Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Automatic security verification of mobile app configurations[☆]

Gabriele Costa[a], Alessio Merlo[a,*], Luca Verderame[a], Alessandro Armando[a,b]

[a] DIBRIS, Università degli Studi di Genova, Italy
[b] Security & Trust Unit, FBK-irst, Trento, Italy

HIGHLIGHTS

- A practical approach to the validation of groups of mobile apps is presented.
- The approach relies on partial model checking for mitigating state explosion.
- Experiments on real applications show that the technique scales on real systems.
- The solution is integrated with a prototype of the Secure Meta-Market (SMM).

ARTICLE INFO

Article history:
Received 27 August 2015
Received in revised form
16 May 2016
Accepted 17 June 2016
Available online xxxx

Keywords:
BYOD paradigm
Android security
Partial model checking
Policy enforcement
Automated verification

ABSTRACT

The swift and continuous evolution of mobile devices is encouraging both private and public organizations to adopt the *Bring Your Own Device* (BYOD) paradigm. As a matter of fact, the BYOD paradigm drastically reduces costs and increases productivity by allowing employees to carry out business tasks on their personal devices. However, it also increases the security concerns, since a compromised device could disruptively access the resources of the organization. The current mobile application distribution model based on application markets does not cope with this issue. In a previous work the concept of secure meta-market has been introduced as a mean to distribute mobile applications always guaranteed to comply with any given BYOD policy. This is achieved through a suitable combination of static analysis (i.e. model checking) and code instrumentation techniques. Although crucial, enforcing security policies over individual applications is not sufficient in general. Indeed, several well documented threats arise from the malicious interaction among applications which are harmless if isolated. In this paper, a novel technique for the security verification of groups of mobile app is proposed. The approach relies on partial model checking (PMC) to extend the existing security guarantees to groups of applications. The experimental results demonstrate the viability of the approach. Moreover, we show through a case study that even a fairly simple security policy can be violated by applications which are compliant if considered one by one.

1. Introduction

The relentless evolution of mobile technologies is pushing forward the interest for novel, ubiquitous paradigms. Among them, the *Bring Your Own Device* [1] (BYOD) is polarizing attention and investments [2]. The BYOD paradigm allows people to join an organization and access its resources by using their own device. Since it promises cost reduction and increased productivity, it is not surprising that the BYOD paradigm is receiving a growing attention from organizations worldwide.

However, securing BYOD environments poses new challenges due to the twofold role of devices. On the one hand, the personal use of devices must comply with the security policy of the organization. Since the behavior of a device depends on the installed software, some policy enforcement technology must verify or monitor its execution. On the other hand, the enforcement of the security policy should not affect the user experience. It must be noted that current mobile applications deployment frameworks, mostly based on dedicated web services called *applications markets* (e.g. Apple Store, Google Play, Samsung Store, . . .), provide little or even no support to tackle these challenges. Some application markets implement a review process, see e.g., [3,4], featuring some

☆ This work has been partially funded by the Italian PRIN project *Security Horizons* (no. 2010XSEMLC).
* Corresponding author.
E-mail addresses: gabriele.costa@unige.it (G. Costa), alessio.merlo@unige.it (A. Merlo), luca.verderame@unige.it (L. Verderame), armando@fbk.eu (A. Armando).

http://dx.doi.org/10.1016/j.future.2016.06.014
0167-739X/© 2016 Elsevier B.V. All rights reserved.

form of security analysis, but they are poorly documented and it is therefore unclear what kind of guarantees they provide. Moreover, severe limitations have been reported. For instance, malicious apps have been positively reviewed and published [5]. This lack of security support is often mitigated by asking to the device owner to take security-critical decisions, e.g. to decide whether to install an application on the ground of the permission requested and the trustworthiness of the developer. Needless to say, the vast majority of users lacks the skills to take informed decisions and this approach therefore exposes organizations to dangerous threats.

The concept of *Secure Meta-Market* (SMM) [6,7] has been put forward to tackle the above challenges, and it is now receiving the attention of the mobile security community and some implementations recently appeared (e.g. see [8]). A SMM masks the actual application markets by mediating all the installation requests and supports the enforcement of BYOD security policies through formal analysis and monitoring inlining. This is achieved by supporting (i) the definition of fine-grained security policies, (ii) the static verification of applications through model checking, and (iii) the security instrumentation necessary to monitor the applications that fail the verification step.

As detailed in [7] where a prototype implementation is proposed, these techniques rely on a model t_a (formally describing the behavior of an application a) and a formula ϕ_p (formally specifying the behavior complying with a security policy p). Verification is done by checking whether t_a enjoys ϕ_p, in symbols $t_a \models \phi_p$. Instrumentation is done by turning a into a new application a' that, by construction, is guaranteed to enjoy ϕ_p, i.e. $t_{a'} \models \phi_p$.

Although the experimental results presented in [7] are encouraging, they are limited to the verification of individual applications. Therefore the scalability of the approach on *application configurations*, i.e. finite sets of applications, say $a_1, \ldots, a_n$, is still to be ascertained. This is an important problem, since attacks leveraging the unexpected interplay of multiple applications are known. For instance, in the *confused deputy attack* [9], a malicious application that does not have the appropriate permissions to download a file invokes a second one (e.g., a browser) to download it.

The problem of checking whether an application configuration $\{a_1, \ldots, a_n\}$ complies with a policy p can be reduced to checking whether $(t_{a_1} \| \cdots \| t_{a_n}) \models \phi_p$, where the t_{a_i} are the models of the applications a_i and $(t_{a_1} \| \cdots \| t_{a_n})$ denotes their parallel composition. Unfortunately even state-of-the-art model checkers scale poorly as n increases because of the well-known state-explosion problem [10]. *Partial Model Checking* (PMC) [11] is a technique that allows to reduce a model checking problem of the form $(t_1 \parallel t_2) \models \phi$ to the problem $t_2 \models \phi /\!\!/ t_1$, where $\phi /\!\!/ t_1$ is a formula obtained by partially evaluating ϕ with respect to t_1. The key observation is that even though $\phi /\!\!/ t_1$ can be larger than ϕ, t_2 is usually considerably smaller than $(t_1 \parallel t_2)$. For this reason, PMC can play a pivotal role in combating the state-explosion problem.

This paper shows that PMC can be effectively used to mitigate the state-explosion problem and considerably improve the scalability of the SMM paradigm. This is done by showing how the verification of application configurations can be reduced to a sequence of PMC problems and by presenting experimental results showing that PMC can lead to verification times that are up to 3 orders of magnitude smaller than those obtained by traditional model checking techniques only.

The rest of the paper is focused on static analysis only. Details on the instrumentation and monitoring techniques adopted in the SMM can be found in [7].

This paper is structured as follows. Section 2 puts the paper in context by discussing the related work. The concept of SMM is introduced in Section 3. The verification of mobile applications through PMC is discussed in Section 4. The usage of PMC in BYODroid is illustrated in Section 5. The experimental results are reported and discussed in Section 6. Finally, Section 7 draws some concluding remarks.

2. Related work

Since the BYOD paradigm has been proposed only recently, the literature addressing the associated security concerns is very limited at present. Nevertheless, several works propose solutions for applications security analysis which might be considered for implementing the BYOD security frameworks.

Android application security. The Android permission system is the basic element for defining application privileges and enforce them at runtime. However, since its first appearance it received many criticisms, e.g., see [12], and many authors, e.g., see [13–15], proposed modifications and improvements, mainly focused on enhancing the security for user's activities (e.g. E-commerce [16]) as well as reducing the exploitability of device vulnerabilities (e.g. [17,18]). Still, permissions are not sufficient to define fine-grained policies as those needed for BYOD systems.

Several authors presented approaches that extend or redefine the Android security framework with a particular attention to data flow analysis. For instance, FlowDroid [19] analyzes data-flows over Android applications and, its extensions [20,21] also deal with multiple Android applications interacting through IPC channels. A similar approach explicitly dealing with enterprise data is put forward in [22]. Instead, Scandroid [23] checks data flows through applications against security specifications extracted from the manifest, while [24] applies a reachability analysis to discover whether applications leak sensitive information through the network. Runtime enforcement solutions include TaintDroid [25], i.e., a system-wide dynamic taint tracking system capable to simultaneously track multiple sources of sensitive data, and Kynoid [26], i.e., a monitoring framework for user-defined security policies for data-items. Although relevant, data flow analysis does not cover usage control aspects which are central in BYOD environments. Moreover, these approaches mostly target the validation of single applications while we are also interested in the analysis of applications composition. Recently, a security enhanced version of Android, namely SEAndroid [27], has been released for defining system-wide mandatory access control (MAC) policies. Furthermore, in [28] it has been extended to support security policies specified through timed automata. Although SEAndroid permits to define global MAC policies over the OS resources of the device, it lacks the sufficient level of abstraction to deal with BYOD environments.

Policy enforcement relying on formal methods. Formal security analysis techniques require a rigorous, mathematical description of both the security policy and the app behavior. For instance, a common practice consists of a runtime monitor which compares the execution trace of an application against the security policy. Security frameworks based on this approach include [29] for Java Standard Edition, [30] for. NET and [31] for Java Micro Edition. Runtime monitoring can effectively control that the execution of programs comply with a formally defined policy. Policy languages provided with a formal semantics include temporal logics, e.g., LTL [32], and finite state machines, e.g., security automata [33]. Still, runtime enforcement must be carried out by the execution environment, i.e., the mobile device, at the cost of undesirable computational overhead and energy consumption.

On the other hand, static enforcement techniques typically rely on a mathematical structure describing the *behavior* of the application, namely a *model*. Models abstract away details that are considered to be irrelevant for the security analysis. Usually, very abstract models are more compact and easier to handle/analyze while detailed ones lead to longer computations and, in certain cases, can cause intractability. A common practice requires to define a *type and effect system* [34] to infer a model from a piece of code. In [35,6] we followed this approach on a minimal core language [36]. Although simplified, such framework included most of the relevant aspects of the Android IPC and components. The type and effect system infers a *history expression* [37] from each Android component. Also, in [35] we showed that the type and effect system generates *safe* history expressions, i.e., they over-approximate the actual behavior of the application.

Although feasible, redefining the type system for the entire Java language (and also for its bytecode) can be a cumbersome process (due to its rich and redundant syntax) and many proposals only address fragments of the full Java language. In [38] the authors opted for generating history expressions out of *control flow graphs* (CFG). A proof of the soundness of this approach is not available. Nevertheless, several authors worked on the formalization of the Java semantics (e.g., see [36,39,40]) which can be used for such a proof. Remarkably, the CFG correctness has been already obtained for Java 1.4 [41].

Building a CFG from a Java program is a rather common operation and some tools exist. For instance, ConFlEx [42] translates bytecode programs into BIR, an intermediate, stack-less language, and creates a CFG without losing critical information. Similarly, Soot [43] relies on Jimple (Java sIMPLE), a stack-less, three address language, to carry out a similar procedure. Also, the generation of a CFG is a rather efficient process as discussed in [44].

Security policy adaptation. The implementation of a policy refinement/specialization procedure based on partial model checking [11] is a further contribution of this paper. At the best of authors' knowledge, the present work is the first proposal for the application of partial model checking to the verification of configurations of applications. Nevertheless, few works exploited this technique in different fields, in particular, in presence of modular/-compositional contexts. For instance, in [45] partial model checking was applied for synthesizing a specialized security controller. Briefly, the authors considered an agent, e.g., a web browser, which interacts with a possibly malicious module, e.g., a third-party plugin. Starting from a security policy for the whole system, they obtain a security controller which effectively enforce the global policy by only monitoring the untrusted module. Following a similar approach, in [46] the same authors use partial model checking for verifying whether a web service composition complies with a security specification. Alternative implementations of partial model checking are included in existing application analysis frameworks like CADP [47] and mCRL2 [48].

3. Secure Meta-Markets

In the current mobile world, the application deployment is carried out through application markets. An application market (Google Play, Samsung Store, . . .) is an OS-specific store where developers upload and sell their mobile applications. Mobile devices are usually equipped with a client application able to interact with the application market, thereby allowing the user to browse among applications and select those to install.

From a security point of view, application markets put much of the responsibility on the shoulder of the user which takes the decision whether to install an application or not. To support the decision, application markets often attach some coarse-grained information to each application (e.g., application rating and reputation of the developer). In a few cases (e.g., Google Bouncer[1]), application markets carry out some sort of security analysis on the applications. However, these solutions are scarcely documented and provide little assurance [49]. All in all, current application markets do not meet the requirements posed by the BYOD paradigm.

[1] http://googlemobile.blogspot.it/2012/02/android-and-security.html.

The concept of SMM aims at overcoming these difficulties. As shown in Fig. 1, a SMM acts as a security-enabled application market which enforces corporate BYOD policies on personal devices. Each corporation can install its own SMM and define the security policies suitable to its needs by using appropriate policy specification languages (e.g., ConSpec [50]—see Section 6). The SMM masks the actual application markets. For each application that the user wants to install, the SMM retrieves the application package directly from the official application market. Then, it verifies the compliance of the application against the current personal device configuration (i.e. the set of installed applications) and the BYOD policy. In case of non-compliance, the SMM proceeds by instrumenting the application. Thus, the installation deploys a piece of software (original or instrumented) which keeps the device configuration compliant with the BYOD policy.

It must be emphasized that the SMM is transparent to both the users and the application markets. The SMM features are supported through the workflow depicted in Fig. 2.

The workflow works as follows: code producers compile (P.0) and generate mobile applications (P.1); then, they publish their applications (P.2) on a standard application market which stores (M.0) the software packages in a database (ADB). When a user requires an application from the meta-market, the corresponding application is retrieved. Then, a model extraction procedure is applied to the application (B.0) to generate an application model (B.1). Since the model is extracted from the application code, no further validation is required. Hence, the model can be directly passed (B.2) to a verification process which checks its compliance against the security policy (B.3). Security policies are retrieved (B.4) from a policy database (PDB) handling policy instances customized over the devices configuration.

If the verification succeeds (B.5 $\rightarrow$ YES) the policy database is updated by means of PMC (B.6) and the application is delivered to the user's device with no further action (B.7). Otherwise (B.5 $\rightarrow$ NO), the SMM attaches monitoring information to the application (B.8). Mainly, monitoring information consists of a digital signature which will be used by the code consumer to obtain a correct instrumentation of the application. When the consumer receives a mobile application package, she checks whether it was marked for monitoring (C.0). If this is the case, before installation the mobile device instruments the application package with security checks by using information attached by the meta-market (C.1). Otherwise, it is directly installed.

4. Formal security assessment

Static verification is a core activity of the SMM and, together with code instrumentation, guarantees that no security violations occur at runtime. The verification process consists of three steps: (i) the extraction of behavioral models from the application packages, (ii) the exhaustive search for illegal execution traces via model checking, and (iii) the partial evaluation of the security policy with respect to the verified models. Below we present these three steps and discuss the existing approaches and technologies implementing them.

4.1. Application modeling

Android software packages (APKs) carry application code and non executable resources (e.g., pictures and multimedia files). Application code primarily consists of Android VM bytecode and, possibly, machine executable, a.k.a. *native*, modules. Every access to the valuable resources of the execution platform is performed by invoking a corresponding API or system call.

Control flow graphs. A control flow graph (CFG) is a data structure representing the possible execution flows of a piece of software.

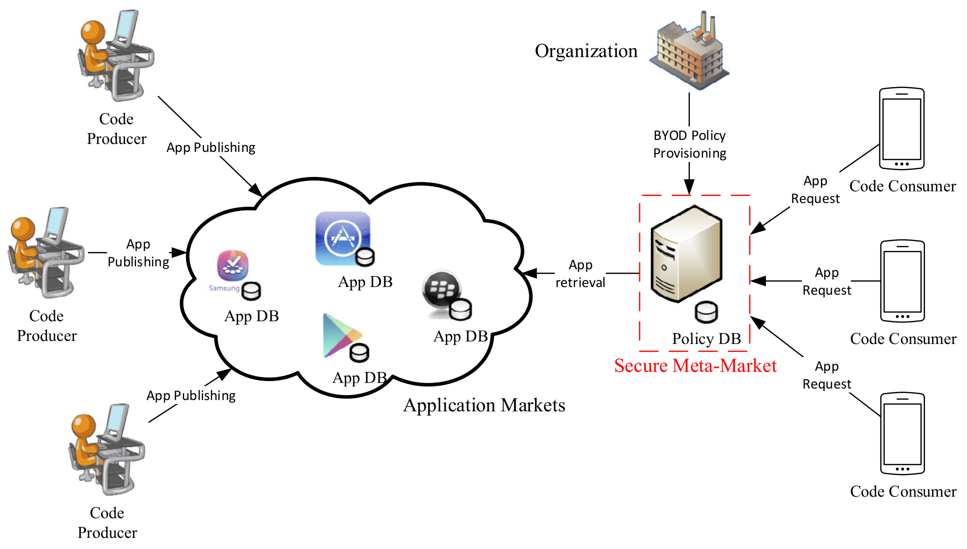

Fig. 1. The Secure Meta-Market approach.

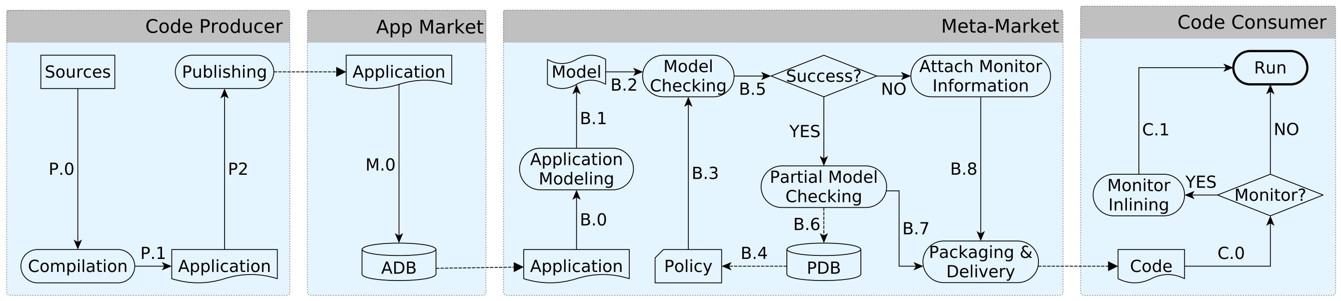

Fig. 2. The Secure Meta-Market workflow.

```
class C extends
  BroadcastReceiver
  {
  void onReceive
     (Context c,
      Intent i) {
    String s =
      "file://sdcard/f";
    String t =
      i.getExtra("path");
    File f = new File(s);
    File g = new File(t);
    if(f.equals(g)) {
      f.createNewFile();
    }
    else {
      f.delete();
      g.createNewFile();
    }
    Intent j =
      new Intent(
        c,
        someService.class);
    j.putExtra("uri", s);
    c.startService(j);
  }
}
```

```
  ldc  v3,  "file://sdcard/f"
  ldc  v4,  "path"
  ivk  v2,  v4,
    .getExtra(String)
  mvr  v5
  new  v6,  java.io.File
  ivk  v6,  v3,  .<init>(String)
  mvr  v6
  new  v7,  java.io.File
  ivk  v7,  v5,  .<init>(String)
  mvr  v7
  ivk  v6,  v7,  .equals()
  mvr  v8
  ifz  v8,  l1
l0:
  ivk  v6,  .delete()
  ivk  v7,  .createNewFile()
  goto l2
l1:
  ivk  v6,  .createNewFile()
l2:
  new  v9,
    android.content.Intent
  sget v10, someService.class
  ivk  v1,  v10,
    .<init>(Context, Class)
  mvr  v9
  ldc  v11, "uri"
  ivk  v9,  v11, v3,
    .putExtra(String, String)
  ivk  v1,  v9,
    .startService(Intent)
  ret
```

Fig. 3. A (simplified) example of the model extraction process.

Intuitively, control flow instructions, i.e., conditional branches, unconditional jumps and loops, result in edges connecting the nodes of the CFG. Nodes can be either basic blocks, i.e., linear, jump-free sequences of instructions, or conditional nodes, i.e., those representing the guard of a branching instruction. The level of abstraction of a CFG mostly depends on the instructions appearing in its nodes. Let assume two sets of instructions Δ and Γ, denoting the security-relevant relevant operations and the IPC invocations, respectively, are defined. Clearly, the content of Δ corresponds to the alphabet of the security policy under evaluation and typically changes when a new policy is considered. Instead, Γ is constantly defined as the set of the Android IPC APIs. To disambiguate, let us write $\bar{\alpha}, \beta \in \Gamma$ to denote that α is an incoming action and β an outgoing one. The following simple example will provide the basic intuition.

Example 1. Consider the fragment of Java code on the left of Fig. 3. It represents a minimal implementation of an Android `BroadcastReceiver`. It defines a method `onReceive` which can be triggered by sending an appropriate intent through the invocation `sendBroadcast`. The method creates two `File` objects (f and g) pointing to the constant path `"file://sdcard/f"` and a location carried by the received intent, respectively. If the two files are equal (i.e., they have the same path), f is created, otherwise f is deleted an g created. The method terminates by invoking an Android service called `someService`. To this end, a new intent is created, populated with some data (here with the pair of values `"uri"` and `"file://sdcard/f"`) and passed to the method `startService`.

Upon compilation, a piece of bytecode, resembling that reported on the right-hand side of Fig. 3, is generated. Method invocations (instruction `ivk`) contain a list of registers (e.g., `v1, v6`) for the actual parameters and a pointer to the invoked method

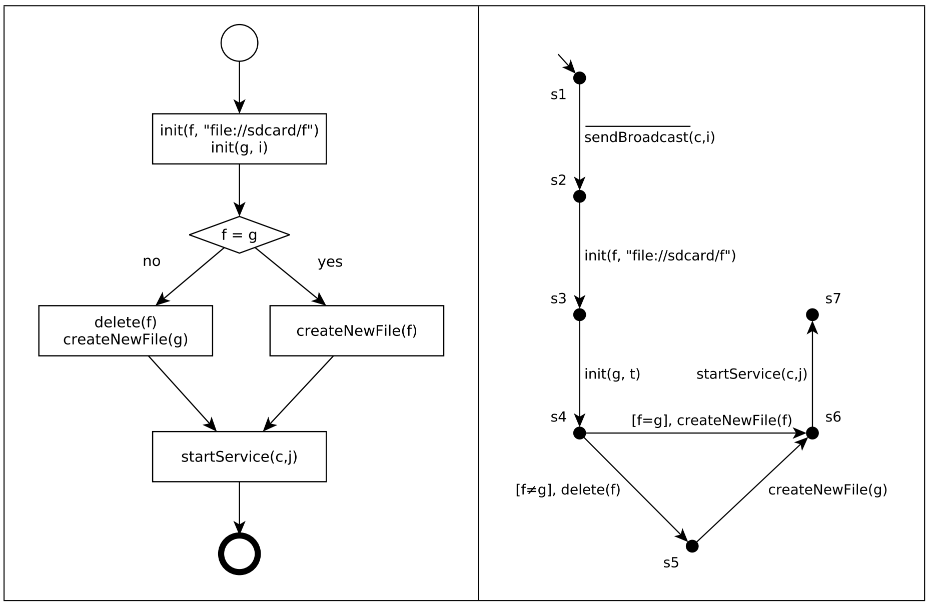

Fig. 4. A (simplified) example of the model extraction process (contd.)

(e.g., `.startService(Intent)`). The other involved instructions are `ldc` (load a constant in a register), `new` (creates an instance of the given class type), `mvr` (puts the result of the previous operation in the specified register), `ifz` (jumps to an instruction if the value of a register is `0/true`), `goto` (jumps to the given instruction), `sget` (gets the value of a static field) and `ret` (exit the current block).

The bytecode consists of four blocks. Labels separate blocks and are placed either after a jump instruction or at the location pointed by a jump. For instance, `l0` is located after a conditional instruction `ifz`, while `l2` is pointed by the `goto` before `l1` (notice that `l1` satisfies both the conditions).

The CFG obtained from the given bytecode could resemble that on the left of Fig. 4, where $\Delta = \{$`File.<init>(String)`, `File.delete()`, `File.createNewFile()`$\}$ (`init`, `delete` and `createNewFile` are defined for brevity).[2] The nodes of the CFG correspond to the basic blocks of the bytecode. Moreover, a branching node $\diamond$ is placed where the control flow can take two different paths. In this example, the instruction `.equals` is interpreted as testing the equality of f and g. Also notice that the actions not appearing in $\Delta \cup \Gamma$ (here the only actions in Γ which also appear in the CFG are `startService` and `sendBroadcast`) are dropped from the nodes.[3] □

CFGs are exploited in several analysis techniques and many authors proposed algorithms for the extraction of *safe*, i.e., correctly representing all the execution paths, graphs. As a consequence, several tools exist for extracting a CFG from binaries and bytecode. Among them, Androguard[4] and ConFlEx [42] are the most suitable for the SMM aim. Androguard consists of a suite of tools for the static analysis of Android applications and is specifically designed for processing the Android bytecode. Instead, ConFlEx generates CFGs from Java bytecode even when some components are statically unspecified. Moreover, ConFlEx implements an *incremental* CFG building procedure which is both sound and rather precise. CFGs obtained in this way can be composed when a missing module becomes available so resulting in model refinement. Needless to say, this features is very appealing when extracting models from Android applications as they often use IPC for invoking statically unknown components. When a component of an application is dovetailed with others to form an execution context, their models compose in a single, refined one.

Thus, each CFG is translated into a corresponding STS. The translation is straightforward and several results exist guaranteeing that no information loss occurs (e.g., see [55,56]). The resulting transition systems are then composed. To clarify the translation process we finalize Example 1.

Example 2. Consider again the CFG of Fig. 4 extracted through the steps sketched in Example 1. The CFG is converted into the STS graphically depicted on the right of Fig. 4. Each transition is labeled with the action appearing in the corresponding node of the CFG. The equality check labeling the branching node results in a pair of constraints appearing in the corresponding transitions (elsewhere the constraints amount to the constant *true* and can be omitted for brevity). Since the receiver begins its computation upon receiving an invocation, it is extended with the initial transition $\overline{\texttt{sendBroadcast}}$. Such transition states that the computation of this STS cannot start autonomously, but needs to be triggered by an intent. □

Naively, one could consider the parallel composition of all the models obtained in this way. Although such composition would safely represent the actual behavior of the applications, it would include many traces which never arise in actual executions. Indeed, in several cases the Android components do not execute concurrently. More commonly, when a component invokes another one, it is suspended or even terminated. According to the Android intent mechanism specification [57], component invocations usually transfer the execution flow from the caller to the callee. This causes a significant reduction of the actual number of states obtained from the composition of two or more agents. Notice that, when required, parallel composition is still used, e.g., for Services and asynchronous tasks.

Further reductions concern the return flows. In many cases, upon termination components return the execution flow to their caller. For instance, this happens when the caller invokes the callee

[2] Notice that, for the sake of presentation, in this examples polyadic actions are used instead of monadic ones. The interested reader can find more details on this aspect in [51,52].

[3] Notice that these operations are removed through simplifications, rather than simply discharged. For instance, *constant propagation* [53,54] can be applied before dropping references to variables.

[4] https://github.com/androguard/androguard.

through `startActivityForResult(...)`. In this case, when the callee terminates, it returns some data and the caller resumes its execution. By applying this reasoning, several flows can be removed, i.e., those representing wrongly delivered responses. Although this step cannot be proved to preserve the soundness of the models, there are strong evidences, e.g., the platform documentation, supporting it.

Summing up, for each Android application the presented approach generates an inter-procedural model representing its security-relevant behavior. Application models carry transitions labeled with synchronization actions. These transitions represent the fact that two applications can interact through IPC, possibly transmitting data. Two or more models can be composed similarly to what has been done for the components of the same application. Intuitively, a composition of models represents the execution context in which the applications (the models have been extracted from) will operate at runtime. Instead, the composed model is context-insensitive w.r.t. the executing platform (i.e., the Android OS).

4.2. Model checking

Model checking [58] is a verification technique that systematically explores the computation tree of a finite-state system t by looking for an execution path, called *counterexample*, witnessing the violation of a given formula φ. If no counterexample exists, then t satisfies φ, in symbols $t \models \varphi$. Otherwise, the model checker returns the discovered counterexample ω.

Model checking proved remarkably successful in a variety of application domains and it is currently routinely used to support the verification of both hardware and software components of real-world complexity. However, model checking large models is computationally very intensive. In particular, models grow extremely rapidly when allowing the parallel composition as we want to do for representing Android application configurations (see 4.1). This issue, known as the *state explosion problem* [10], is inherent to model checking. Also the adopted specification formalism matters. For instance, model checking LTL formulae is in PSPACE [59] (w.r.t. the size of models). The situation is even worse, i.e., the problem is n-EXPTIME complete for n-recursive schemes, when considering the modal μ-calculus [60]. This happens because the μ-calculus is an extremely expressive logic which allows for the definition of many security properties of interest [61]. More precisely, the μ-calculus is strictly more powerful than other modal logics including LTL and CTL [62]. Mitigation techniques are needed for making the approach scale on real numbers.

Here the equational μ-calculus is adopted. Equations systems have the same expressive power of the μ-calculus, i.e., for each μ-formula φ there exists an equivalent set of equations E_φ and vice versa. The syntax of the equations is as follows.

$$E, E' ::= X =_\mu A; E' \mid X =_\nu A; E' \mid \varepsilon$$
$$A, A' ::= T \mid F \mid \langle \alpha(\dot{x}) \rangle A \mid [\alpha(\dot{x})]A \mid \langle \tau \rangle A \mid [\tau]A \mid A \wedge A' \mid A \vee A' \mid X.$$

An equation system is a finite sequence (being ε the empty one) of equations $X =_\sigma A$ (with $\sigma \in \{\mu, \nu\}$) where X is a variable (appearing only once on the left-hand side of the equations) and A is a formula. A formula can be a truth value T or F, a variable X (which must appear on the left-hand side of some equation) and is built out of the following connectives: conjunction ($\wedge$), disjunction ($\vee$), existential modality ($\langle \cdot \rangle$), and universal modality ($[\cdot]$). Modalities are defined over parametric actions $\alpha(\dot{x})$, $\beta(\dot{y})$ (belonging to the alphabet $\Delta \cup \{\tau\}$ where τ is the silent action). Actions can be either inputs or outputs (input actions are denoted $\bar{\alpha}$ and it is possible to assume $\bar{\bar{\alpha}} \equiv \alpha$) and their parameters are resource names (e.g., Android URI) or variables x, y. Finally, each specification E must have an entry variable X, written $E \downarrow X$.

Given a STS $t = \langle i, S, \rightarrow \rangle$, the semantics of an equations system E, written $\llbracket E \rrbracket_\rho$ (where ρ is a mapping from variables to sets of states), associates to each variable X, appearing on the left-hand side of an equation $X =_\sigma A$ of E, a set of states of t satisfying the assertion A. The set of such states, in symbols $\llbracket A \rrbracket_\rho$, is inductively defined as

$$\llbracket T \rrbracket_\rho = S$$
$$\llbracket F \rrbracket_\rho = \emptyset$$
$$\llbracket X \rrbracket_\rho = \rho(X)$$
$$\llbracket A \wedge A' \rrbracket_\rho = \llbracket A \rrbracket_\rho \cap \llbracket A' \rrbracket_\rho$$
$$\llbracket A \vee A' \rrbracket_\rho = \llbracket A \rrbracket_\rho \cup \llbracket A' \rrbracket_\rho$$
$$\llbracket [\alpha(\dot{x})]A \rrbracket_\rho = \{s \in S \mid \forall s', \theta . s \xrightarrow{\phi, \alpha(\dot{y})} s' \text{ and } \theta \vdash [\dot{x} = \dot{y}]; \phi \text{ imply } s' \in \llbracket \theta A \rrbracket_\rho\}$$
$$\llbracket \langle \alpha(\dot{x}) \rangle A \rrbracket_\rho = \{s \in S \mid \exists s', \theta . s \xrightarrow{\phi, \alpha(\dot{y})} s' \text{ and } \theta \vdash [\dot{x} = \dot{y}]; \phi \text{ and } s' \in \llbracket \theta A \rrbracket_\rho\}$$
$$\llbracket [\tau]A \rrbracket_\rho = \{s \in S \mid \forall s', \theta . s \xrightarrow{\phi, \tau} s' \text{ and } \theta \vdash \phi \text{ imply } s' \in \llbracket \theta A \rrbracket_\rho\}$$
$$\llbracket \langle \tau \rangle A \rrbracket_\rho = \{s \in S \mid \exists s', \theta . s \xrightarrow{\phi, \tau} s' \text{ and } \theta \vdash \phi \text{ and } s' \in \llbracket \theta A \rrbracket_\rho\}.$$

Thus, a STS t satisfies a specification $E \downarrow X$ if and only if its initial state i is among those satisfying the entry equation, i.e., $i \in \llbracket E \rrbracket_{[]}(X)$ (where $[]$ is the empty mapping). The main difference with the original rules given in [11] is the presence of symbolic constraints in the rules for $[\alpha(\dot{x})]A$ and $\langle \alpha(\dot{x}) \rangle A$. Intuitively, symbolic names can be handled by means of a unifier θ. A unifier maps a name into another name or a value. Let us say that a unifier θ satisfies a constraint ϕ (in symbols $\theta \vdash \phi$) if, after applying θ, ϕ reduces to a tautology. Slightly abusing the notation, let us write θA to denote the formula obtained from A after applying θ to its names. Also notice that, θX has the side effect of creating in E a new equation (if it does not already exist) $X^\theta =_\sigma \theta A$ where $X =_\sigma A \in E$. The following example clarifies the rules given above.

Example 3. Consider the following system consisting of a single μ-equation.

$$E \triangleq X =_\mu \langle \tau \rangle X \vee [\mathtt{init(x, y)}]\langle \mathtt{createNewFile(x)} \rangle T.$$

Let t be the model of Fig. 4. To check whether $t \models (E \downarrow X)$ it is necessary to check if $s_1 \in \llbracket E \rrbracket_{[]}(X)$. A way to proceed is by applying the Tarski's fixed point theorem [63]. Let $U_0 = \emptyset$ and $\rho_0 = [X \mapsto U_0]$, then

$$\begin{aligned}
U_1 &= \llbracket \langle \tau \rangle X \vee [\mathtt{init(x, y)}]\langle \mathtt{createNewFile(x)} \rangle T \rrbracket_{\rho_0} \\
&= \{s \mid \exists s', \theta . s \xrightarrow{\phi, \tau} \text{ and } \theta \vdash \phi \text{ and } s' \in \emptyset = \llbracket X \rrbracket_{\rho_0}\} \\
&\cup \{s \mid \forall s', \theta . s \xrightarrow{\phi, \mathtt{init(z,w)}} s' \text{ and } \theta \vdash [z = x, w = y]; \phi \text{ imply } \\
&\quad s' \in \llbracket \theta \langle \mathtt{createNewFile(x)} \rangle T \rrbracket_{\rho_0}\} \\
&= \emptyset \cup \{s \mid \forall s', \theta . s \xrightarrow{\phi, \mathtt{init(z,w)}} s' \text{ and } \\
&\quad \theta \vdash [z = x, w = y]; \phi \text{ imply } \\
&\quad \exists s'', \theta' . s' \xrightarrow{\phi', \theta \mathtt{createNewFile(k)}} s'' \text{ and } \theta' \vdash \theta([k = x]; \phi') \text{ and } \\
&\quad s'' \in S = \llbracket T \rrbracket_{\rho_0}\}.
\end{aligned}$$

Intuitively, all the states of t having no outgoing transitions labeled with $\mathtt{init}(\cdots)$ satisfy the definition above (by definition of the universal quantifier) and, consequently, belong to U_1, which means $\{s_1, s_4, s_5, s_6, s_7\} \subseteq U_1$. Thus, let us restrict to verifying whether s_2 and s_3 belong to U_1 or not. For s_2 it must checked whether the following holds.

$$\forall\theta.\theta \vdash [x = \mathrm{f}, y = \texttt{"file://sdcard/f"}] \text{ implies}$$
$$\exists s'', \theta'.s_3 \xrightarrow{\phi', \mathtt{createNewFile(k)}} s'' \text{ and}$$
$$\theta' \vdash \theta([\mathrm{k} = x]; \phi') \text{ and } s'' \in S$$

However, this condition is trivially not satisfied. As a matter of fact, a counterexample can be found by setting $\bar{\theta} = \{\mathrm{f} \setminus x, \texttt{"file://sdcard/f"} \setminus y\}$. Although $\bar{\theta}$ satisfies the premise of the implication, the conclusion is false (there are no outgoing transitions labeled with $\mathtt{createNewFile}(\cdots)$ in s_3).

For what concerns s_3, it belongs to U_1 if and only if the following formula is satisfied.

$$\forall\theta.\theta \vdash [x = \mathrm{g}, y = \mathrm{t}] \text{ implies}$$
$$\exists s'', \theta'.s_4 \xrightarrow{\phi', \mathtt{createNewFile(k)}} s'' \text{ and}$$
$$\theta' \vdash \theta([\mathrm{k} = x]; \phi') \text{ and } s'' \in S$$

In this case there is a single candidate for s'', that is s_6. Hence, by instantiating the existential quantifier, the following result is obtained:

$$\forall\theta.\theta \vdash [x = \mathrm{g}, y = \mathrm{t}] \text{ implies}$$
$$\exists\theta'.\theta' \vdash \theta([\mathrm{f} = x, \mathrm{f} = \mathrm{g}]) \text{ and } s_6 \in \emptyset = S.$$

It can be trivially observed that, for every $\bar{\theta}$, $\bar{\theta} \vdash [x = \mathrm{g}, y = \mathrm{t}]$ iff $\bar{\theta}(x) = v = \bar{\theta}(\mathrm{g})$ and $\bar{\theta}(y) = u = \bar{\theta}(\mathrm{t})$. Then, assuming $\bar{\theta}(\mathrm{f}) = w$, $\bar{\theta}([\mathrm{f} = x, \mathrm{f} = \mathrm{g}]) = [w = v, w = u]$ which is satisfied by any $\bar{\theta}'$ assigning the same value to u, v and w. This suffices to state that $s_3 \in U_1$. Thus, $U_1 = \{s_1, s_3, s_4, s_5, s_6, s_7\}$.

Let us iterate the previous reasoning to compute U_2 by setting $\rho_1 = [X \mapsto U_1]$. Following the same approach used above, it is possible to infer that $\{s_1, s_4, s_5, s_6, s_7\} \subseteq U_2$. Again, it reduces to checking s_2 and s_3. The two cases are analogous to the previous step and it is possible to conclude that only $s_3 \in U_2$.

Since $U_2 = U_1$ a fixed point have been found and, since $s_1 \in U_2$ it is possible to conclude that $t \models E$. □

4.3. Symbolic Partial Model Checking

In [11] Partial Model Checking (PMC) was proposed as a technique for extending the applicability of model checking to large models consisting of the parallel composition of simpler ones. It consists of a partial evaluation procedure which specializes a formula of the μ-calculus against a model. Here an extension is provided to support symbolic analysis. Formally, the partial evaluation corresponds to applying a *quotienting* operator $E' = E/\!\!/_{L,M}\, t$ where E is a specification, t is a STS, L and M are sets of (input and output) actions/channels that t uses for synchronizing with other agents. L is the set of restricted actions and M is the set of input/output ones (thus t can only synchronize through the actions of $\bar{L} \cap M$—where $\bar{L}$ stands for $\Lambda \setminus L$). Notice that, in this context $L = \bar{\Delta}$ and $M = \Delta \cup \Gamma \cup \bar{\Gamma}$. Specifications E, E' are lists of (least μ and greatest ν) fixed point equations.

The quotienting operator originally presented in [11] was defined for being applied to LTS models. Here we extend such definition for dealing with STS models. The new rules are given in Table 1.

The partial evaluation of a specification E is defined by induction. As expected, (i) the partial evaluation of a specification $(E \downarrow X)$ begins from its entry variable X. Clearly, (ii) if the system of equations is empty it remains unchanged. Otherwise, (iii) each equation generates a list of new ones, one for each state s_i of t. The right-hand side of these new equations is the partial evaluation of the original formula A against the states $s_0, \ldots, s_n$. Rules from (iv) to (xii) are for the partial evaluation of formulae against a state s. Thus, (iv) the partial evaluation of a variable X against a state s consists in generating a fresh new variable X_s. According to (v), a modality $\langle\alpha(\dot{x})\rangle A$ generates a disjunction including one modality-free disjunct for each $\alpha(\dot{y})$ transition starting from the node s and one keeping the modality (representing the fact that also t' could perform $\alpha(\dot{x})$). For each of the transition $s \xrightarrow{\phi, \alpha(\dot{y})} s'$, the formula A is both rewritten by applying the name substitution θ where θ is the *most general unifier* (*mgu*) of $[\dot{x} = \dot{y}]; \phi$ and partially evaluated against s'. Briefly, the *mgu* is the smaller substitution which satisfies a set of constraints (the interested reader can refer to [64]). The case (vi) for $\langle\tau\rangle A$ is similar. However, since τ actions can arise from synchronous communications, new branches modeling such behavior are added. In particular, for each input (output) α that t might receive (produce, respectively), a modality for the complementary action $\bar{\alpha}(y)$ is added (where y is a fresh variable name). Again, this operations requires to rewrite A through the $mgu\theta'$. The $[\alpha(\dot{x})]A$ and $[\tau]A$ cases, i.e., (vii) and (viii), behave symmetrically to the previous two. Finally, the partial evaluation of a disjunction/conjunction, (ix) and (x), is the disjunction/conjunction of the partial evaluation of its branches and truth values, (xi) and (xii), are unaffected by the partial evaluation.

The main property of PMC is that it transforms an instance of the model checking problem involving many, concurrent models into a new one with a single model. Such property is stated by the following theorem (which extends that given in [11]).[5]

Theorem 1. *If $E' = E/\!\!/_{L,M}\, t$, then for all t' holds that $t' \parallel_{L,M} t \models E$ if and only if $t' \models E'$.*

Theorem 1 states that, for every t', model checking $t' \parallel_{L,M} t$ (i.e., the parallel composition of the two STS) against E can be reduced to model checking t' against E'. In the following L, M are not used when clear from the context. Since the size of t' is usually much smaller than the size of $t' \parallel t$, verifying $t' \models E'$ requires

Table 1
Symbolic PMC quotienting operator.

(i)	$(E \downarrow X)/\!\!/_{L,M}\, t = (E/\!\!/_{L,M}\, t) \downarrow X_{s_0}$ where $t = \langle s_0, \{s_0, s_1, \ldots, s_n\}, \rightarrow\rangle$
(ii)	$\varepsilon/\!\!/_{L,M}\, t = \varepsilon$
(iii)	$(X =_\sigma A; E)/\!\!/_{L,M}\, t = (X_{s_0} =_\sigma A/\!\!/_{L,M}\, s_0; \cdots; X_{s_n} =_\sigma A/\!\!/_{L,M}\, s_n; E/\!\!/_{L,M}\, t)$
(iv)	$X/\!\!/_{L,M}\, s = X_s$
(v)	$\langle\alpha(\dot{x})\rangle A/\!\!/_{L,M}\, s = \langle\alpha(\dot{x})\rangle(A/\!\!/_{L,M}\, s) \vee \bigvee_{s \xrightarrow{\phi, \alpha(\dot{y})} s'} (\theta A/\!\!/_{L,M}\, s')$ with $\theta = mgu([\dot{x} = \dot{y}]; \phi)$
(vi)	$\langle\tau\rangle A/\!\!/_{L,M}\, s = \langle\tau\rangle(A/\!\!/_{L,M}\, s) \vee \bigvee_{s \xrightarrow{\phi, \tau} s'} (\theta A/\!\!/_{L,M}\, s')$ $\vee \bigvee^{\alpha \in \bar{L} \cap M}_{s \xrightarrow{\phi', \alpha(\dot{x})} s'} \langle\bar{\alpha}(y)\rangle(\theta' A/\!\!/_{L,M}\, s')$ with y fresh and $\theta = mgu(\phi)$ and $\theta' = mgu([\dot{x} = \dot{y}]; \phi')$
(vii)	$[\alpha(\dot{x})]A/\!\!/_{L,M}\, s = [\alpha(\dot{x})](A/\!\!/_{L,M}\, s) \wedge \bigwedge_{s \xrightarrow{\phi, \alpha(\dot{y})} s'} (\theta A/\!\!/_{L,M}\, s')$ with $\theta = mgu([\dot{x} = \dot{y}]; \phi)$
(viii)	$[\tau]A/\!\!/_{L,M}\, s = [\tau](A/\!\!/_{L,M}\, s) \wedge \bigwedge_{s \xrightarrow{\phi, \tau} s'} (\theta A/\!\!/_{L,M}\, s')$ $\wedge \bigwedge^{\alpha \in \bar{L} \cap M}_{s \xrightarrow{\phi', \alpha(\dot{x})} s'} [\bar{\alpha}(y)](\theta' A/\!\!/_{L,M}\, s')$ with y fresh and $\theta = mgu(\phi)$ and $\theta' = mgu([\dot{x} = y]; \phi')$
(ix)	$A_1 \vee A_2/\!\!/_{L,M}\, s = A_1/\!\!/_{L,M}\, s \vee A_2/\!\!/_{L,M}\, s$
(x)	$A_1 \wedge A_2/\!\!/_{L,M}\, s = A_1/\!\!/_{L,M}\, s \wedge A_2/\!\!/_{L,M}\, s$
(xi)	$F/\!\!/_{L,M}\, s = F$
(xii)	$T/\!\!/_{L,M}\, s = T$

[5] The proof is provided in the Appendix.

a smaller search space. It must be noted that the application of PMC does not affect the worst-case complexity of the problem. As a matter of fact, by using Theorem 1 one trades a smaller model for a longer specification. Nevertheless, as outlined in [11], several simplification techniques can be carried out on E'. As a result, in many practical cases the size of E' grows more slowly than the size of $t' \parallel t$ and this leads to better performances.

Example 4. Consider again the specification of Example 3. We define L and M as follows.[6]

$$L = \{\texttt{init}, \texttt{createNewFile}\}$$
$$M = \{\overline{\texttt{sendBroadcast}}, \texttt{startService}\}.$$

The partial evaluation of $E \downarrow X$ against t generates the following equation system.

$$E' \triangleq \begin{cases} X_{s_1} =_\mu \langle\tau\rangle X_{s_1} \vee \langle\texttt{sendBroadcast}(\texttt{x}_\texttt{c}, \texttt{x}_\texttt{i})\rangle X^{\theta}_{s_2} \\ \quad \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ X_{s_2} =_\mu \langle\tau\rangle X_{s_2} \vee ([\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \wedge \langle\texttt{createNewFile}(\texttt{f})\rangle T) \\ X_{s_3} =_\mu \langle\tau\rangle X_{s_3} \vee ([\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \wedge (\langle\texttt{createNewFile}(\texttt{g})\rangle T \vee T)) \\ X_{s_4} =_\mu \langle\tau\rangle X_{s_4} \vee [\texttt{init}(\texttt{x}, \texttt{y})](\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \vee \langle\texttt{createNewFile}(\texttt{f})\rangle T) \\ X_{s_5} =_\mu \langle\tau\rangle X_{s_5} \vee [\texttt{init}(\texttt{x}, \texttt{y})](\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \vee \langle\texttt{createNewFile}(\texttt{g})\rangle T) \\ X_{s_6} =_\mu \langle\tau\rangle X_{s_6} \vee \langle\overline{\texttt{startService}(\texttt{y}_\texttt{c}, \texttt{y}_\texttt{j})}\rangle X^{\theta'}_{s_7} \\ \quad \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ X_{s_7} =_\mu \langle\tau\rangle X_{s_7} \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ X^{\theta}_{s_2} =_\mu \langle\tau\rangle X^{\theta}_{s_2} \vee ([\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \wedge \langle\texttt{createNewFile}(\texttt{f})\rangle T) \\ X^{\theta'}_{s_7} =_\mu \langle\tau\rangle X^{\theta'}_{s_7} \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \end{cases}$$

where $\theta = \{\texttt{c} \setminus \texttt{x}_\texttt{c}, \texttt{i} \setminus \texttt{x}_\texttt{i}\}$ and $\theta' = \{\texttt{c} \setminus \texttt{y}_\texttt{c}, \texttt{j} \setminus \texttt{y}_\texttt{j}\}$. □

As shown by Example 4, PMC can increase the size of the original specification. This phenomenon can be effectively countered by applying the following simplifications proposed in [11] (and slightly modified to cope with symbolic actions).

- **Simple Evaluation** (SE), replace instances of $A \vee T$ and $A \wedge F$ with T and F, respectively, and replace $A \vee F$ and $A \wedge T$ with A;
- **Reachability Analysis** (RA), remove equations that are unreachable from the entry variable;
- **Constant Propagation** (CP), remove equations of the form $X =_\sigma T$ and $X =_\sigma F$ and replace all the instances of X with the corresponding truth value;
- **Unguardedness Removal** (UR), collapse groups of variables that do not appear in modal assertions;
- **Trivial Equation Elimination** (TEE), replace $X =_\sigma \langle\alpha(\dot{x})\rangle F$ and $X =_\sigma [\alpha(\dot{x})]T$ with $X =_\sigma F$ and $X =_\sigma T$, respectively;
- **Equivalence Reduction** (ER), collapse equations whose right-hand side assertions are equivalent;
- **Implication Reduction** (IR), collapse the right-hand side of two equations when one implies the other.

Example 5. Let E' be as in Example 4. The following simplifications can be applied. Eq. (3) can be reduced to $X_{s_3} =_\mu \langle\tau\rangle X_{s_3} \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T$ (by applying SE two times).

Listing 1: Verification Procedures for App Configurations.

```
Procedure VerifyConfV1(App $a$, DevID $d$)
  output: OK or Trace $\sigma$
  effect: modify CDB

  $t_a$ := extract_model($a$)
  $t$ := CDB.get_configuration($d$)
  $E$ := SYSTEM_POLICY
  $\sigma$ := model_check($t_a \| t, E$)
  if $\sigma \neq$ NO_TRACE then
    return $\sigma$
  else
    CDB.set_configuration($d, t_a \| t$)
    return OK
  endif
End

Procedure VerifyConfV2(App $a$, DevID $d$)
  output: OK or Trace $\sigma$
  effect: modify PDB

  $t_a$ := extract_model($a$)
  $E$ := PDB.get_policy($d$)
  $\sigma$ := model_check($t_a, E$)
  if $\sigma \neq$ NO_TRACE then
    return $\sigma$
  else
    $L$ := $\{\alpha, \bar{\alpha} \mid \alpha$ is an Android IPC action$\}$
    $M$ := $\{\beta \mid \beta \in E\}$
    $E^*$ := quotient($E, L, M, t_a$)
    $E^*$ := simplify($E^*$)
    PDB.set_policy($d, E^*$)
    return OK
  endif
End
```

Also, by applying RA it derives that from X_{s_1} only $X^{\theta}_{s_2}$ is reachable. Thus, the following system of equations is obtained:

$$E'' \triangleq \begin{cases} X_{s_1} =_\mu \langle\tau\rangle X_{s_1} \vee \langle\texttt{sendBroadcast}(\texttt{x}_\texttt{c}, \texttt{x}_\texttt{i})\rangle X^{\theta}_{s_2} \\ \quad \vee [\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ X^{\theta}_{s_2} =_\mu \langle\tau\rangle X^{\theta}_{s_2} \vee ([\texttt{init}(\texttt{x}, \texttt{y})]\langle\texttt{createNewFile}(\texttt{x})\rangle T \\ \quad \wedge \langle\texttt{createNewFile}(\texttt{f})\rangle T). \quad \square \end{cases}$$

5. BYODroid, a prototype Secure Meta-Market

BYODroid[7] is the first, prototype implementation of a Secure Meta-Market for Android devices and supports the specification, verification and enforcement of fine-grained security policies over a network of federated devices. BYODroid v.1 has been presented in previous work [7]. Here the focus is on the latest version of BYODroid, namely BYODroid v.2, which unlike BYODroid v.1, features a verification procedure leveraging the PMC techniques discussed in Section 4.3.

The verification procedures implemented in BYODroid v.1 and v.2, `VerifyConfV1` and `VerifyConfV2` respectively, are outlined in Listing 1. The verification procedure `VerifyConfV1` extracts the model t_a from application a. Then, it retrieves the current configuration of device d, stored as an STS t in a configuration database CDB, and the given policy E. Then model checking is applied to verify whether $t_a \parallel t \models E$. If a counterexample trace σ is found, then the procedure terminates by returning it. Otherwise, the configuration of d is extended with (the model of) a and the algorithm terminates.

[6] For the sake of presentation, the two sets only contain the minimal number of elements necessary for the evaluation. Actual executions include in L all the security-relevant operations and in M all the Android IPC methods. Both would have no effect on the current example.

[7] http://csec.it/software/byodroid/index.html.

Similarly to `VerifyConfV1`, the verification procedure `VerifyConfV2` extracts a model t_a from the application a and retrieves the current policy for d, namely E, from the policy database `PDB`. Then, model checking is applied to check whether $t_a \models E$. If counterexample trace σ is discovered, then it is returned and the procedure terminates. Otherwise PMC is applied. To this end, the procedure builds sets L and M (cf. Section 4.3) and then applies the quotienting operator $E /\!\!/_{L,M}\, t_a$. Then, simplifications are applied until a fixed point, i.e., a policy E^* admitting no further reduction, is reached. Finally, `PDB` is updated with E^* and the algorithm terminates.

Other techniques implemented in BYODroid are discussed in the following paragraphs.

Model checking. Several model checkers have been proposed in the literature (see [65–67] for surveys on model checking tools and their application to some problems of interest). Among them, Spin [68] is a major proposal with a long-standing tradition in software analysis and verification [69]. Pragmatically, Spin offers some advantages when coming to applications. As a matter of fact, it has been effectively exploited in industrial processes and products (e.g., see [70]). Moreover, it has been recently extended for supporting concurrent, multi-core verifications. For these reasons Spin version 6.2.5 [71] has been adopted as model checking engine.

Spin is an LTL model checker for the Promela specification language. Unlike other model checkers, e.g., mCRL2 [72], Spin does not natively support the μ-calculus. Thus, formulae must be translated into agents which reach a faulty state whenever the corresponding policy is violated. To do that we follow the reasoning presented in [73] and [74] for mapping formula satisfiability into (bi)simulation relation between agents. Then, we follow the procedure defined in [75] to append the agent to the original Promela specification obtained from the application model (see Section 4.1). Notice that our implementation significantly simplifies the general version of [75]. As a matter of fact, we do not need to implement a generic framework for checking relations between agents. Under our assumptions, we only need to check a precise relation, i.e., strong simulation. Moreover, we do not need to be parametric with respect to the observable channels/actions as they are statically defined as the elements of Δ (see Section 4.1).

The verification using Spin consists of three steps. First, Spin generates a C source file, called *pan.c*, implementing the verifier for the given agents and specification. Then, a C compiler is used to obtain the verification program (i.e. PAN) which is finally executed. The adopted compiler is gcc version 4.6.3.

The main difference between BYODroid v.1 and v.2 resides in the instance of the model checking problem being processed by Spin. As a matter of fact, BYODroid v.1 does not include PMC. Hence, it verifies a configuration $a_1, \ldots, a_n$ (against a specification E) by directly solving $t_{a_1} \| \cdots \| t_{a_n} \models E$. The differences between the two approaches for the verification of configurations are also reported in Listings 1 (see below).

Other model checkers can be considered for the implementation of the SMM. For instance, nuSMV [76] (and its extension nuXMV) relies on a symbolic representation of states. This approach can actually reduce the problem search space in many cases. Another possibility is provided by CADP [77]. Briefly, CADP is a toolbox including several modules for the design and analysis of software. Also, it includes a model checker, namely the evaluator module, natively accepting μ-calculus specifications [78]. A further candidate is mCRL2 [72] which has the desirable features of directly handling μ-calculus specifications. These alternatives require further investigation and a systematic evaluation which lay out of the scope of the present work.

Model extraction. Model extraction, being exploited in both BYODroid v.1 and v.2, consists of few steps. CFGs are generated by means of (a slightly modified version of) Androguard 1.3 [79]. Androguard has been adopted since it is an off-the-shelf technology for Android (while ConFLEx would require to be adapted from pure Java). The extraction process produces a distinct CFG for each method of the target code. These graphs are simplified, as detailed in Section 4.1, in order to reduce their size by removing irrelevant paths. Moreover, information about internal application invocations, i.e., how methods of the application invoke each other, is maintained. Such information is then used for the CFGs of the application components. The list of existing components is obtained by inspecting the implemented interfaces. For each of them, a general CFG is created by starting from interface methods, e.g., `startActivity` for activities. If the CFG of such methods contains invocations to other parts of the application, the corresponding sub-graphs are attached. Each of these CFG is then converted to a corresponding STS as explained in Section 4.1.

Partial model checking. The PMC engine is included in BYODroid v.2. Only recently partial model checking tools have been presented. At the best of our knowledge, two implementations exist. CADP has been extended with a (non symbolic) PMC component [80]. Similarly, the tool *formulaquotient* implements a PMC algorithm for mCRL2 specifications. At the best of our knowledge, these tools are in a preliminary development stage and their applicability to our environment needs further investigation. For the implementation of BYODroid v.2 we opted for developing a PMC engine from scratch. The tool is a straight implementation of the algorithm described in Section 4.3.

As explained in Section 4.3, simplification play a crucial role for the effectiveness of PMC. Apart from the Implication Reduction (which relaxes the specification and may therefore introduce false positives), all the simplifications presented above are implemented in BYODroid v.2.

6. Experiments

This section presents an extensive experimental evaluation aimed at assessing both the effectiveness and the feasibility of the proposed approach. Previous papers reported preliminary experimental results confirming the feasibility of some components in isolation (e.g., model extraction and model checking) and applied to individual applications. Here, the whole verification chain is considered, namely model extraction, model checking and (symbolic) PMC. Experiments belong to two distinct categories dedicated to testing *effectiveness* and *performances/scalability*.

6.1. Experimental setup

To evaluate the performances of the prototype the two versions of BYODroid described in Section 5 are tested against a large number of application configurations and a real-world BYOD policy. Instead, to show that the prototype can effectively detect policy violating applications, the evaluation focuses on a smaller set of applications. In particular, a few dozens of applications which were very likely to violate the security policy (see below for details) have been identified.

Security policy. Each app configuration was tested against a security policy drawn by the BYOD security guidelines of the White House [81]. The policy consists of few rules stating how the mobile devices should access and manage the resources of the organization. Albeit simple, it is a significant excerpt of a real-world BYOD security policy. Informally, it can be summarized as follows:

(R1) Never download business data on the mobile device.
(R2) Always delete locally stored sensitive data (e.g., emails).
(R3) Never transfer data to non-agency devices.

The policy can be seen as restricting the usage of some security-relevant APIs[8] and can be formally recast to the ConSpec specification in Listing 2.[9] ConSpec [50] is a specification language that has been designed to specify security policies and contracts and it has been already exploited for the security enforcement on mobile environments (e.g., see [31]). For instance (R1), data download consists of network requests, e.g., HTTP-GET, for some critical URL, e.g., https://agency.gov/, followed by a file system writing action. The involved Java APIs are `java.net.URL.¡init¿(String s)`, which creates an URL object from a string, `java.net.URL.openConnection()`, which sets up a connection to the given URL, and `java.io.FileOutputStream.write(...)`, which writes data in a given file object. The sequence of these actions may cause a policy violation. To represent the security state, which changes as a consequence of the observed actions, the policy exploits *state variables*. The syntax of ConSpec statements and expressions resembles that of most imperative programming languages. The policy uses a three-variable state (`agency_host`, `agency_url`, and `connected`). Variables are labeled as **SESSION** as they only refer to a single execution of a single program (as opposed to **MULTISESSION** and **GLOBAL**). Policy rules refer to events, i.e., guarded API invocations, and can be activated **BEFORE** or **AFTER** the event is observed. When this happens, the clauses below the rule are evaluated and, if their guards are satisfied, executed. For instance, the first rule says that, after observing a `java.net.URL.<init>(String addr)` operation, the policy evaluates whether the url address contains the string `"agency.gov/"`. If this is the case, the returned `URL` object is stored for checking its usage in future events. Otherwise (**ELSE**) nothing happens (**skip**). Similarly, the second rule states that, before permitting a `java.net.URL.openConnection()` invocation, the policy compares the current url **this** with that stored in variable `agency_url`, e.g., as a result of the application of the previous rule. If the two objects are equal, the policy assigns **true** to `connected`. Otherwise, the state is not changed. Finally, notice that the third rule has a single guard for the case `!connected`. This implies that the policy is violated, since no transition is allowed, whenever the rule is triggered in a state such that `connected = `**true**.

The ConSpec policy showed above is then translated into a corresponding set of μ-equations. This process is always possible as the μ-calculus is more expressive than ConSpec. Briefly, the conversion proceeds by creating a μ-equation for each possible state of the ConSpec policy. The left-hand side of the equation is a variable $X_{[\bar{v}]}$ representing a state of the policy, i.e., an assignment of values $\bar{v}$ to its variables.[10] Hence, the right-hand side is obtained by a finite disjunction of the type $\bigvee \langle \alpha(\bar{u}) \rangle X_{[\bar{w}]}$ where α is the action labeling the ConSpec rule,[11] $\bar{u}$ are the values to the parameter (including the target object) of action α and $\bar{w}$ are the values after the application of the rule. The disjunction only contains variables corresponding to (states reachable through) the guards which are satisfied by the values $\bar{v}$ and $\bar{u}$. Since variables can only appear once in the left-hand side of an equation system, all the equations of type $X =_\mu A_i$ are collapsed in $X =_\mu \bigvee_i A_i$. The entry variable of the equation system is the $X_{[\bar{v}]}$ such that $\bar{v}$ are the values of the initial assignments. To illustrate, consider the second rule of the block `(R1)` and the initial state of the White House policy. They result in the equation

$$X_{[\mathtt{agency.gov/},\mathtt{null},\mathtt{false}]} =_\mu \bigvee_{i \in \{1,\ldots,k\}} \langle \mathtt{openConnection}(\mathtt{o_i}) \rangle \times X_{[\mathtt{agency.gov/},\mathtt{null},\mathtt{false}]}$$

Listing 2: ConSpec encoding of US government BYOD rules.

```
SECURITY STATE
  SESSION Str agency_host = "agency.gov/";
  SESSION Obj agency_url = null;
  SESSION Bool connected = false;
/* (R1) Never download business data
on the mobile device */
AFTER Obj url = java.net.URL.<init>(Str addr) PERFORM
  (addr.contains(agency_host)) -> { agency_url := url; }
  ELSE -> { skip; }
BEFORE java.net.URL.openConnection() PERFORM
  (this.equals(agency_url)) -> { connected := true; }
  ELSE -> { skip; }
BEFORE java.io.FileOutputStream.write(Nat i) PERFORM
  (!connected) -> { skip; }
/* (R2) Always delete locally stored sensitive data */
AFTER Obj file = java.io.File.createTempFile() PERFORM
  (true) -> { file.deleteOnExit(); }
/* (R3) Never transfer data to non-agency devices */
BEFORE android.bluetooth.BluetoothSocket.getOutputStream()
  PERFORM
  (!connected) -> { skip; }
```

where $\{\mathtt{o_1}, \ldots, \mathtt{o_k}\}$ is the finite domain for **this**.[12] Indeed, since it never holds that `oi .equals(`**null**`)`, the only satisfied guard is **ELSE**. As the statement **skip** does not cause state changes, it results in a self loop on variable $X_{[\mathtt{agency.gov/},\mathtt{null},\mathtt{false}]}$.

The resulting equation system consists of 4 μ-equations for a total size of 64. The size of a policy is computed as the number of terminal symbols, i.e., constants, variables and modalities, appearing in a specification. For instance, the equation given above has size $2k$, as it consists of a disjunction of k formulae of size 2.

App configuration testbed. To assess the scalability of the proposed approach, Android applications drawn from the Google Play service have been used. The sheer size of the store, which contains more than 1 million applications [82], does not permit an exhaustive analysis of all possible configurations. A smaller, but still interesting set is represented by the "top free chart" applications, which contains the most popular free applications in Google Play.[13] This includes over 800 applications from a number of heterogeneous categories, e.g., entertainment, productivity, and games. From this set of applications, six sets of 30 applications each have been randomly selected. For any such set of applications $\mathcal{C} = \{a_1, \ldots, a_{30}\}$ the following 30 configurations $C_k = \{a_1, \ldots, a_k\}$ for $k = 1, \ldots, 30$ have been considered. This results in a total of 180 app configurations that have been used in the experiments. It must be noted that the size of the largest considered configurations (i.e. 30) exceeds the average number of applications installed on mobile devices which statistics[14] set to 26.

For what concerns the effectiveness of the policy violation detection procedure, two sets N and B are identified. They contain applications using the network and the bluetooth interfaces, respectively. Network and bluetooth usage can be predicted by considering the permissions requested by an application, that is, by checking whether `android.permission.INTERNET` and `android.permission.BLUETOOTH` appear in the application manifest. Then, the configurations in $N \times B$, i.e., all the possible

8 For brevity only few of them are considered here.

9 The syntax has been slightly simplified for the sake of readability.

10 Notice that this step is feasible since ConSpec uses finite type domains.

11 More precisely, special actions α_B and α_A are introduced for discriminating between **BEFORE** and **AFTER**.

12 Assuming $\{\mathtt{o_1}, \ldots, \mathtt{o_k}, \mathtt{null}\}$ to be the domain for objects, it is possible to always force the variable **this** $\neq$ **null** as the self reference must exist inside a when invoking non static methods.

13 https://play.google.com/store/apps/collection/topselling_free (accessed on September 30, 2014).

14 http://www.statista.com/chart/1435/top-10-countries-by-app-usage/ (accessed on September 15, 2014).

pairs of applications, have been submitted to the prototypes. Finally, reverse engineering techniques have been applied and the configurations have been manually tested to confirm or reject the reported violations.

Evaluation criteria. The experiments have been run using a Dell Optiplex 9010 server with an Intel Core i7 3.40 GHz and 16 GB of RAM, equipped with Java JDK version 1.7. The time out for PAN verification was set to 20 min.

The main goal of the experiments on scalability is to highlight the advantage of including PMC in the verification process. Hence, experiments must compare the behavior of the two procedures of Listing 1, i.e., the algorithms used for the implementation of BYODroid v.1 and v.2. In particular, three aspects are crucial:

Memory usage. Model checkers using an explicit representation of the state space (as Spin does) can rapidly occupy the available memory. More applications means larger models and increased memory usage.

Execution time. Model checking requires a significant time span for validating an application. Also, other tools involved in the verification chain may cause further delay. Understanding how the tools contribute to the whole execution time can help in identifying bottlenecks and possible optimizations.

Policy growth. The application of PMC may increase the side of the policies. This is countered by the simplification techniques outlined in Section 4.3 but the effectiveness of the simplification must be ascertained.

Instead, for the effectiveness of the approach, only BYODroid v.2 has been tested. For each execution the evaluation result (i.e. whether there is a violation or not) is reported. Then, the violation is tested to check whether it can be actually be replicated or not, in order to point out false positives. Also, to show the actual behavior of a configuration violating the security policy, reverse engineering and manual code inspection have been carried out.

6.2. Experimental results

Briefly, tests covered the automatic model generation process and the verification of configurations. In particular, the verification has been carried out with the two versions of BYODroid. The two executions are then compared for highlighting the impact of PMC on the proposed solution.

Model extraction. Model size and extraction time are affected by the security policy. For BYODroid v.1, it is reasonable to expect that for any given policy, the size of models grows (exponentially) with the size of the app configuration. However, it must be noted that models undergo several manipulation steps aiming at reducing their size. Moreover, Spin internally applies several reduction techniques which often lead to even more compact representations. Fig. 5 shows the (geometric) mean of the number of states visited by Spin during the verification of models. This value is indicative of the actual size of the search space associated to the model. Clearly, when time out occurs, the value only corresponds to the states explored by the model checker before the forced termination. For this reason, Fig. 5 is truncated when all the verifications result in a time out (i.e., for configurations of more than 11 apps). As expected, the number of states tends to grow very rapidly (notice that Y axis is in logarithmic scale) and the expected exponential trend can be observed.

Model verification. When applying BYODroid v.1, verification of models $t_1, \ldots, t_n$ against policy E amounts to solving $t_1 \parallel \cdots \parallel t_n \models E$. Instead, with BYODroid v.2, it amounts to verifying whether $t_n \models ((E /\!\!/_{L,M} t_1) \cdots /\!\!/_{L,M} t_{n-1})$. The scatter plots in Fig. 6 provide a comparative analysis of the two approaches,

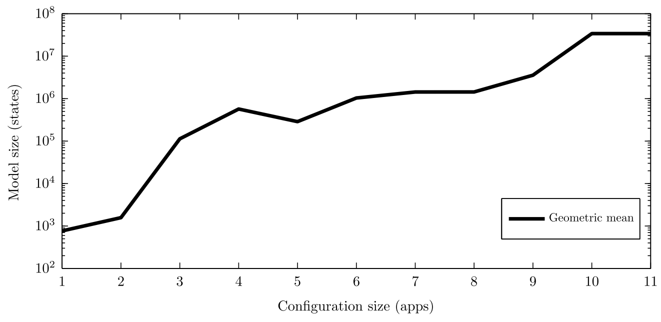

Fig. 5. Size of models generated from groups of applications.

i.e., verification with v.1 and v.2. The left plot compares the required verification time for each app configuration considered. The right plot compares the number of states actually stored and visited by the model checker. Symbolic PMC leads to a substantial improvement of both the computation time and the size of the search space. This improvement is even more evident as the size of models increases, since they require longer verification times. Also notice that several points are clustered on the top edge of the graph area since, for the corresponding experiments, the verification with PMC terminates successfully, while BYODroid v.1 leads to a time out.[15]

The trend of the execution time for the two approaches is reported in Fig. 7. The two lines correspond to the average times for running the verification program, i.e., the PAN executable, with (solid line) and without (dashed line) PMC. The dotted line indicates the time out threshold (i.e., 1200 s). Lines are truncated where experiments result in time outs only: 12 for MC-only and 21 for PMC. Although both of them show an exponential trend, using PMC substantially postpones the cut-off point corresponding to the time limit. This allows the system to process significantly (75%) larger configurations.

Partial policy evaluation. PMC preprocessing consists of the application of the quotienting operator and the simplifications as described in Section 4.3. As previously outlined, the growth of the policy size is a crucial aspect. Fig. 8 shows the growth of the size of the policy. The average value is represented by the solid line, while vertical bars represent the whole range of results for the experiments with the given configuration size.

Time distribution per technique. As mentioned in Section 6.1, application verification is the result of the application of a tool chain, i.e., Spin, gcc and PAN. On large models, verification tends to become the dominating factor. Nevertheless, understanding the distribution of the verification time over these three steps can be useful for discovering bottlenecks and possible optimizations. Fig. 9 reports the percentage of time taken by the verification steps, i.e., black for Spin, white for gcc and gray for PAN. Times for the approach without PMC are not reported as, compared to the verification time, Spin and gcc times are negligible.

Interestingly, PMC reduces the impact of PAN verification over the total time. Still, when models grow, it is responsible of the main computational effort. This suggests that, even though further optimizations of Spin and gcc could lead to better performances, on the long run, PAN verifier remains the main bottleneck.

Policy violations and false results. Let us consider the 100 configurations obtained by pairing the elements of N with those of B (with $\#N = \#B = 10$). As expected many of them were recognized as illegal. The 38 policy-violating pairs are labeled with ✗and ↯(while ✓ denotes the configuration passing the verification).

[15] Marks in the top-right corner correspond to time outs of both v.1 and v.2.

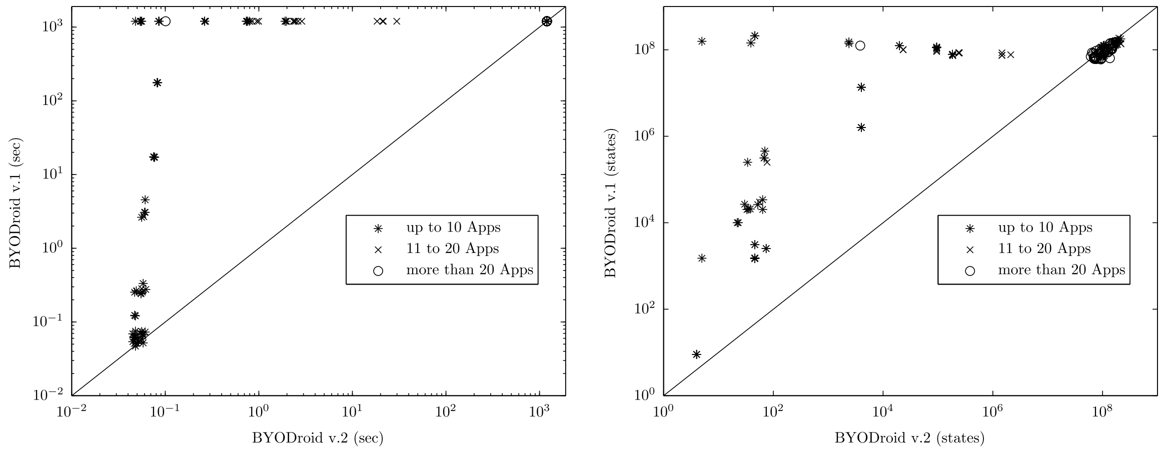

Fig. 6. Comparison of the verification time and search space for BYODroid v.1 and v.2.

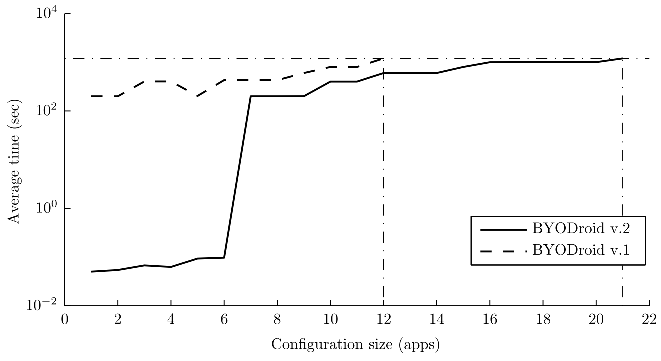

Fig. 7. Comparison of the execution times of the PAN verifiers launched by BYODroid v.1 and v.2.

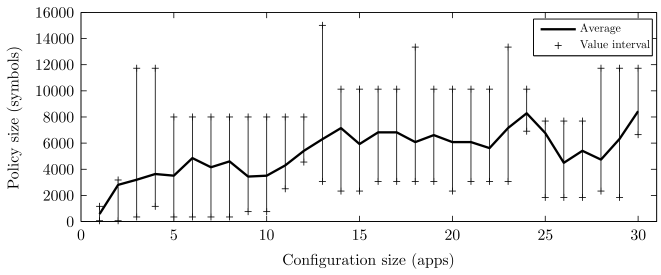

Fig. 8. Policy growth upon PMC application.

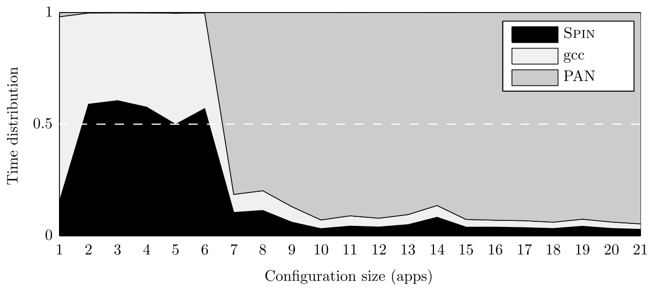

Fig. 9. Distribution of execution times for verification with BYODroid v.2.

For the validation of the results, we manually run a decompiler to reconstruct the source code of the 38 policy-violating configurations and we identified the components invoking the security-relevant APIs. In some cases it is possible to identify that the policy is not actually violated by the considered configurations, i.e., there are false positives. These 13 (34.2%) cases have been marked with ⚡̸.

The reasons of the false positives are manifold, but it is possible to identify at least two factors.

1. Some applications do not connect to any URL, but they include the Google Ads Library. Google Ads use the network to establish a connection with a remote server owned by Google. Reasonably, that library cannot be exploited to access arbitrary network locations, but the analysis fails in discovering it.
2. One of the applications of a pair uses IPC in a way that actually prevents the interaction with the second one. Again, this behavior derives from the over-approximation generated by the modeling process.

The experimental results are summarized in Table 2.

Checking the absence of false negatives is more tricky. Indeed, positive results, i.e., detected policy violations, have the side effect of indicating some, suspicious components that one can use for aimed testing. Instead, negative ones produce no similar information. As a consequence, searching false negatives would require to execute extensive penetration testing on all the application pairs. That would be a cumbersome task and, still, since testing is not exhaustive by definition, the result itself could suffer from false negatives. As discussed above, false negatives can only arise when unsafe models are generated. Although possible, there are strong, yet informal, guarantees that CFGs correctly represent the actual behavior of programs (see Section 2).

As a further step, we report the policy violation occurring on a specific configuration, i.e., the pair composed by the *Dolphin* browser and the *Bluetooth Transfer Any File* (BTAF) utility. By means of existing platforms for reverse engineering and code analysis, i.e., Dexter[16] and Maveric [83], it has been possible to inspect the structure of the applications. Although the code consists of 7095 (mostly obfuscated) Java files, it has been possible to identify few relevant classes and their relationships (reported in Fig. 10).

As expected, the Dolphin browser allows one to connect to an arbitrary URL, being accessed through a `openConnection()` call. Then, from the main activity, the user can press a button "Share" which sends the content of the current web page to another application via an Intent `ACTION_SEND`. If the application BTAF is installed, it candidates to receive the Intent (carrying the sensitive data). When the user selects it as the actual receiver, BTAF is launched and the data can be transmitted to a nearby bluetooth device. The sequence of operations leading to the violation of the policy are showed in Fig. 11.

16 http://dexter.dexlabs.org/.

Table 2
Policy verification result of 100 suspicious configurations.

N \ *B*	Arduino RC car	App sender	Bluetooth bear	Bluetooth check	Bluetooth talkie	Device picker	File transfer	Settings shortcut	Share files	Transfer any file
4G Fast	↯	✓	✓	✓	✓	↯	✓	✓	✓	↯
Baidu	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗
Browser WEB	↯	✓	✓	✓	✓	✗	✗	✓	✓	✗
Dolphin	↯	✓	✓	✓	↯	✗	✗	✓	✓	✗
Firefox	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗
Light Browser	↯	✓	✓	✓	↯	✗	✗	✓	✓	✗
Maxthon	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗
Opera	↯	✓	✓	✓	↯	✗	✗	✓	✓	✗
Puffin Browser	✓	✓	✓	✓	↯	✗	✗	✓	✓	✗
SkyDrive	↯	✓	✓	✓	↯	✗	✗	✓	✓	✗

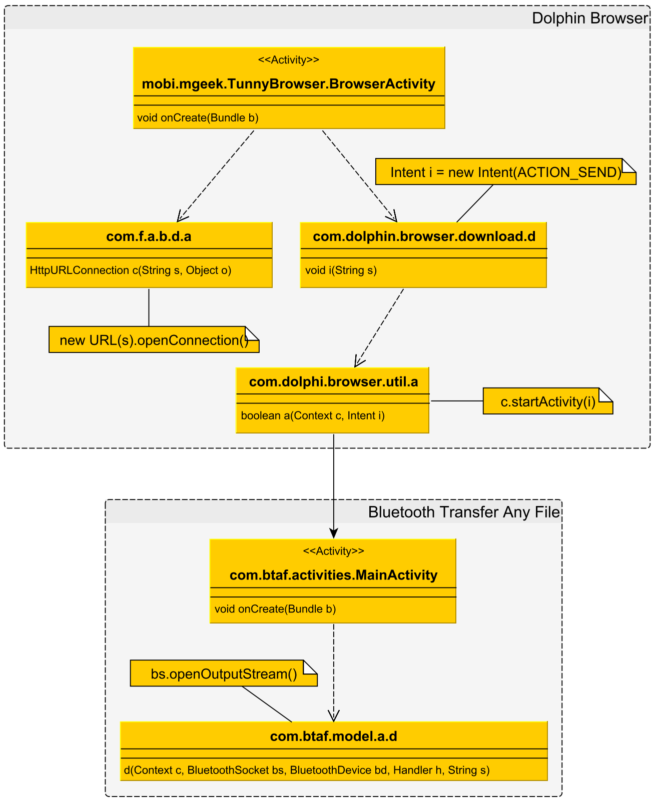
Dolphin Browser
<<Activity>>
mobi.mgeek.TunnyBrowser.BrowserActivity
void onCreate(Bundle b)
Intent i = new Intent(ACTION_SEND)
com.f.a.b.d.a
HttpURLConnection c(String s, Object o)
com.dolphin.browser.download.d
void i(String s)
new URL(s).openConnection()
com.dolphi.browser.util.a
boolean a(Context c, Intent i)
c.startActivity(i)
Bluetooth Transfer Any File
<<Activity>>
com.btaf.activities.MainActivity
void onCreate(Bundle b)
bs.openOutputStream()
com.btaf.model.a.d
d(Context c, BluetoothSocket bs, BluetoothDevice bd, Handler h, String s)

Fig. 10. The (simplified) class diagram showing the interaction between Dolphin and BTAF.

6.3. Discussion

The experiments presented above outline two facts: (i) the proposed approach can effectively discover configurations of applications violating a fine-grained security policy and (ii) the application of PMC is crucial for the scalability of the SMM architecture.

Although the approach can generate false positives, it can effectively point out dangerous app configurations and, vice versa, certify the legal ones. Also, even without theoretical guarantees that PMC always counters the state explosion, the experimental results confirm that in most cases it significantly simplify the model checking problem.

Moreover, some of the involved technologies are still subject to active research which are likely to lead to better implementations and new features. For instance, recently, in [84] another extension of PMC has been proposed. Roughly, it exploits boolean equation systems for representing symbolic conditions over the transitions of labeled transition systems. This approach is equivalent to the symbolic PMC as it evaluates the guards of a program for simplifying a specification while applying the quotienting operator. The authors present preliminary results which confirm that their algorithm could lead to a further extension of the applicability domain of model checking (w.r.t. the standard PMC). Using their technology would require minor changes of the SMM architecture.

Model inference is another point of possible optimization. As already mentioned, better models could reduce the number of false positives and also improve the overall performances. An intriguing proposal in this direction is Sawja/BIR [85]. Sawja translates Java bytecode in a compact, but formally defined, language called BIR. BIR admits the implementation of a Type and Effect system as described in Section 4.1 which might be a valid alternative to CFG extraction.

7. Conclusion

The BYOD paradigm requires the enforcement of corporate security policies on personal devices. To this aim, some proposals have been put forward, being Secure Meta-Markets a major one. SMMs lay in the middle of the application distribution infrastructure, mask standard application markets and provide formal security guarantees on the mobile code. A SMM gets a formally defined BYOD policy from the corporation and enforces it on personal mobile devices, by means of static verification and application instrumentation techniques.

The static verification ensures that all the applications installed on a personal device comply with the BYOD policy, even considering all their possible interactions. Such verification is carried out via model checking which suffers from the state explosion problem. Hence, actual implementations might not scale over large numbers of applications. Therefore, in this paper a solution based on Partial Model Checking (PMC) techniques has been proposed for granting the applicability of the approach to real-world scenarios.

The feasibility of the approach has been proved by validating groups of applications, chosen among the most popular of the Google Play store, against a policy encoding security guidelines of the US Government. The experimental results outline that the proposed PMC solution enables SMMs to assess the security verification of configurations of applications in actual BYOD scenarios. The size of the configurations which have been taken into account during the experimental validation is closed to the average number of applications commonly installed on actual mobile devices.

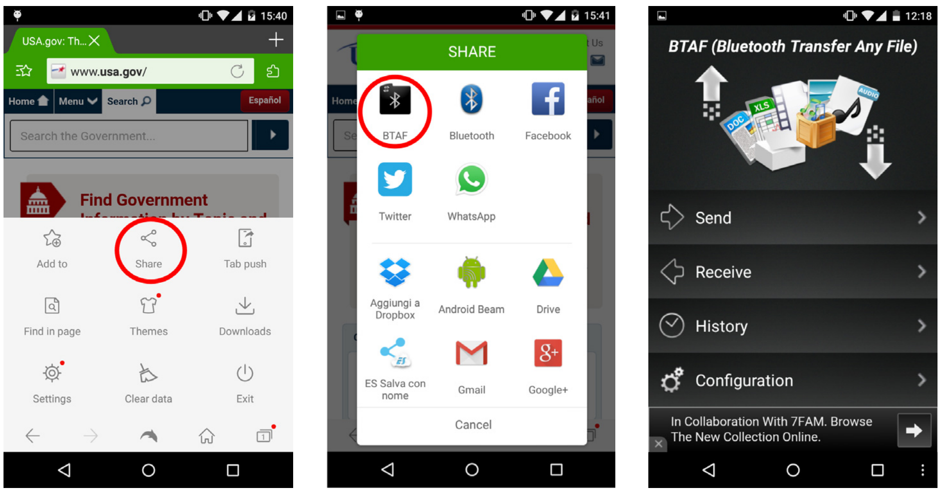

Fig. 11. Sequence of user operations causing a policy violation.

Although the presented prototype, namely BYODroid, is mature enough for the application to real BYOD environments, further improvements can be achieved. In particular, some novel techniques have been identified which could effectively enlarge the applicability domain of formal verification.

Appendix. Technical proofs

Definition 1. Resource constraints are defined as follows.

$$\phi, \phi' ::= [] \mid [\dot{x} = \dot{y}] \mid [\dot{x} \neq \dot{y}] \mid \phi; \phi'$$

where $\dot{x}, \dot{y}$ are defined as for actions. Let us use Φ to denote the set of all the formulae ϕ, ϕ'.

Definition 2. A substitution $\theta : \mathsf{Var} \to \mathsf{Nam}$ is a finite mapping from variables to names. The application of a substitution θ to actions is defined as follows.

$$\theta\tau = \tau$$
$$\theta(\alpha(\dot{x})) = \alpha(\theta\dot{x})$$
$$\theta(\bar{\alpha}(\dot{x})) = \bar{\alpha}(\theta\dot{x}).$$

Let us trivially extend the domain of θ to Nam by defining $\theta(v) = v$ if $v \in \mathsf{Val}$.

Definition 3. A substitution θ satisfies ϕ ($\theta \vdash \phi$) according to the following inductive rules.

$$\theta \vdash [] \qquad \frac{\theta\dot{x} = \dot{z} = \theta\dot{y}}{\theta \vdash [\dot{x} = \dot{y}]} \qquad \frac{\theta\dot{x} = \dot{z} \quad \theta\dot{y} = \dot{w}}{\theta \vdash [\dot{x} \neq \dot{y}]}\ \dot{z} \neq \dot{w}$$
$$\frac{\theta \vdash \phi \qquad \theta \vdash \phi'}{\theta \vdash \phi; \phi'}$$

Definition 4. A *symbolic transition system* (STS) is a triple $(S, \to, i)$ where

- S is a finite set of states ranged over by $s, s', \ldots$;
- $\to \subseteq S \times \Phi \times Ev \times S$ is a symbolic transition relation, and;
- $i \in S$ is the initial state

with $Ev = Act \times (\mathsf{Var} \cup \mathsf{Nam})$.

Definition 5. Given a STS $t = (S, \to, i)$, the set of states satisfying an assertion A is denoted with $\llbracket A \rrbracket^{\rho}_{t}$ where ρ is an environment assigning a value to each variable (t can be omitted when it is clear from the context). Let us use $\{\}$ for the empty environment, $\rho\{X \leftarrow U\}$ for the environment behaving as ρ but for X which is mapped to U and $\rho \circ \rho'$ for the composition of the two environments ρ and ρ'.

The function $\llbracket \cdot \rrbracket^{\cdot}_{\cdot}$ is defined as follows.

$$\llbracket F \rrbracket^{\rho}_{t} = \emptyset \qquad \llbracket T \rrbracket^{\rho}_{t} = S \qquad \llbracket X \rrbracket^{\rho}_{t} = \rho(X)$$
$$\llbracket A \vee A' \rrbracket^{\rho}_{t} = \llbracket A \rrbracket^{\rho}_{t} \cup \llbracket A' \rrbracket^{\rho}_{t} \qquad \llbracket A \wedge A' \rrbracket^{\rho}_{t} = \llbracket A \rrbracket^{\rho}_{t} \cap \llbracket A' \rrbracket^{\rho}_{t}$$
$$\llbracket \langle \alpha(\dot{x}) \rangle A \rrbracket^{\rho}_{t} = \{s \in S \mid \exists s', \theta. s \xrightarrow{\phi, \alpha(\dot{z})} s' \text{ and}$$
$$\theta \vdash \phi; [\dot{x} = \dot{z}] \text{ and } s' \in \llbracket \theta A \rrbracket^{\rho}_{t}\}$$
$$\llbracket \langle \tau \rangle A \rrbracket^{\rho}_{t} = \{s \in S \mid \exists s'. s \xrightarrow{\phi, \tau} s' \text{ and } s' \in \llbracket A \rrbracket^{\rho}_{t}\}$$
$$\llbracket [\alpha(\dot{x})] A \rrbracket^{\rho}_{t} = \{s \in S \mid \forall s', \theta. s \xrightarrow{\phi, \alpha(\dot{z})} s' \text{ and}$$
$$\theta \vdash \phi; [\dot{x} = \dot{z}] \text{ implies } s' \in \llbracket \theta A \rrbracket^{\rho}_{t}\}$$
$$\llbracket [\tau] A \rrbracket^{\rho}_{t} = \{s \in S \mid \forall s'. s \xrightarrow{\phi, \tau} s' \text{ implies } s' \in \llbracket A \rrbracket^{\rho}_{t}\}.$$

Slightly abusing the notation, it is possible to apply θ to an assertion A to represent the assertion obtained from A by applying θ to its actions.

The solution of a set of assertion equations E is an environment ρ assigning a set of states to each variable. Hence, let us extend $\llbracket \cdot \rrbracket$ to E in the following, inductive way.

$$\llbracket \varepsilon \rrbracket^{\rho}_{t} = \{\}$$
$$\llbracket X =_{\sigma} AE \rrbracket^{\rho}_{t} = (\llbracket E \rrbracket^{\rho\{X \leftarrow U'\}}_{t})\{X \leftarrow U'\}$$
$$\text{where } U' = \sigma U.\llbracket A \rrbracket^{\rho\{X \leftarrow U\} \circ \psi(U)}_{t}$$
$$\text{and } \psi(U) = \llbracket E \rrbracket^{\rho\{X \leftarrow U\}}_{t}.$$

Also, let us define $\llbracket E \downarrow X \rrbracket_{t} = \llbracket E \rrbracket^{\{\}}_{t}(X)$. In general, let us say that t *satisfies* $E \downarrow X$, in symbols $t \models E \downarrow X$, if $i \in \llbracket E \downarrow X \rrbracket_{t}$.

Instead, it is possible to use $\langle\!\langle \cdot \rangle\!\rangle^{\cdot}_{\cdot}$ to denote the semantic function for LTS of [11].

Definition 6. Given an STS $t = (S, \to, i)$ and an LTS $\mathbf{t} = (\mathbf{S}, \Rightarrow, \mathbf{i})$ a state $s \in S$ instantiates to $\mathbf{s} \in \mathbf{S}$ through substitution θ (in symbols $s \gg_{\theta} \mathbf{s}$) if and only if for all transitions $s \xrightarrow{\phi, a} s'$ there exists a transition $\mathbf{s} \overset{\theta a}{\Rightarrow} \mathbf{s}'$ and $s' \gg_{\theta} \mathbf{s}'$.

t instantiates to $\mathbf{t}$ through substitution θ (in symbols $t \gg_{\theta} \mathbf{t}$) if and only if $i \gg_{\theta} \mathbf{i}$.

Let us write $U \gg_{\theta} \mathbf{U}$ if and only if $\forall s \in U. \exists \mathbf{s} \in \mathbf{U}. s \gg_{\theta} \mathbf{s}$. Given an environment ρ let us write $\lfloor \rho \rfloor_{\theta}$ to denote the environment $\bar{\rho}$ such that $\forall X. \rho(X) \gg_{\theta} \bar{\rho}(X)$.

Lemma 1.

$$\forall\theta, A, \rho, \mathbf{t}.\langle\!\langle A\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta} \equiv \langle\!\langle \theta A\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$$

Proof. Let us proceed by induction on A. Most cases are straightforward since either $A = \theta A$, i.e., cases T, F and X, or they trivially reduce to the inductive hypothesis, i.e., cases $A_1 \wedge A_2$, $A_1 \vee A_2$, $\langle\tau\rangle A'$ and $[\tau]A'$. The remaining two cases, i.e., $\langle\alpha(\dot{x})\rangle A'$ and $[\alpha(\dot{x})]A'$, are symmetric. If $A = \langle\alpha(\dot{x})\rangle A'$ it must be proved that $\forall \mathbf{s} \in \mathbf{S}.\mathbf{s} \in \langle\!\langle \langle\alpha(\dot{x})\rangle A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$ iff $\mathbf{s} \in \langle\!\langle \theta(\langle\alpha(\dot{x})\rangle A')\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$.

By definition $\theta(\langle\alpha(\dot{x})\rangle A') = \langle\alpha(\theta\dot{x})\rangle(\theta A')$. Also, $\mathbf{s} \overset{\alpha(\theta\dot{x})}{\Rightarrow} \mathbf{s}'$ only if one of the following two cases occurs. Either $\alpha(\dot{x}) = \alpha(\underline{u})$ (by definition $\theta u = u$ which suffice to conclude) or $\alpha(\dot{x}) = \bar{\beta}(z)$. In the second case, $\theta z = v$ and it must be proved that $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(z)\rangle A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$ iff $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(v)\rangle\theta A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$. Clearly, by setting $z = v$, $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(v)\rangle\theta A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$ implies $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(z)\rangle A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$. Vice versa, since $s' \in \langle\!\langle A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$ by inductive hypothesis, $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(z)\rangle A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$ implies $\mathbf{s} \in \langle\!\langle \langle\bar{\beta}(v)\rangle\theta A'\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$. □

Lemma 2. *Given a STS $t = (S, \rightarrow, i)$ and a set of equations E, for every $\mathbf{t} = (\mathbf{S}, \Rightarrow, \mathbf{i})$ and θ such that $t \gg_\theta \mathbf{t}$ then $\forall X, s.s \in \llbracket E\rrbracket^\rho_t(X)$ if and only if $\exists \mathbf{s} \in \langle\!\langle E\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}(X)$ and $s \gg_\theta \mathbf{s}$.*

Proof. The proof starts by proving that (under the assumption of the lemma) for all A.

$$\llbracket A\rrbracket^\rho_t \gg_\theta \langle\!\langle A\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}. \tag{A.1}$$

By induction on A

- Cases T and F. Trivially from $S \gg_\theta \mathbf{S}$ and $\emptyset \gg_\theta \emptyset$.
- Case X. By definition $\rho(X) = U \gg_\theta \mathbf{U} = \lfloor\rho\rfloor_\theta(X)$.
- Cases $A \wedge A'$ and $A \vee A'$. The proof reduces to the inductive hypothesis applied to A and/or A'.
- Case $\langle\tau\rangle A$. By the inductive hypothesis it is possible to infer that $\llbracket A\rrbracket^\rho_t \gg_\theta \langle\!\langle A\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta}$. Since $t \gg_\theta \mathbf{t}$, if there exists $\mathbf{s} \in \mathbf{S}$ such that $\mathbf{s} \overset{\tau}{\Rightarrow} \mathbf{s}'$, there is also $s \in S$ such that $s \xrightarrow{\phi,\tau} s'$ and $\theta \models \phi$. The other way round, if exists $s \in S$ such that $s \xrightarrow{\phi,\tau} s'$ and $\theta \models \phi$ then there must be $\mathbf{s} \in \mathbf{S}$ such that $\mathbf{s} \overset{\tau}{\Rightarrow} \mathbf{s}'$. In both cases the inductive hypothesis suffices to conclude.
- Case $\langle\alpha(\dot{x})\rangle A$. The proof proceeds similarly to the previous case. Only note that, to reduce to the inductive hypothesis, Lemma 1 must be applied.
- Cases $[\tau]A$ and $[\alpha(\dot{x})]A$. Both are symmetrical to the previous two cases.

Then, it is possible to proceed by induction on E. If $E = \varepsilon$ the property trivially holds. Instead, if E is of type $Y =_\sigma AE'$, there are two cases. If $Y \neq X$ it must be proved that $s \in \llbracket E'\rrbracket^{\rho\{X \mapsfrom U\}}_t(Y)$ iff $\mathbf{s} \in \llbracket E'\rrbracket^{\lfloor\rho\rfloor_\theta\{X \mapsfrom \mathbf{U}\}}_t(Y)$. To reduce to the inductive hypothesis it must be shown that $U \gg_\theta \mathbf{U}$. Since both U and $\mathbf{U}$ are obtained through fixed point computation, induction must be applied.

- **Base case** Depending on σ it can be $U = \mathbf{U} = \emptyset$ ($\sigma = \mu$) or $U = S$ and $\mathbf{U} = \mathbf{S}$ ($\sigma = \nu$). Both are trivial since $\emptyset \gg_\theta \emptyset$ and $S \gg_\theta \mathbf{S}$ (by assumption).
- **Inductive step** It shows the proof for $\sigma = \mu$ (the case for $\sigma = \nu$ is symmetric). Let us assume $U \gg_\theta \mathbf{U}$ and show that

  $$\llbracket A\rrbracket^{\rho\{X \mapsfrom U\}\circ\psi(U)}_t \gg_\theta \langle\!\langle A\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\rfloor_\theta\{X \mapsfrom \mathbf{U}\}\circ\underline{\psi}(\mathbf{U})}$$

  where the $\underline{\psi}$ is defined as in [11]. Trivially, $U \gg_\theta \mathbf{U}$ implies that $\lfloor\rho\rfloor_\theta\{X \mapsfrom \mathbf{U}\} = \lfloor\rho\{X \mapsfrom U\}\rfloor_\theta$. Moreover, it is possible to prove that $\psi(U) \gg_\theta \underline{\psi}(\mathbf{U})$. Indeed, by inductive hypothesis $\llbracket E\rrbracket^{\rho\{X \mapsfrom U\}}_t \gg_\theta \langle\!\langle E\rangle\!\rangle^{\mathbf{t}}_{\lfloor\rho\{X \mapsfrom U\}\rfloor_\theta}$. Thus, $\lfloor\rho\rfloor_\theta\{X \mapsfrom \mathbf{U}\} \circ \underline{\psi}(\mathbf{U}) = \lfloor\rho\{X \mapsfrom U\} \circ \psi(U)\rfloor_\theta$ and it conclude by applying (A.1).

Instead, if $Y = X$ there just the need to show that $U \gg_\theta \mathbf{U}$, as done in the previous case. □

Theorem 2. *Given a STS t and a specification $E \downarrow X$, for every $\mathbf{t}$ and θ such that $t \gg_\theta \mathbf{t}$ then $i \in \llbracket E \downarrow X\rrbracket_t$ if and only if $\mathbf{i} \in \langle\!\langle E \downarrow X\rangle\!\rangle^{\mathbf{t}}$.*

Proof. A corollary of Lemma 2. □

Definition 7. Given two STS $t = (S, \rightarrow, i)$ and $t' = (S', \rightarrow', i')$, let us define the STS $t \parallel_{L,M} t' = (\mathcal{S}, \rightsquigarrow, \iota)$ as follows.

- $\mathcal{S} = S \times S'$
- $\iota = (i, i')$
- $\rightsquigarrow = T \cup T' \cup T''$

where

$$\begin{aligned} T &= \{((s, s'), \alpha, \phi, (s'', s')) \mid (s, \phi, a, s'') \in\rightarrow \wedge s' \in S'\} \\ T' &= \{((s, s'), \alpha, \phi, (s, s'')) \mid (s', \phi, a, s'') \in \rightarrow' \wedge s \in S\} \end{aligned}$$

and

$$T'' = \left\{ ((s_1, s'_1), \phi'', \tau, (s_2, s'_2)) \;\middle|\; \begin{array}{l} (s_1, \phi, \alpha(\dot{x}), s_2) \in\rightarrow \wedge \\ (s'_1, \phi', \bar{\alpha}(\dot{y}), s'_2) \in \rightarrow' \wedge \\ \alpha \in \bar{L} \cap M \wedge \\ \phi'' = \phi; \phi'; [\dot{x} = \dot{y}] \end{array} \right\}.$$

Definition 8. Given two STS $t = (S, \rightarrow, i)$ and $t' = (S', \rightarrow', i')$ and an environment $\rho : \mathit{Var} \rightarrow 2^{S\times S'}$, let us define $\rho /\!\!/ t$ as the environment such that

$$\rho /\!\!/ t(X_s) = \{s' \in S' : (s, s') \in \rho(X)\}.$$

Lemma 3.

$$(s', s) \in \llbracket A\rrbracket^\rho_{t \parallel_{L,M} t'} \quad \textit{if and only if} \quad s' \in \llbracket A /\!\!/_{L,M}\, s\rrbracket^{\rho /\!\!/ t}_{t'}.$$

Proof. By induction on A.

- Cases T and F. Trivial.
- Case X. The property is instantiated to

  $$(s', s) \in \llbracket X\rrbracket^\rho_{t \parallel_{L,M} t'} \quad \text{if and only if} \quad s' \in \llbracket X_s\rrbracket^{\rho /\!\!/ t}_{t'}$$

  which is true by definition of $\rho /\!\!/ t$.
- Cases $A \wedge A'$ and $A \vee A'$. It is possible to directly apply the inductive hypothesis
- Case $\langle a\rangle A$. There are two sub-cases for $a = \alpha(\dot{x})$ and $a = \tau$. If $a = \alpha(\dot{x})$ it must be shown that

  $$(s', s) \in \llbracket \langle\alpha(\dot{x})\rangle A\rrbracket^\rho_{t \parallel_{L,M} t'} \quad \text{iff } s' \in \llbracket \langle\alpha(\dot{x})\rangle A /\!\!/_{L,M}\, s\rrbracket^{\rho /\!\!/ t}_{t'}.$$

  Hence, the proof is split in two parts.
  (⇒) By definition $(s', s) \in \llbracket \langle\alpha(\dot{x})\rangle A\rrbracket^\rho_{t \parallel_{L,M} t'}$ if and only if there exist $(\bar{s}', \bar{s})$ and θ such that $(s', s) \xrightarrow{\phi,\alpha(\dot{y})} (\bar{s}', \bar{s})$ and $\theta \vdash \phi; [\dot{x} = \dot{y}]$ implies $(\bar{s}', \bar{s}) \in \llbracket \theta A\rrbracket^\rho_{t \parallel_{L,M} t'}$. Moreover, by definition of $t \parallel_{L,M} t'$, there are two possibilities. Either $s' \xrightarrow{\phi,\alpha(\dot{y})} \bar{s}'$ (and $\bar{s} = s$) or $s \xrightarrow{\phi,\alpha(\dot{y})} \bar{s}$ (and $\bar{s}' = s'$). By applying the quotienting operator it follows that $\llbracket \langle\alpha(\dot{x})\rangle A /\!\!/_{L,M}\, s\rrbracket^{\rho /\!\!/ t}_{t'}$ reduces to

  $$\llbracket \langle\alpha(\dot{x})\rangle (A /\!\!/_{L,M}\, s)\rrbracket^{\rho /\!\!/ t}_{t'} \cup \bigcup_{s \xrightarrow{\phi,\alpha(\dot{y})} \bar{s}} \llbracket \gamma A /\!\!/_{L,M}\, \bar{s}\rrbracket^{\rho /\!\!/ t}_{t'}$$

  where $\gamma = \mathrm{mgu}(\phi; [\dot{x} = \dot{y}])$. However, since $(\bar{s}', s) \in \llbracket \theta A\rrbracket^\rho_{t \parallel_{L,M} t'}$, by applying the inductive hypothesis it happens that $\bar{s}' \in \llbracket (\theta A /\!\!/_{L,M}\, s)\rrbracket^{\rho /\!\!/ t}_{t'}$. Also, being a γ the most general unifier for $\phi; [\dot{x} = \dot{y}]$ it is possible to use it in place of θ. This suffices to conclude that $s' \in \llbracket \langle\alpha(\dot{x})\rangle A /\!\!/_{L,M}\, s\rrbracket^{\rho /\!\!/ t}_{t'}$.

($\Leftarrow$) Since $s' \in \llbracket \langle\alpha(\dot{x})\rangle A \sslash_{L,M} s \rrbracket_{t'}^{\rho\sslash t}$ there are two possible cases (by definition of quotienting operator). Either (a) $s' \in \llbracket \langle\alpha(\dot{x})\rangle (A \sslash_{L,M} s) \rrbracket_{t'}^{\rho\sslash t}$ or (b) $s' \in \bigcup_{s \xrightarrow{\phi,\alpha(\dot{y})} \bar{s}} \llbracket \gamma A \sslash_{L,M} \bar{s} \rrbracket_{t'}^{\rho\sslash t}$. If (a) holds, then there exists $\bar{s}'$ such that $\bar{s}' \in \llbracket \theta(A \sslash_{L,M} s) \rrbracket_{t'}^{\rho\sslash t}$ (where θ is obtained as in the previous case). Hence, by inductive hypothesis, it follows that $(\bar{s}', s) \in \llbracket \theta A \rrbracket^{\rho}_{t \,\|_{L,M}\, t'}$ which suffices to conclude. Instead, if (b) holds, there must be at least one $\bar{s}$ such that $s' \in \llbracket \gamma A \sslash_{L,M} \bar{s} \rrbracket_{t'}^{\rho\sslash t}$ (where γ is the mgu of the previous case). Then, by inductive hypothesis, it follows that $(s', \bar{s}) \in \llbracket \gamma A \rrbracket^{\rho}_{t \,\|_{L,M}\, t'}$ which implies $(s', s) \in \llbracket \langle\alpha(\dot{x})\rangle A \rrbracket^{\rho}_{t \,\|_{L,M}\, t'}$, that is the thesis.

Finally, if $a = \tau$ there are three cases: s admits a τ transition, s' admits a τ transition or s' and s admit a synchronization step. The first two are analogous to the previous case (it is enough to consider τ in place of $\alpha(\dot{x})$). Thus let us only develop the third case. By assumption $s' \xrightarrow{\phi',\alpha(\dot{x})} \bar{s}'$ and $s \xrightarrow{\phi,\alpha(\dot{y})} \bar{s}$. By definition of $\|$, $(s', s) \xrightarrow{\phi'',\tau} (\bar{s}', \bar{s})$ where $\phi'' = \phi'; \phi; [\dot{x} = \dot{y}]$. Again, the proof is split in two parts.

($\Rightarrow$) Here it follows that $(\bar{s}', \bar{s}) \in \llbracket \theta A \rrbracket^{\rho}_{t \,\|_{L,M}\, t'}$ (where $\theta \vdash \phi''$) and it must be shown that $s' \in \llbracket \langle\tau\rangle A \sslash_{L,M} s \rrbracket_{t'}^{\rho\sslash t}$. Applying the definition of $\sslash$, it suffices to prove that

$$s' \in \left\llbracket \bigvee_{s \xrightarrow{\phi,\alpha(\dot{x}_1)} \bar{s}}^{\bar{\alpha}\in\bar{L}\cap M} \langle\bar{\alpha}(\dot{x}_1)\rangle(\gamma A \sslash_{L,M} \bar{s}) \right\rrbracket_{t'}^{\rho\sslash t}$$

which can be reduced to $s' \in \llbracket \langle\alpha(\dot{x})\rangle\, \gamma A \sslash_{L,M} \bar{s} \rrbracket_{t'}^{\rho\sslash t}$ by considering the transitions of s' and s in the assumptions of this case (i.e., by setting $x = x_1$). This statement holds if $\bar{s}' \in \llbracket \theta'(\gamma A) \sslash_{L,M} \bar{s} \rrbracket_{t'}^{\rho\sslash t}$. Clearly, it is possible to combine θ' and γ in a single substitution $\bar{\theta}$ such that $\bar{\theta}(\dot{x}) = \theta'(\gamma(\dot{x}))$. Hence, it can be concluded by applying the inductive hypothesis.

($\Leftarrow$) In this case the assumption is that $s' \in \llbracket \langle\tau\rangle A \sslash_{L,M} s \rrbracket_{t'}^{\rho\sslash t}$. By definition of $\sslash$ this implies (at least) one of the following three possibilities.

1. $s' \in \llbracket \langle\tau\rangle (A \sslash_{L,M} s) \rrbracket_{t'}^{\rho\sslash t}$
2. $s' \in \llbracket \bigvee_{s \xrightarrow{\phi,\tau} s''} (\gamma A \sslash_{L,M} s'') \rrbracket_{t'}^{\rho\sslash t}$
3. $s' \in \llbracket \bigvee^{\bar{\alpha}\in\bar{L}\cap M}_{s \xrightarrow{\alpha(\dot{x}),\phi'} s''} \langle\alpha(\dot{y})\rangle(\gamma' A \sslash_{L,M} s'') \rrbracket_{t'}^{\rho\sslash t}$

where γ and γ' are defined as usual. The first two cases are trivially satisfied by applying the definition of $\|$. In the third case the two transitions for s' and s in the hypothesis must be considered, i.e., it is possible to set $\dot{x} = \dot{y}$ and $s'' = \bar{s}$. Again, the proof concludes by applying the inductive hypothesis.

- Case $[a]A$. Analogous to the previous case. □

Theorem 3.

$t' \,\|_{L,M}\, t \models E \downarrow X$ *if and only if* $t' \models (E \downarrow X) \sslash_{L,M} t$.

Proof. A corollary of Lemma 3. □

References

[1] R. Ballagas, M. Rohs, J.G. Sheridan, J. Borchers, BYOD: Bring Your Own Device, in: Proceedings of the Workshop on Ubiquitous Display Environments, 2004.

[2] I. Cook, BYOD—Research findings released, November 2012, http://cxounplugged.com/2012/11/ovum_byod_research-findings-released/.

[3] Apple Inc., Apple App Review Guidelines, October 2013, https://developer.apple.com/appstore/guidelines.html.

[4] Google Inc., Google Play Developer Program Policies, October 2013, https://play.google.com/about/developer-content-policy.html.

[5] T. Wang, K. Lu, L. Lu, S. Chung, W. Lee, Jekyll on iOS: When benign apps become evil, in: Proceedings of the 22nd USENIX Security Symposium, 2013, pp. 559–572.

[6] A. Armando, G. Costa, A. Merlo, Bring your own device, securely, in: S.Y. Shin, J.C. Maldonado (Eds.), SAC, ACM, 2013, pp. 1852–1858.

[7] A. Armando, G. Costa, A. Merlo, L. Verderame, Enabling byod through secure meta-market, in: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, WiSec'14, ACM, New York, NY, USA, 2014, pp. 219–230. URL http://doi.acm.org/10.1145/2627393.2627410.

[8] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, A. Weber, Cassandra: Towards a certifying app store for android, in: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices, SPSM'14, ACM, New York, NY, USA, 2014, pp. 93–104.

[9] A.P. Felt, H.J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: Attacks and defenses, in: Proceedings of the 20th USENIX Conference on Security, SEC'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 22–22. URL http://dl.acm.org/citation.cfm?id=2028067.2028089.

[10] A. Valmari, The state explosion problem, in: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets, Springer-Verlag, London, UK, 1998, pp. 429–528. URL http://dl.acm.org/citation.cfm?id=647444.727054.

[11] H.R. Andersen, Partial model checking (extended abstract), in: In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1995, pp. 398–407.

[12] A.P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS'11, 2011, pp. 627–638. URL http://doi.acm.org/10.1145/2046707.2046779.

[13] Y.J. Park, D. Chung, M.H. Dwijaksara, J. Kim, K. Kim, An Enhanced Security Policy Framework for Android, Symposium on Cryptography and Information Security, SCIS 2011.

[14] M. Nauman, S. Khan, X. Zhang, Apex: extending android permission model and enforcement with user-defined runtime constraints, in: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS'10, ACM, New York, NY, USA, 2010, pp. 328–332. URL http://doi.acm.org/10.1145/1755688.1755732.

[15] A. Armando, R. Carbone, G. Costa, A. Merlo, Android permissions unleashed, in: Computer Security Foundations Symposium, CSF, 2015 IEEE 28th, 2015, pp. 320–333.

[16] A. Castiglione, R. Prisco, A. Santis, E-Commerce and Web Technologies: 10th International Conference, EC-Web 2009, Linz, Austria, September 1–4, 2009. Proceedings, chap. Do You Trust Your Phone? Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 50–61. URL http://dx.doi.org/10.1007/978-3-642-03964-5_6.

[17] A. Merlo, M. Migliardi, N. Gobbo, F. Palmieri, A. Castiglione, A denial of service attack to umts networks using sim-less devices, IEEE Trans. Dependable Secure Comput. 11 (3) (2014) 280–291.

[18] N. Gobbo, A. Merlo, M. Migliardi, A denial of service attack to gsm networks via attach procedure, in: Lecture Notes in Computer Science, vol. 8128, 2013, pp. 361–376.

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, SIGPLAN Not. 49 (6) (2014) 259–269.

[20] W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer, Android taint flow analysis for app sets, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, SOAP'14, ACM, New York, NY, USA, 2014, pp. 1–6.

[21] L. Li, A. Bartel, T.F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. Mcdaniel, IccTA: Detecting inter-component privacy leaks in android apps, in: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, 2015.

[22] P. Kodeswaran, V. Nandakumar, S. Kapoor, P. Kamaraju, A. Joshi, S. Mukherjea, Securing enterprise data on smartphones using run time information flow control, in: Mobile Data Management (MDM), 2012 IEEE 13th International Conference on, 2012.

[23] A.P. Fuchs, A. Chaudhuri, J.S. Foster, Scandroid: Automated Security Certification of Android Applications, Tech. Rep., UM Computer Science Department, 2009, URL http://hdl.handle.net/1903/11847.

[24] C. Gibler, J. Crussell, J. Erickson, H. Chen, Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale, in: Trust and Trustworthy Computing, Springer, Berlin, Heidelberg, 2012, pp. 291–307.

[25] W. Enck, P. Gilbert, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N. Sheth, TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–6. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[26] D. Schreckling, J. Köstler, M. Schaff, Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android, Information Security Technical Report.

[27] S. Smalley, R. Craig, Security enhanced (se) android: Bringing flexible mac to android.

[28] A. Merlo, G. Costa, L. Verderame, A. Armando, Android vs. seandroid: An empirical assessment, Pervasive Mobile Comput. (2016) URL http://www.sciencedirect.com/science/article/pii/S1574119216000080.

[29] L. Bauer, J. Ligatti, D. Walker, Composing security policies with polymer, SIGPLAN Not. 40 (6) (2005) 305–314. URL http://doi.acm.org/10.1145/1064978.1065047.

[30] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, D. Vanoverberghe, Security-by-contract on the.net platform, Secur. Tech. Rep. 13 (1) (2008) 25–32.

[31] G. Costa, F. Martinelli, P. Mori, C. Schaefer, T. Walter, Runtime monitoring for next generation Java ME platform, Comput. Secur. 29 (1) (2010) 74–87.

[32] A. Pnueli, The temporal logic of programs, in: Foundations of Computer Science, 1977, 18th Annual Symposium on, 1977, pp. 46–57.

[33] F.B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50.

[34] F. Nielson, H.R. Nielson, Type and effect systems, in: E.-R. Olderog, B. Steffen (Eds.), Correct System Design, in: Lecture Notes in Computer Science, vol. 1710, Springer, 1999, pp. 114–136. URL http://dblp.uni-trier.de/db/conf/birthday/langmaack1999.html#NielsonN99.

[35] A. Armando, G. Costa, A. Merlo, Formal modeling and reasoning about the android security framework, in: C. Palamidessi, M.D. Ryan (Eds.), Trustworthy Global Computing, in: Lecture Notes in Computer Science, vol. 8191, Springer, Berlin, Heidelberg, 2013, pp. 64–81.

[36] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: A Minimal Core Calculus for Java and GJ, in: ACM Transactions on Programming Languages and Systems, 1999, pp. 132–146.

[37] C. Skalka, S. Smith, History effects and verification, in: Second ASIAN Symposium on Programming Languages and Systems, APLAS, Springer, 2004, pp. 107–128.

[38] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, R. Zunino, Securing java with local policies, J. Object Technol. 8 (4) (2009) 5–32.

[39] I. Attali, D. Caromel, M. Russo, A Formal Executable Semantics For Java, Princeton University, 1990.

[40] G. Bierman, G.M. Bierman, G.M. Bierman, M. Parkinson, M.J. Parkinson, M.J. Parkinson, A.M. Pitts, A.M. Pitts, A.M. Pitts, MJ: An Imperative Core Calculus For Java and Java With Effects, Tech. Rep., University of Cambridge, 2003.

[41] D. Bogdanas, G. Roşu, K-java: A complete semantics of java, in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'15, ACM, New York, NY, USA, 2015, pp. 445–456. URL http://doi.acm.org/10.1145/2676726.2676982.

[42] P. de Carvalho Gomes, A. Picoco, D. Gurov, Sound control flow graph extraction from incomplete java bytecode programs, in: S. Gnesi, A. Rensink (Eds.), Fundamental Approaches to Software Engineering, in: Lecture Notes in Computer Science, vol. 8411, Springer, Berlin, Heidelberg, 2014, pp. 215–229. URL http://dx.doi.org/10.1007/978-3-642-54804-8_15.

[43] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot, in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP'12, ACM, New York, NY, USA, 2012, pp. 27–38. URL http://doi.acm.org/10.1145/2259051.2259056.

[44] A. Amighi, P. de C Gomes, D. Gurov, M. Huisman, Sound control-flow graph extraction for java programs with exceptions, in: Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEFM'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 33–47. URL http://dx.doi.org/10.1007/978-3-642-33826-7_3.

[45] F. Martinelli, I. Matteucci, Synthesis of local controller programs for enforcing global security properties, in: Availability, Reliability and Security, 2008. ARES 08. Third International Conference on, 2008, pp. 1120–1127.

[46] F. Martinelli, I. Matteucci, Partial model checking for the verification and synthesis of secure service compositions, in: S. Katsikas, I. Agudo (Eds.), Public Key Infrastructures, Services and Applications, in: Lecture Notes in Computer Science, vol. 8341, Springer, Berlin, Heidelberg, 2014, pp. 1–11. URL http://dx.doi.org/10.1007/978-3-642-53997-8_1.

[47] H. Garavel, F. Lang, R. Mateescu, W. Serwe, Cadp 2011: a toolbox for the construction and analysis of distributed processes, Int. J. Softw. Tools Technol. Transfer 15 (2) (2012) 89–107. URL http://dx.doi.org/10.1007/s10009-012-0244-z.

[48] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. Vink, W. Wesselink, T.A.C. Willemse, Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings, chap. An Overview of the mCRL2 Toolset and Its Recent Advances, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 199–213. URL http://dx.doi.org/10.1007/978-3-642-36742-7_15.

[49] J. Oberheide, C. Miller, Dissecting the android bouncer, in: SummerCon, 2012, http://jon.oberheide.org/research/.

[50] I. Aktug, K. Naliuka, ConSpec: A formal language for policy specification, Sci. Comput. Program. 74 (1–2) (2008) 2–12.

[51] R. Milner, The Polyadic π-Calculus: a Tutorial, Tech. Rep., Logic and Algebra of Specification, 1991.

[52] M. Bartoletti, P. Degano, G.L. Ferrari, R. Zunino, Model checking usage policies, in: Trustworthy Global Computing: 4th International Symposium, TGC 2008, Barcelona, Spain, November 3–4, 2008, Revised Selected Papers, 2009, pp. 19–35.

[53] D. Callahan, K.D. Cooper, K. Kennedy, L. Torczon, Interprocedural constant propagation, SIGPLAN Not. 21 (7) (1986) 152–161. URL http://doi.acm.org/10.1145/13310.13327.

[54] M.N. Wegman, F.K. Zadeck, Constant propagation with conditional branches, ACM Trans. Program. Lang. Syst. 13 (2) (1991) 181–210. URL http://doi.acm.org/10.1145/103135.103136.

[55] R. De Nicola, F. Vaandrager, Action versus state based logics for transition systems, in: I. Guessarian (Ed.), Semantics of Systems of Concurrent Processes, in: Lecture Notes in Computer Science, vol. 469, Springer, Berlin, Heidelberg, 1990, pp. 407–419. URL http://dx.doi.org/10.1007/3-540-53479-2_17.

[56] M.A. Reniers, T.A.C. Willemse, Folk theorems on the correspondence between state-based and event-based systems, CoRR abs/1011.0136.

[57] A.D. Guide, Android Application Fundamentals, October 2013, available at http://developer.android.com/guide/components/fundamentals.html.

[58] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263. URL http://doi.acm.org/10.1145/5397.5399.

[59] P. Schnoebelen, The complexity of temporal logic model checking, Advances in Modal Logic 4 (2002) 393–436.

[60] C.-H. Ong, On model-checking trees generated by higher-order recursion schemes, in: 21st Annual IEEE Symposium on Logic in Computer Science, 2006, pp. 81–90.

[61] A. Arnold, A. Vincent, I. Walukiewicz, Games for synthesis of controllers with partial observation, Theoret. Comput. Sci. 303 (1) (2003) 7–34. Logic and Complexity in Computer Science.

[62] J. Bradfield, C. Stirling, Modal mu-calculi, in: Hndbook of Modal Logic, Elsevier, 2007, pp. 721–756.

[63] A. Tarski, A lattice-theoretical fixpoint theorem and its applications.

[64] H. Comon, Disunification: a Survey, in: Computational Logic: Essays in Honor of Alan Robinson, MIT Press, 1991, pp. 322–359.

[65] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4) (2009) 21:1–21:54.

[66] E.A. Strunk, M.A. Aiello, J.C. Knight, A Survey of Tools for Model Checking and Model-Based Development, Tech. Rep. CS-2006-17, University of Virginia, 2006.

[67] P. Zhang, H. Muccini, B. Li, A classification and comparison of model checking software architecture techniques, J. Syst. Softw. 83 (5) (2010) 723–744.

[68] G.J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (1997) 279–295.

[69] G.J. Holzmann, Software model checking with SPIN, Adv. Comput. 65 (2005) 78–109. URL http://dx.doi.org/10.1016/S0065-2458(05)65002-4.

[70] B. Long, J. Dingel, T.C.N. Graham, Experience applying the SPIN model checker to an industrial telecommunications system, in: 30th ACM/IEEE International Conference on Software Engineering, ICSE '08, 2008, pp. 693–702.

[71] G.J. Holzmann, Parallelizing the SPIN model checker, in: Proceedings of the 19th International Conference on Model Checking Software, SPIN'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 155–171.

[72] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, T.A.C. Willemse, An overview of the mCRL2 toolset and its recent advances, in: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 199–213. URL http://dx.doi.org/10.1007/978-3-642-36742-7_15.

[73] M. Müller-Olm, Derivation of characteristic formulae, Electron. Notes Theor. Comput. Sci. 18 (1998) 159–170, mFCS'98 Workshop on Concurrency.

[74] I. Matteucci, Synthesis of secure systems (Ph.D. thesis), Università degli Studi di Siena, Dipartimento di Scienze Matematiche e Informatiche "R. Magari", 2008.

[75] H. Erdogmus, Verifying semantic relations in SPIN, in: Proceedings of the 1st SPIN Workshop, 1995, pp. 1–15.

[76] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An Opensource Tool for Symbolic Model Checking, Springer, 2002, pp. 359–364.

[77] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A toolbox for the construction and analysis of distributed processes, in: Tools and Algorithms for the Construction and Analysis of Systems—TACAS 2011, Saabrücken, Germany, 2011, URL http://hal.inria.fr/inria-00583776.

[78] D. Kozen, Results on the propositional μ-calculus, Theoret. Comput. Sci. 27 (3) (1983) 333–354. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982. URL http://www.sciencedirect.com/science/article/pii/0304397582901256.

[79] Anthony Desnos et. al., Androguard: reverse engineering, malware and goodware analysis of Android applications and more. URL http://code.google.com/p/androguard/, (accessed on October 2013).

[80] F. Lang, R. Mateescu, Partial model checking using networks of labelled transition systems and boolean equation systems, Log. Methods Comput. Sci. 9 (4) (2012) URL http://lmcs.episciences.org/763.

[81] Digital Services Advisory Group and Federal Chief Information Officers Council, Bring Your Own Device—A Toolkit to Support Federal Agencies Implementing Bring Your Own Device (BYOD) Programs, Tech. Rep, White House, August 2013, available at http://www.whitehouse.gov/digitalgov/bring-your-own-device.

[82] Chris Welch, Google: Android app downloads have crossed 50 billion, over 1M apps in Play. URL http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available (accessed on October 2013).

[83] A. Armando, G. Bocci, G. Chiarelli, G. Costa, G.D. Maglie, R. Mammoliti, A. Merlo, SAM: the Static Analysis Module of the MAVERIC Mobile App Security Verification Platform, in: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, 2015.

[84] F. Lang, R. Mateescu, Partial model checking using networks of labelled transition systems and boolean equation systems, in: C. Flanagan, B. König (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 7214, Springer, Berlin, Heidelberg, 2012, pp. 141–156. URL http://dx.doi.org/10.1007/978-3-642-28756-5_11.

[85] L. Hubert, N. Barré, F. Besson, D. Demange, T.P. Jensen, V. Monfort, D. Pichardie, T. Turpin, Sawja: Static analysis workshop for java, CoRR abs/1007.3353. URL http://arxiv.org/abs/1007.3353.

Gabriele Costa is assistant professor at the Informatics, Bioengineering, Robotics and System Engineering Department (DIBRIS) at the University of Genova and is a member of the Computer Security Laboratory (CSEC Lab). He received his Ph.D. in Computer Science from the University of Pisa in 2011. From 2008 to 2011 he worked as researcher for the IIT institute of the National Research Council (CNR) in Pisa. His main research activities and interests include language-based security, formal methods for security analysis, information flow analysis, distributed systems security, security verification and enforcement. He actively participated in the European projects Aniketos, CONNECT, NESSoS and SPaCIoS.

Alessio Merlo received his Ph.D. in Computer Science from University of Genova (Italy) where he worked on performance and access control issues related to Grid Computing. He is currently assistant professor at the Informatics, Bioengineering, Robotics and System Engineering Department (DIBRIS) of the University of Genova and a member of the Computer Security Laboratory (CSEC Lab). His research interests are focused on performance and security issues related to Web and distributed systems (Grid, Cloud). He is currently working on security issues related to Android platform.

Luca Verderame is a Ph.D. student, currently working at the Computer Security Laboratory (CSEC Lab), DIBRIS, University of Genova as research fellow.

He obtained his master degree in Computer Engineering, University of Genova, on march 2012.

His research interests mainly cover information security applied, in particular, to mobile devices. Recently his work led to discover a previously unknown vulnerability in Android OS.

Alessandro Armando is associate professor at the University of Genova, where he received his Laurea degree in Electronic Engineering in 1988 and his Ph.D. in Electronic and Computer Engineering in 1994. His appointments include a postdoctoral research position at the University of Edinburgh (1994–1995) and one as visiting researcher at INRIA-Lorraine in Nancy (1998–1999). He is co-founder and leader of the Artificial Intelligence Laboratory (AI-Lab) and of the Computer Security Laboratory (CSEC Lab) at DIBRIS. He is also head of the Security and Trust Research Unit at the Center for Information Technologies of Bruno Kessler Foundation in Trento. He has contributed to the discovery of a serious vulnerability on the SAML-based Single Sign-On for Google Apps and to the discovery and fixing of a vulnerability that leads to a Denial of Service attack on all Android devices.

His current focus is on developing cutting-edge automated reasoning techniques and on using them to build a new generation of push-button software verification and debugging tools supporting the development of complex, large-scale, distributed IT applications.