policies consider two related events, e_1 and e_2, such that e_2 occurs in the interval $[t_s, t_e]$ and must be preceded by the occurrence of e_1, such that the number of events (steps) separating the two events should not be larger than d. The use of a temporal window parameterized by d is necessary to avoid relationships between events that are too much distant in time. The role of the probabilistic parameter p is to introduce a tolerance factor that is necessary to relax conditions that, for real-world apps, could be too precise, restrictive, and hard to satisfy.

Policy SMS-H states that the app is allowed to send text messages mainly as a consequence of an active user action, i.e., the explicit opening of the contact list to select the message recipient. In fact, the policy specifies that an app has a low probability of sending directly an SMS message without allowing the user to choose the recipient from the contact list. Notice that Policy 3 was

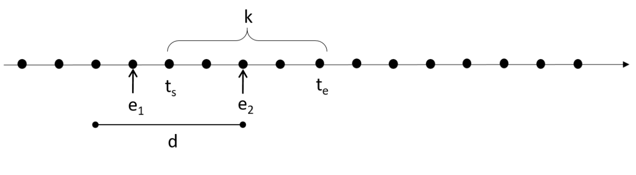

Fig. 7 Example of timeline for the two history-based policies.

Table 3 Action-based verification: Enforcement results.

App Name	Type	Policy 1	Policy 3	Policy 2	Policy 4
`DogoWar`	Malicious	30%	N.E.	N.E.	N.E.
`FakePlayer`	Malicious	50%	50%	N.E.	N.E.
`HippoSMS`	Malicious	10%	10%	N.E.	N.E.
`Kmin`	Malicious	N.E.	N.E.	20%	N.E.
`Mania`	Malicious	30%	30%	N.E.	N.E.
`OpFake`	Malicious	20%	20%	10%	N.E.
`Raden`	Malicious	20%	20%	N.E.	N.E.
`Vidro`	Malicious	N.E.	N.E.	30%	N.E.
`Angry Birds`	Genuine	N.E.	N.E.	10%	N.E.
`Browser`	Genuine	N.E.	N.E.	30%	10%
`CrossyRoad`	Genuine	N.E.	N.E.	40%	N.E.
`CustomSMSManager`	Genuine	30%	10%	N.E.	N.E.
`Firefox`	Genuine	N.E.	N.E.	30%	N.E.
`Flow`	Genuine	N.E.	N.E.	30%	N.E.
`FruitNinja`	Genuine	N.E.	N.E.	10%	N.E.
`Google Earth`	Genuine	N.E.	N.E.	40%	N.E.
`Google Maps`	Genuine	N.E.	N.E.	40%	N.E.
`KingCalculator`	Genuine	N.E.	N.E.	20%	N.E.
`SMS Manager`	Genuine	40%	10%	N.E.	N.E.
`WhatsApp`	Genuine	10%	10%	50%	N.E.
`YouTube`	Genuine	N.E.	N.E.	50%	N.E.

not able to detect such a causality. We can argue similarly for Policy HTTP-H, stating that with a certain probability the opening of a `webView` by the user is an event expected before performing a http connection.

The definition of causality relations and, more in general, history-based policies increases the expressiveness, enabling the specification of conditions that allow to discern the actions of benign apps from malicious ones. For example, the `Whatsapp` app sends an SMS message for registration to a number that is not in the contact list, approximately between the fifth and tenth step of the app execution. Hence, it is possible to shape a specific policy to allow the SMS related action only once in this specific time interval.

The verification of the probabilistic history-based policies is conducted by using the PRISM model checker [14,30], by setting the values of k and d of Figure 7 to 5. Tables 4 and 5 report the values for the contracts generated from the genuine and malicious apps, respectively, with respect to the two history-based policies.

By analyzing Policy SMS-H, we first observe that no genuine apps exhibit behaviors that may violate it, except for `Whatsapp`, which confirms the behavior previously discussed. In particular, notice that the genuine SMS managers, which do not satisfy Policies 1 and 3 (see Table 4) for several threshold values, in the history-based setting are policy compliant for every $p \geq 0$. In other words, Policies 1 and 3 are too restrictive for genuine apps, while Policy SMS-H is not at all.

In the case of malware apps, the experimental results reveal the malicious actions of SMS Trojans similarly as done in the action-based setting. However, they prove to be more effective if compared to the action-based SMS policies. For instance, the malicious family `DogoWar`, which does not violate Policy 3 (see Table 1) as it sends (spam) messages to contacts only, violates Policy SMS-H as it sends messages to contacts that are not selected by the user through the contact list. As another example, the malicious behavior of the SMS Trojans can be easily detected with any policy whose admitted parameter p is lower than 20%, see Table 5.

We can argue similarly in the case of Policy HTTP-H. Such a policy is aimed at limiting the number of covert channel connections performed by both genuine and malicious apps, in particular malware belonging to the spyware class.

As shown in Table 5, the unexpected connections of the spyware families `KMin` and `Vidro` are easily detected, and the same holds in specific intervals of time also for `OpFake`, which attempts to download additional malicious packages to install on the device [40]. In the case of Table 4, the analysis has a controversial effect. In fact, Internet functionalities are embedded in many applications, even when they are not network-based, being used to upload or download data and to provide in-app advertisement. An example of genuine app that would violate the HTTP-H policy even for high values of parameter p is given by `YouTube`, which, being a streaming application, makes massive use of http traffic, without actually opening WebViews. However, the obtained results are still more expressive than the action-based ones. Indeed, action-based policies would

Table 4 History-based verification: contract values for policy critical actions.

Family	**Policy SMS-H**		**Policy HTTP-H**	
	$[t_s, t_e]$	p	$[t_s, t_e]$	p
`AngryBirds`	1-5	0%	1-5	8%
	6-10	0%	6-10	15%
	11-15	0%	11-15	16%
	16-20	0%	16-20	39%
`Browser`	1-5	0%	1-5	0%
	6-10	0%	6-10	0%
	11-15	0%	11-15	0%
	16-20	0%	16-20	0%
`CustomSMSManager`	1-5	0%	1-5	0%
	6-10	0%	6-10	0%
	11-15	0%	11-15	0%
	16-20	0%	16-20	0%
`Firefox`	1-5	0%	1-5	0%
	6-10	0%	6-10	0%
	11-15	0%	11-15	0%
	16-20	0%	16-20	0%
`Google Earth`	1-5	0%	1-5	28%
	6-10	0%	6-10	71%
	11-15	0%	11-15	78%
	16-20	0%	16-20	64%
`Google Maps`	1-5	0%	1-5	21%
	6-10	0%	6-10	51%
	11-15	0%	11-15	60%
	16-20	0%	16-20	54%
`SMS Manager`	1-5	0%	1-5	0%
	6-10	0%	6-10	0%
	11-15	0%	11-15	0%
	16-20	0%	16-20	0%
`WhatsApp`	1-5	0%	1-5	19%
	6-10	38%	6-10	32%
	11-15	0%	11-15	28%
	16-20	0%	16-20	25%
`You Tube`	1-5	0%	1-5	37%
	6-10	0%	6-10	71%
	11-15	0%	11-15	84%
	16-20	0%	16-20	79%

stop connections from the `Firefox` and native Android `Browser` apps. Instead, any history-based policy would not consider their connections as malicious since they are always preceded by the opening of a `WebView`. Moreover, as previously noticed, the effect of blocking connections might be desirable for users, especially when the blocked connections are related to in-app advertisement. In particular, apps such as games might generate a considerable amount of data traffic mainly for providing undesirable contents such as in-app advertisement. The game `Angry Birds`, considered in our analysis, falls in this category, generating a considerable amount of traffic, partially identifiable through the results in Table 4, to provide advertisement. In the history-based scenario such an overhead can be monitored in specific intervals of time, by possibly preventing peaks that would decrease the quality of experience. Thus, a mechanism of policy enforcement would produce an effect similar to the one presented in [39], where the user has been able to play the game, without too many interruptions caused by advertisement.

6.3 Discussion

The policy specification language of the history-based probabilistic security-by-contract approach is a probabilistic logic allowing for the definition of complex and expressive policies, taking into account probabilistic elements and sequence of events. Such an approach is supported by an enforcement infrastructure sufficiently capillar, which is effectively able to capture and control actions in Android devices at API level performed by any app running on the device. Powered by these two elements, the S×C×P-App becomes a complete device security manager, able to enforce virtually any control policy and prevent the action of several malware types. In particular, the enhanced expressiveness of probabilistic policies with respect to standard, determinis-

Table 5 History-based verification: contract values for policy critical actions.

Family	Policy SMS-H		Policy HTTP-H	
	$[t_s, t_e]$	p	$[t_s, t_e]$	p
Dogowar	1-5	23%	1-5	29%
	6-10	58%	6-10	2%
	11-15	61%	11-15	14%
	16-20	64%	16-20	12%
FakePlayer	1-5	34%	1-5	0%
	6-10	26%	6-10	0%
	11-15	26%	11-15	0%
	16-20	23%	16-20	0%
HippoSMS	1-5	15%	1-5	0%
	6-10	53%	6-10	0%
	11-15	21%	11-15	0%
	16-20	18%	16-20	0%
Kmin	1-5	0%	1-5	35%
	6-10	0%	6-10	24%
	11-15	0%	11-15	21%
	16-20	0%	16-20	27%
Mania	1-5	34%	1-5	0%
	6-10	61%	6-10	0%
	11-15	39%	11-15	0%
	16-20	42%	16-20	0%
OpFake	1-5	1.2%	1-5	2.8%
	6-10	23%	6-10	61%
	11-15	11%	11-15	23%
	16-20	18%	16-20	43%
Raden	1-5	12%	1-5	0%
	6-10	21%	6-10	0%
	11-15	14%	11-15	0%
	16-20	20%	16-20	0%
Vidro	1-5	0%	1-5	69%
	6-10	0%	6-10	31%
	11-15	0%	11-15	47%
	16-20	0%	16-20	33%

tic, action-based ones, has been remarked by defining policies that are effective in controlling the actions of SMS-Trojans and Spyware, which currently amount at more than 90% of the existing malware in the wild [25]. However, the same approach, based on the probabilistic formalization of causality relations among actions, is sufficiently expressive to consider other malware types. More examples of controlled behaviors are the following ones:

– Controlling events related to administrator privileges, requiring express consents from the user. In particular, it is possible to define policies like, e.g., "A new device PIN should not be set if the user is not prompted to insert and choose it ". This kind of policy would be effective in blocking the new Android ransomware `DoubleLocker`[1].
– Controlling crypto-library invocations, thus ensuring, e.g., that a file cannot be encrypted by an app unless the app itself generated such a file. This would prevent the action of ransomware belonging to the Cryptolocker families.
– Controlling the event of accessing the SMS message outbox, where a message, if accessed, should have the content to be shown to the user. This policy would be effective in stopping the actions of banking Trojans such as `Zitmo`.

6.4 Performance Considerations

As discussed, the operative workflow of the S×C×P framework is made of two main phases, the Contract-Policy matching ($C \preceq P$) and the Enforcement ($A \preceq P$), which impact differently on the user experience. The $C \preceq P$ phase has a slight impact at app deploy time, and is quantified in less than 2 seconds. The deployment is subject to the matching result. In fact, if the contract is compliant with the policy, the application is going to be installed without further delay, otherwise the user will be asked to remove the app, or to enforce the policy dynamically. The enforcement ($A \preceq P$) brings a more consistent overhead, related

[1] https://www.welivesecurity.com/2017/10/13/doublelocker-innovative-android-malware/

Table 6 Performance overhead measures.

	Action-based	**History-based**
$C \preceq P$	0.7s	2.7s
$A \preceq P$	0.5s	2.2s
CPU overhead	4%	16%
Memory Overhead	12%	18%
I/O	0,7%	13%
Battery Depletion	14%	65%

to the hijacking of critical API calls and continuous policy reevaluation. This overhead is however limited and differs for the two approaches, i.e., action-based and history-based. For action-based enforcement, the evaluation is not dependent from the execution of the PRISM model checker, hence provides a quite limited overhead. On the other hand, the history-based enforcement does require continuous verification through the PRISM evaluation tool. This introduces a noticeable overhead, related both to execution delay and energy consumption.

Table 6 summarizes the registered overhead introduced by the S×C×P framework when operating in the action-based or the history-based modes. The experiments have been performed on a Samsung Galaxy Nexus with the same configuration of those performed on apps, averaging five set of experiments repeated in the same conditions. The first row of Table 6 reports the time needed to perform a contract-policy analysis on the `AngryBirds` app. The second line instead reports the mean time needed to verify a single critical action. Hence, this reports the effective delay in the performance of an operation, representing thus the effective delay that might be experienced by the user even if the action is allowed. As anticipated, this overhead is manageable for the action-based verification, instead it impacts on the user experience for the history-based one. The parameters on overhead for CPU, Memory and I/O have been extracted by using the `Quadrant` app, computing the performance difference with the S×C×P-App active and inactive while performing policy enforcement. As expected, the overhead is higher for the history-based mode, especially for what concerns CPU overhead and I/O, mainly due to the continuous write and read operations in the shared-preference buffer. Finally, Table 6 reports the impact on battery duration, by measuring and averaging five discharge cycles both with the S×C×P-App active and inactive, while a user is actively interacting with a monitored app. As expected the enforcement overhead is consistent, noticeably shortening the battery duration, especially in the case of history-based enforcement, which more than halves the battery duration. Therefore, the two approaches have both advantages and disadvantages. The history-based approach is, in fact, more expressive and allows the enforcement of more complex policies, which brings the benefits previously discussed. However, the enforcement of the history-based approach imposes an overhead that might make the solution non-practical for commercial use. A possible trade-off would consist of using the history-based approach in the $C \preceq P$ phase and then, if the contract is not compliant, the policy would effectively be enforced by using the action-based approach, thus guaranteeing a more manageable overhead.

It is worth pointing out that the current implementation of the S×C×P has to be considered as a proof of concept, mainly intended for research uses related to application behavioral analysis. A noticeable performance improvement can derive from the integration of the action hijacking mechanism naively in the Android OS. Such an integration would make unnecessary the usage of the shared preference buffer, which is responsible for the most consistent part of the history-based approach overhead. In particular, in a proof-of-concept implementation of modified Android OS, tested on the native Android emulator, the overall overhead has been reduced of 70% with respect to the shared-preference-based approach.

7 Conclusion

Verifying the compliance of Android app behaviors with security policies is a challenging task, which requires expressive formalisms and effective enforcement mechanisms. In this paper, we have described the S×C×P framework for Android app security, a system which allows the definition and enforcement of probabilistic history-based security policies directly on device. The proposed framework is designed to be applied in different environments, to enforce policies related to private user security, BYOD and business related use cases. In fact, though in the current implementation we have only considered a small set of actions, the approach can be extended to any Android API, allowing thus the enforcement of virtually every policy. The probabilistic model introduces the possibility of having more flexible policies, useful to ensure security without limiting the genuine still complex activity of good applications [2]. We have reported a set of experiments to show the effectiveness in tackling malicious and unwanted behavior, for both malware and genuine apps.

Future work stemming from this research are integration of additional APIs for more domain specific policies, such as BYOD, driver safety and e-health. Furthermore, it should be investigated the integration of the S×C×P framework directly in the OS, both for Android and other systems, such as desktop environments, Internet of Things settings, or SCADA systems.

References

1. Xposedbridge development tutorial (2012). URL `https://github.com/rovo89/XposedBridge/wiki/Development-tutorial`
2. Aldini, A., Bernardo, M.: A formal approach to the integrated analysis of security and qos. Reliability Engineering and System Safety **92**(11), 1503–1520 (2007)
3. Aldini, A., Bravetti, M., Di Pierro, A., Gorrieri, R., Hankin, C., Wiklicky, H.: Two formal approaches for approximating noninterference properties. In: Foundations of Security Analysis and Design II, vol. LNCS 2946, pp. 1–43. Springer (2004)
4. Aldini, A., Di Pierro, A.: A quantitative approach to noninterference for probabilistic systems. In: Procs. of Formal Methods for Security and Time, vol. 99, pp. 155–182. ENTCS (2004)
5. Aldini, A., Martinelli, F., Saracino, A., Sgandurra, D.: A collaborative framework for generating probabilistic contracts. In: Procs. of the 2013 IEEE Int. Conf. on Collaboration Technologies and Systems, pp. 139–143 (2013). DOI 978-1-4763-6404-1/13
6. Aldini, A., Martinelli, F., Saracino, A., Sgandurra, D.: Detection of repackaged mobile applications through a collaborative approach. Concurrency and Computation: Practice and Experience **27**(11), 2818–2838 (2015)
7. Aldini, A., Seigneur, J.M., Lafuente, C., Titi, X., Guislain, J.: Design and validation of a trust-based opportunity-enabled risk management system. Information and Computer Security **25**(1), 2–25 (2017). DOI 10.1108/ICS-05-2016-0037
8. Backes, M., Bugiel, S., Derr, E., Gerling, S., Hammer, C.: R-Droid: Leveraging android app analysis with static slice optimization. In: Pros. of the 11th ACM on Asia Conf. on Computer and Communications Security, ASIA CCS'16, pp. 129–140. ACM, New York, NY, USA (2016). DOI 10.1145/2897845.2897927
9. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
10. Bergstra, J., Ponse, A., Smolka, S.: Handbook of Process Algebra. Elsevier (2001)
11. Bielova, N., Massacci, F.: Predictability of enforcement. In: Procs. of the Int. Symposium on Engineering Secure Software and Systems, ESSoS'11, *LNCS*, vol. 6542, pp. 73–86. Springer (2011)
12. BusinessOfApps: App statistic report. Tech. rep. (2016). Available at: http://www.businessofapps.com/data/app-statistics/
13. Cha, S.H.: Comprehensive survey on distance/similarity measures between probability density functions. International Journal of Mathematical Models and Methods in Applied Sciences **1**(4), 300–307 (2007)
14. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Procs. of the 19th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), *LNCS*, vol. 7795, pp. 185–191. Springer (2013)
15. Costa, G., Dragoni, N., Lazouski, A., Martinelli, F., Massacci, F., Matteucci, I.: Extending Security-by-Contract with quantitative trust on mobile devices. In: Procs. of the 4th Int. Conf. on Complex, Intelligent and Software Intensive Systems, pp. 872–877. IEEE CS (2010)
16. Delahaye, B., Caillaud, B., Legay, A.: Probabilistic contracts: A compositional reasoning methodology for the design of stochastic systems. In: Procs. of 10th Int. Conf. on Application of Concurrency to System Design, ACSD'10, pp. 223–232. IEEE (2010)
17. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theoretical Computer Science **318**, 323–354 (2004)
18. Dini, G., Martinelli, F., Matteucci, I., Petrocchi, M., Saracino, A., Sgandurra, D.: A multi-criteria-based evaluation of Android applications. In: Procs. of the 4th Int. Conf. on Trusted Systems (INTRUST'12), LNCS, pp. 67–82. Springer (2012)
19. Dini, G., Martinelli, F., Matteucci, I., Petrocchi, M., Saracino, A., Sgandurra, D.: Risk analysis of android applications: A user-centric solution. Future Generation Computer Systems (2016). DOI https://doi.org/10.1016/j.future.2016.05.035
20. Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.: Madam: A multi-level anomaly detector for android malware. In: I. Kotenko, V. Skormin (eds.) Computer Network Security, *LNCS*, vol. 7531, pp. 240–253. Springer (2012)
21. 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. In: E. Di Nitto, A.M. Sassen, P. Traverso, A. Zwegers (eds.) At Your Service - Service-Oriented Computing from an EU Perspective, pp. 429–456. MIT Press (2008)
22. Easwaran, A., Kannan, S., Lee, I.: Optimal control of software ensuring safety and functionality. Tech. Rep. MS-CIS-05-20, University of Pennsylvania (2005)
23. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones. Commun. ACM **57**(3), 99–106 (2014). DOI 10.1145/2494522
24. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. In: Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012, p. 3 (2012)
25. Funk, C., Garnaeva, M.: Kaspersky security bullettin 2013. Tech. rep. (2013). URL `http://media.kaspersky.com/pdf/KSB_2013_EN.pdf`
26. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of Android malware using embedded call graphs. In: Procs. of the 2013 ACM Workshop on Artificial Intelligence and Security, AISec'13, pp. 45–54. ACM (2013). DOI 10.1145/2517312.2517315
27. Grinstead, C., Snell, J.: Introduction to Probability. American Mathematical Society (2012)
28. Hoang, X., Hu, J.: An efficient hidden Markov model training scheme for anomaly intrusion detection of server applications based on system calls. In: Procs. of 12th IEEE Int. Conf. On Networks, ICON'04, vol. 2, pp. 470–474. IEEE (2004)
29. Kosoresow, A., Hofmeyer, S.: Intrusion detection via system call traces. Software **14**(5), 35–42 (1997)
30. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: G. Gopalakrishnan, S. Qadeer (eds.) Proc. of the 23rd Int. Conf. on Computer Aided Verification (CAV'11), *LNCS*, vol. 6806, pp. 585–591. Springer (2011)

31. Larsen, R., Marx, M.: An Introduction to Mathematical Statistics and Its Applications. Pearson (2011)
32. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing **7**(4), 381–395 (2010)
33. Marra, A.L., Martinelli, F., Saracino, A., Aldini, A.: On probabilistic application compliance. In: 2016 IEEE Conf. Trustcom/BigDataSE/ISPA, Tianjin, China, pp. 1848–1855 (2016). DOI 10.1109/TrustCom.2016.0283
34. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. ENTCS **179** (2007)
35. Martinelli, F., Mercaldo, F., Saracino, A., Visaggio, C.A.: I find your behavior disturbing: Static and dynamic app behavioral analysis for detection of android malware. In: 2016 14th Annual Conference on Privacy, Security and Trust (PST), pp. 129–136 (2016). DOI 10.1109/PST.2016.7906947
36. Martinelli, F., Morisset, C.: Quantitative access control with partially-observable Markov decision processes. In: Procs. of 2nd ACM Conf. on Data and Application Security and Privacy, CODASPY'12, pp. 169–180. ACM (2012)
37. Ponemon Institute: The state of mobile application insecurity. Tech. rep. (2015)
38. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (2010)
39. Saracino, A., Martinelli, F., Alboreto, G., Dini, G.: Data-Sluice: Fine-grained traffic control for Android application. In: IEEE Symposium on Computers and Communication, ISCC'16, pp. 702–709 (2016). DOI 10.1109/ISCC.2016.7543819
40. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: MADAM: Effective and efficient behavior-based android malware detection and prevention. IEEE Transactions on Dependable and Secure Computing (2016). DOI 10.1109/TDSC.2016.2536605
41. Suarez-Tangil, G., Tapiador, J., Lombardi, F., Di Pietro, R.: Thwarting obfuscated malware via differential fault analysis. Computer **47**(6), 24–31 (2014). DOI 10.1109/MC.2014.169
42. Wang, R., Azab, A.M., Enck, W., Li, N., Ning, P., Chen, X., Shen, W., Cheng, Y.: SPOKE: Scalable knowledge collection and attack surface analysis of access control policy for security enhanced Android. In: Procs. of the 2017 ACM on Asia Conf. on Computer and Communications Security, ASIA CCS '17, pp. 612–624. ACM, New York, NY, USA (2017). DOI 10.1145/3052973.3052991
43. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Procs. of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14, pp. 1105–1116. ACM (2014). DOI 10.1145/2660267.2660359