Enabling BYOD through Secure Meta-Market*

Alessandro Armando
Security and Trust
Fondazione Bruno Kessler
Via Sommarive, 18
Trento, Italy
armando@fbk.eu

Gabriele Costa
DIBRIS - University of Genoa
Via all'Opera Pia, 13
16145, Genova (Italy)
gabriele.costa@unige.it

Alessio Merlo
E-Campus University
Via Isimbardi, 10
22060, Novedrate (Italy)
alessio.merlo@uniecampus.it

Luca Verderame
DIBRIS - University of Genoa
Via all'Opera Pia, 13
16145, Genova (Italy)
luca.verderame@unige.it

ABSTRACT

Mobile security is a hot research topic. Yet most of available techniques focus on securing individual applications and therefore cannot possibly tackle security weaknesses stemming from the combined use of one or more applications (e.g. confused deputy attacks). Preventing these types of attacks is crucial in many important application scenarios. For instance, their prevention is a prerequisite for the widespread adoption of the BYOD paradigm in the corporate setting.

To this aim, in this paper we propose a secure meta-market which supports the specification and enforcement of security policies spanning multiple applications. Moreover, the meta-market keeps track of the security state of devices and—through a functional combination of static analysis and code instrumentation techniques—supervises the installation of new applications thereby ensuring the enforcement of the security policies. Also, we developed a prototype implementation of the secure meta-market and we used it for validating a wide range of popular Android applications against a security policy drawn from the US Government BYOD Security Guidelines. Experimental results obtained by running the prototype confirm the effectiveness of the approach.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*; D.2.4 [**Operating Systems**]: Software/Program Verification

*This work has been partially funded by the Italian PRIN project *Security Horizons* (no. 2010XSEMLC).

WiSec'14, July 23–25, 2014, Oxford, UK.

http://dx.doi.org/10.1145/2627393.2627410.

Keywords

Application Meta-Market; Android Security; Formal Verification; Code Instrumentation;

1. INTRODUCTION

The world-wide spread of smartphones and tablets as well as the emerging "Bring Your Own Device" (BYOD) paradigm are pushing mobile devices towards a professional use. Many device manufacturers are thus considering the development of architectural solutions supporting the use of personal devices in corporate environments, e.g., Apple[1], Samsung[2], Blackberry[3], Huawei[4]. Yet, the level of protection offered by current mobile operating systems does not allow to cope with the complexity and the stringent security requirements arising in corporate environments. As a matter of fact, business activities carried out on mobile devices can handle valuable resources on platforms where users, which cannot be expected to understand all the security implications, are likely to install harmful applications.

The current trend for mobile code distribution is based on application markets. Markets allow users to browse and install myriads of applications. A centralized application repository offers some advantages for the security assessment as the market can implement procedures for the analysis and testing of the mobile code. However, the most popular markets offer very limited security guarantees (see for instance [31, 41, 39]).

In this paper we propose a security-enabled application marketplace, namely *meta-market*, which enforces BYOD security policies through analysis and monitoring of mobile applications. The meta-market masks the actual application markets by mediating all the application installation requests. A fruitful combination of static and dynamic techniques is used to check whether the applications comply with the organizational security policy. More in detail, the meta-market supports: *(i)* the definition of fine-grained security

[1]http://www.apple.com/ipad/business/it/byod.html
[2]http://www.samsung.com/global/business/mobile/solution/security/samsung-knox
[3]http://it.blackberry.com/business/software/bes-10.html
[4]http://enterprise.huawei.com/en/solutions/byod/byod-mobility/index.htm

policies, *(ii)* the static verification of applications against the security policy and *(iii)* the runtime monitoring of applications failing the verification step.
We present an implementation of the meta-market[5] which deals with Android-based devices and supports the security assessment of the Google Play application market against corporate BYOD policies. To assess the effectiveness of our approach, we applied our meta-market to check the compliance of 860 Android applications taken from the Google Play store. Applications are validated against a real world BYOD security policy (extracted from the US Government BYOD Security Guidelines [17]) expressed in ConSpec [2], a well-known security specification formalism. The experimental results confirm the effectiveness of the proposed approach.

Structure of the paper. In Section 2 we discuss the state-of-the-art of the security assessment of mobile applications and some related work. Section 3 introduces the formal application and security framework adopted in this paper. In Section 4 we present the techniques and logical workflow of the proposed meta-market solution. Then, in Section 5 we describe our prototype implementation and in Section 6 we report and discuss the experimental results. We conclude in Section 7 with some final remarks and future research directions.

2. RELATED WORK

Recent researches on the security of mobile devices outline that malware are appearing more and more frequently in official application markets. In [41] the authors systematically evaluate the malware affecting Android OS which have been discovered in the last years. Also, they check the presence of malware in application markets and they consider whether most used anti-virus applications succeed in detecting them. They find that all the application markets they inspected contain several malware instances and, even worse, anti-viruses very often fail in discovering them.
The weakness of application markets security checks is also reported by Wang et al. [39]. Briefly, they developed a method for injecting malicious control flows in the application code. Applying their technique, they could modify and submit an iOS application to the Apple Store. After passing the review process of the market, the application carrying illegal instructions was successfully published. These works indicate that security mechanisms adopted by application markets are far from being sufficient to guarantee that the applications they deploy are safe for the user's device.
Furthermore, the basic security mechanisms offered by mobile OSes seem to be inadequate for providing actual security guarantees, e.g., see [21]. Moreover, OSes implementations are not immune to vulnerabilities. Indeed, some of them have been already identified and fixes have been proposed, e.g., see [8]. Clearly, unsafe applications acquired from markets can exploit these weaknesses to perform disruptive attacks.
Several proposals advocate extensions to the native Android security mechanisms (e.g., permissions system and application sandboxing). For instance, in [29] the authors present an enhancement of the basic Android permission system, while in [42] the authors suggest new privacy-related security policies for protecting users' personal data. Similarly, [11] presents a mixed static and dynamic approach for both detecting suspicious patterns in an installing application and, possibly, running it in a sandboxed environment.
A few approaches where security activities are carried out by the application market have been proposed. Among them, Google Bouncer[6] is a malware detector running on the Google Play store. However, it has been recently shown how it can be easily circumvented [31].
All the proposals listed above, although indirectly dealing with security policies, do not support user-defined security policies. In this respect, [19] presents Kirin, a local service certifying that an application under installation complies with user-defined security rules. Similarly, Saint [32] assigns privileges to applications at install time and monitors them at run time. Although both systems use security rules which can be arbitrarily complex, they still refer to the existing, insufficient set of permissions[7]. Since these are coarse-grained and informally defined, building security mechanisms on top of them is rather controversial.
A different approach is the *Security-by-Contract* [16] (SxC). SxC is an integrated framework exploiting several techniques for granting that a mobile application respects the policy of the device running it. Roughly, application developers attach a contract to their code through *proof-carrying code* [30]. After downloading the application, the mobile platform automatically verifies the correctness of the contract, i.e., whether it effectively models the application. If the check succeeds, the application is validated by comparing its contract against the platform policy, e.g., via *model checking* [12]. If one of these two steps fails, i.e., whenever the contract is incorrect or the policy is not fulfilled, the application is rejected, otherwise it is installed. Software installed in this way is proved to never violate the policy.
Although the Security-by-Contract prevents the execution of dangerous mobile code, it hardly copes with the current market-based distribution paradigm. In fact, the SxC tasks executed on the mobile device, i.e., contract validation and model checking, are non trivial. In particular, model checking requires heavy computation which may take long time and rapidly exhaust the device resources. Moreover, proof-carrying code significantly increase the size of the mobile code (as it injects proof annotations). Larger software packages would require extra storage space, which is usually non-free, on actual application markets.
Our proposal aims at effectively enforcing fine-grained BYOD security policies. Unlike the approaches listed above, our solution is designed to apply to the current market-based application distribution paradigm without executing cumbersome tasks on resource-constrained mobile devices.

3. SECURITY FRAMEWORK

Our approach relies on formal descriptions of the application behavior and the policy specification. In particular, two aspects are crucial under our assumptions, i.e., modeling the interactions among applications and the specification of fine-grained BYOD policies. A convenient abstraction of the behavior of a program is provided by its *execution trace* [1], representing the sequence of security-relevant operations it performs. In this respect, security policies

[5]Further information can be found at `http://www.ai-lab.it/byodroid/`

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

[7]`http://source.android.com/tech/security/`

are defined as temporal properties over an execution trace. Below we present our history-based application and policy modeling frameworks.

3.1 Mobile Application Framework

Mobile code is typically distributed in the form of application packages. For instance, a package may contain application code and resources (e.g., audio files and pictures).
A common way to represent the behavior of a piece of code is by building its *control flow graph* (CFG). Briefly, a CFG is a data structure representing all the possible execution flows of a program. It basically consists of states and transitions. States denote linear, i.e., jumps-free, fragments of code. In contrast, transitions represent (un)conditional jump instructions that correspond to computational branches. CFGs can be extracted from assembly code [13] as well as intermediate languages like the Java bytecode [3].
In [10] a proof system allowing to prove that a CFG complies with a security specification is discussed. A similar approach is not viable under our assumptions. Here, we are interested in considering BYOD policies regulating the behavior of the mobile device as a whole. In principle, illegal executions may involve more than one application. This is typical of well-known privilege escalation attacks, e.g., based on the application collusion and confused deputy problems [23].
Hence, we follow a different approach. In particular, we translate the CFG into a corresponding *history expression*, i.e., a process algebraic representation of a set of execution traces. The procedure resembles the one in [9] with some crucial differences. First of all, we adopt an extended version of history expressions which has been specifically designed for Android applications (see [6] for a detailed presentation). These history expressions can represent the *inter-process communication* (IPC) which is a common mechanism of interaction among applications running on mobile devices. In this way we capture the possible flows of information among applications.

3.2 Policy Specification

Several policy specification languages have been presented for defining temporal properties over the execution traces. Among them, temporal logics, e.g., LTL [33], are a major proposal. Briefly, they extend first-order logic with temporal modalities, e.g., saying that a certain property will hold on a certain trace segment.
Automata-based formalisms have similar features and are also widely adopted. For instance, each LTL formula can be efficiently translated into a corresponding *Büchi automaton* [22] which accepts a trace if and only if it satisfies the original formula. *Security automata* [35] behave similarly. Intuitively, a security automaton defines an abstract controller which reads the elements of an execution trace. As far as the next symbol is accepted by the automaton, it respects the policy. Security automata have been also used for defining the formal semantics of some policy specification languages like PSLang [20] and ConSpec [2]. In particular, ConSpec has been specifically designed for high level languages, e.g., Java, which makes it suitable for the Android environment.
For the sake of presentation, we do not provide a full description of the ConSpec syntax and semantics. Here, we simply provide the basic intuition about the language and its features. A ConSpec policy consists of a *security state* and a list of *security rules*. The state contains a set of variables representing the configuration of the policy. Variables have a scope defining how they behave on multiple targets, e.g., *session* means that the variable is reinitialized at every execution of an application (see [2] for further details). Moreover, rules define how the state changes when a certain security-relevant operation is evaluated. In order to give a basic intuition, we propose the following ConSpec policy.

```
1 SECURITY STATE
2   SESSION Str[11] agency_host = "agency.gov/";
3   SESSION Obj agency_url = null;
4   SESSION Bool connected = false;
5
6 /*(R1) No download of business data on device
      */
7 AFTER Obj url = java.net.URL.<init>(Str[64]
      spec)
8   PERFORM
9   (spec.contains(agency_host)) -> { agency_url
        := url; }
10  ELSE -> { skip; }
11
12 AFTER Obj url = java.net.URL.<init>(Str[0]
      protocol, Str[64] host, Nat[0] port,
      Str[0] file)
13  PERFORM
14  (host.contains(agency_host)) -> { agency_url
        := url; }
15  ELSE -> { skip; }
16
17 BEFORE java.net.URL.openConnection()
18  PERFORM
19  (this.equals(agency_url)) -> { connected :=
      true; }
20  ELSE -> { skip; }
21
22 BEFORE java.io.FileOutputStream.write(int i)
23  PERFORM
24  (!connected) -> { skip; }
25
26 /* (R2) "When in Doubt, Delete it Out"
      */
27 AFTER Obj file = java.io.File.createTempFile()
28  PERFORM
29  (true) -> { file.deleteOnExit(); }
30
31 /* (R3) No transfer of sensitive data
32       to non-agency devices
           */
33 BEFORE android.bluetooth.BluetoothSocket.
      getOutputStream()
34  PERFORM
35  (!connected) -> { skip; }
```

The policy encodes three clauses[8] informally stated in the US Government BYOD security guidelines [17].
Intuitively, the first group of rules (R1) says that users cannot store sensitive data on their device. We encoded this behavior by means of four rules. The first two (lines 7 and 12) say that after an application creates a `URL` object, the policy checks whether it points at some sensitive data sources (here we identify it with the `"agency.gov/"` path). If this is the case (lines 9 and 14), the policy stores the `URL` reference in the variable `agency_url`, otherwise (lines 10 and 15) the operation is simply allowed (`skip` command). The third rule (line 17) is triggered before a `URL` is used to create a data connection. If the `URL` is the same stored in `agency_url` (line 19), the policy state changes by setting the `connected`

[8]For brevity, here we omit few irrelevant elements.

flag to true, which means that the application has accessed to sensitive data. Finally, according to the fourth rule (line 22), the target can write on local files only if `connected` is false (following a *default-deny* approach, the policy blocks an operation whenever none of its rules apply to it). In addition, rule (R2, line 27) states that temporary data, which could include sensitive information, must be deleted at the end of each application session. To do that, right after the creation of every temporary file, the policy requires the invocation of `deleteOnExit()` (line 29). Eventually, rule (R3, line 33) exploits the same flag `connected` used in (R1) to decide whether to disable the Bluetooth output streaming for an application which might have accessed a sensitive `URL`. Again, when evaluating the Bluetooth operation, if the rule does not apply, i.e., if `connected` is true, the action is blocked.

4. META-MARKET PARADIGM

In this section we present the architectural aspects of the *meta-market* software distribution model. In particular, we explain how it provides formal security guarantees. Then we give a detailed description of the meta-market components and their behavior.

4.1 Formal Security Assessment Workflow

Intuitively, our proposal consists in moving most of the security assessment steps and logic on a meta-market. The meta-market holds the security policies of one or more organizations, e.g. private companies or public agencies, and its customers are the organizations' members, e.g. the employees of a company. For simplicity, in the rest of the paper we assume the meta-market to handle a single policy of a single organization. Figure 1 shows the code deployment workflow in the meta-market paradigm.

Code producers compile their applications and publish them through a standard application market. An application market simply implements a database of applications that registered devices can browse. Then, the meta-market interacts with an application market by passing the customers requests, e.g. to obtain the list of applications. In this way, the meta-market is totally transparent to the application market, which handles standard user sessions. Also, it only requires minor extensions of the user's device, which only needs to mount a dedicated installer (as most of the application markets do).

Our model works as follows. Code producers compile (P.0) and generate mobile applications[9] (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 code consumer requires an application from the meta-market, the corresponding code package is retrieved. Then, a model extraction procedure is applied to the code (B.0) to generate an application model (B.1). Since the model is extracted from the mobile code, no further validation is required (as needed, for instance, by the Security-by-Contract approach [16]). 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. More details about this aspect are discussed in Section 4.2. If the verification succeeds (B.5 $\rightarrow$ YES) the policy database is updated and the application is delivered to the user's device with no further action (B.6). Otherwise (B.5 $\rightarrow$ NO), the meta-market attaches monitoring information to the application (B.7). Mainly, monitoring information consists of a digital signature which will be used by the code consumer to obtain a correct instrumentation of the application as detailed in Section 4.2. 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.

4.2 Meta-Market Components and Activities

The workflow of the secure meta-market is obtained by a fruitful combination of techniques we describe in the following.

Model validity.

Model-based verification systems rely on the validity of models. Indeed, fake or incorrect models can compromise the whole verification process. To be valid, a model must denote (at least) every security-relevant behavior of the application it refers to. For instance, since in [16] the application model is created by an untrusted producer, it must be validated at consumer-side.

Model validation is a non-trivial task. To reduce the consumer's overhead, a possible solution is *model-carrying code* [37] (MCC). MCC is a variant of *proof-carrying code* [30] in which a piece of code, i.e. the mobile application, is instrumented with the steps of a proof of compliance against its own model. Although the proof instrumentation allows for a faster validation, it significantly increases the size of the software package [30]. Such a growth depends on the proof size and poorly fits with large scale distribution paradigms. Moreover, as model extraction is performed without knowing the consumers' policies, applying this method requires to generate inclusive, large models. Clearly, larger models exacerbate the problem of instrumenting and storing the model-carrying packages.

In our approach, the verification process takes place on the meta-market. Thus, application models can be generated locally with some clear advantages. Mainly, to obtain correct models the system just needs to rely on a sound model extraction procedure and validity proofs must be neither generated nor verified[10]. Moreover, being aware of the policy, the meta-market can extract models that only refer to the policy-relevant operations. In this way, models do not contain irrelevant information and their size is substantially reduced.

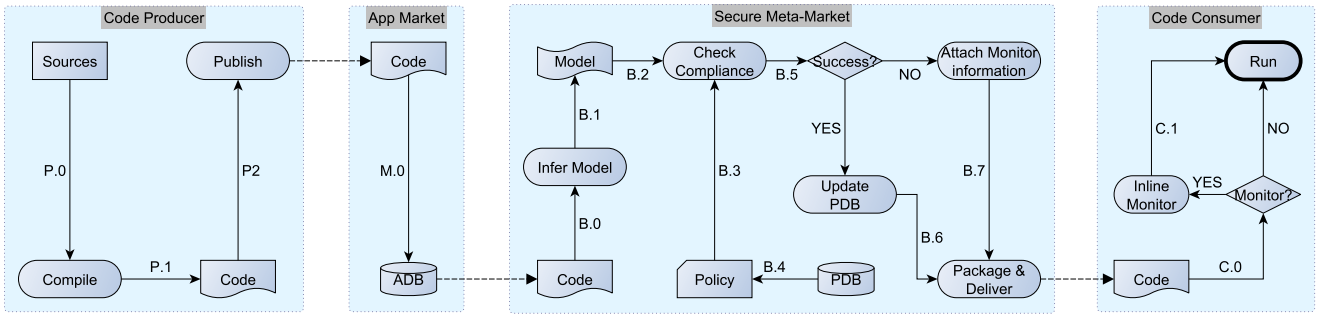
Policy compliance verification.

Policy compliance is the central activity of the secure application deployment process. Model checking [12] can be suitably applied to verify whether a given model satisfies a formal specification, i.e., a security policy. Models and specifications are often given as (labeled) transition systems (see,

[9]For the sake of brevity, we do not distinguish here between applications and their code.

[10]The model extraction program can be verified and tested before adopting it.

Figure 1: The meta-market software distribution workflow.



e.g. [26]) and temporal logic formulae, e.g., linear-time temporal logic [33], respectively. Several model checkers have been proposed in the last decades and many of them have been successfully applied to verification the correctness of real systems, e.g., see for instance [40, 27, 28].

Nevertheless, in some cases solving the model checking problem can be extremely hard and model checkers poorly scale over the size of their search space (see [36] for a survey). More specifically, it is well known that model checkers suffer from the state explosion problem [38], that is the exponential growth of the search space due to the representation of concurrent agents. Since they can run in parallel, mitigating this issue is crucial for applying model checking to the verification of mobile applications.

A first step consists in using previous analysis results in order to avoid repeated instances of the same model checking problem. This approach can be effectively supported by centralizing the use of the model checking process in the proposed meta-market architecture. In particular, the meta-market disposes of both the security policy and the application models. Hence, the system can store verification results and check whether a given instance was already considered in the past without running the model checker again. Although this approach can avoid repetitions of cumbersome computations, it does not reduce the complexity of the model checking.

A further optimization derives from including platform-specific aspects in the model generation process as it is has been proposed in [6] for the Android inter-process communication (IPC) framework.

Briefly, IPC defines application-level primitives for data and control flow composition. Each IPC function abstracts low-level operations which actually implement the communication. Hence, IPC-based communications can be modeled as simple messages over reserved channels. Since the IPC mechanisms also rules the application life-cycle, we can avoid concurrency in many models. For more details on the Android IPC and its formalization we refer the reader to [6]. A similar reasoning can be applied to other mobile platforms, e.g., see [5].

Moreover, we upper-bound the model checking phase, i.e., we force the model checker to terminate after a finite number of steps (visited states or amount of time). Setting a threshold guarantees the model checker to terminate within a fixed time but introduces a third analysis outcome, i.e., the timeout. A similar approach is also adopted in [18]. In this way, the meta-market prevents indefinitely long executions which would be unacceptable in the workflow outlined above. The meta-market treats a timeout as a verification failure and marks the application for security instrumentation and monitoring. Clearly, this may cause false positives, i.e., harmless applications being monitored. Hence, the time threshold is a parameter that need to be carefully considered to obtain a reasonable trade-off. In Section 6 we describe how we deal with timeouts in our implementation. We will discuss in detail the advantages of adopting runtime monitoring below, at the end of this section.

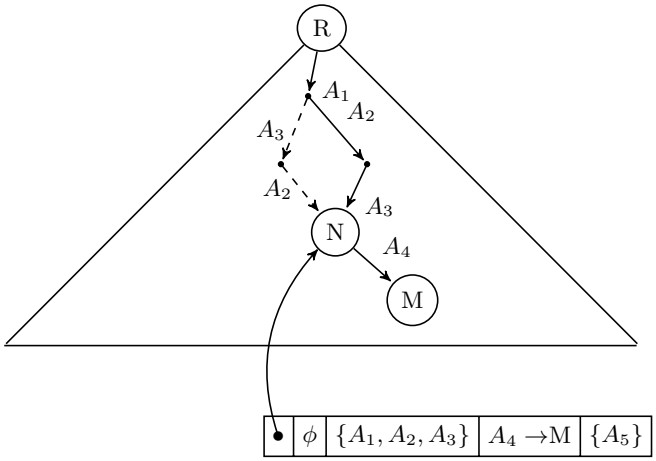
Partial policy evaluation and storage.

As we outlined in the previous paragraph, models and policies play a central role for the feasibility of model checking-based validation. In particular, the security state of each mobile device is evaluated against the models of the installed applications. This implies that the meta-market must keep track of all the set of installed applications for each device. Also, these records must be updated whenever a device installs or removes an application. Hence, we use a *direct acyclic graph* (DAG) to represent device configurations and their dependencies [7]. For instance, consider the DAG depicted in Figure 2. Node R is the root configuration denoting the original policy. The installation of applications A_1, A_2 and A_3 leads to a new configuration N (solid path). Also, installing the three applications in a different order can generate a different path from R to N (dashed path). Eventually, installing application A_4 on a device being associated to N can cause the migration to a different configuration (node M). This happens only if A_4 does not violate the policy holding in N, i.e., ϕ. Summing up, each device is associated to a node, e.g., N. The node corresponds to a record containing the current policy of the device, e.g., ϕ, the set of installed applications, e.g., $\{A_1, A_2, A_3\}$, a list of applications known to be safely installed (pointing to the corresponding node), e.g., $A_4 \rightarrow M$, and a set of applications that are known to violate the policy of the node, e.g., $\{A_5\}$. The implementation and the management of the DAG is done through a relational database as described in Section 5.3.

Beside implementation, the way device configurations are encoded must be carefully considered. Observe that a mere list of application models can be very inefficient. Indeed, consider a device which already installed applications $A_1, \ldots, A_k$, wanting to install a new application A. If the

meta-market uses the models of the installed applications, i.e., $C_1, \ldots, C_k$, to represent the configuration of the device, the model checking problem to be solved for validating A through its model C is $C_1 \mid \ldots \mid C_k \mid C \models \phi$. Where ϕ is the security policy and $\mid$ the parallel model composition operator. As we discussed above, the complexity of this problem rapidly grows when k increases.

Figure 2: Policy data structure.



In [4] Andersen presents partial model checking (PMC). Roughly, PMC consists in partially evaluating a specification against a model. In a nutshell, given one or more models $C_1 \mid \ldots \mid C_k$ and a formula ϕ, PMC returns a new formula ϕ' such that for every C

$$(C_1 \mid \ldots \mid C_k) \mid C \models \phi \text{ if and only if } C \models \phi'$$

Moreover, in [4] the author reports experimental results showing that PMC outperforms model checking when dealing with a model resulting from the parallel composition of smaller models. This is mainly due to a number of simplification techniques which can be applied for simplifying a formula. Hence, relying on PMC we just need to associate each device to its customized policy, i.e., the policy which has been partially evaluated against the device's configuration. When a new application is evaluated for installation, its model can be directly model-checked against the customized policy. If the verification is successful the model is included in the configuration via PMC, i.e., a new policy is obtained by partially evaluating the old one against the model.
For these reasons, we exploit PMC in the policy customization and device configuration management processes of the meta-market workflow. Further details about the PMC framework adopted in the meta-market are provided in Section 5.3.

Application instrumentation and monitoring.

Mobile application monitoring is a consolidated technique for enforcing policy compliance at runtime. Several implementations appeared in the literature, also running on mobile, resource-constrained devices [15, 14]. Typically, a monitoring environment consists of a *policy decision point* (PDP) and many *policy enforcement points* (PEPs). The

Listing 1: A fragment of instrumented code.

```
URL url;
PDP.checkBefore("java.net.URL.<init>",addr);
try {
  url = new URL(addr);
} catch(Exception e) {
  PDP.checkException
  ↪("java.net.URL.<init>",e,addr);
  throw e;
}
PDP.checkAfter("java.net.URL.<init>",url,addr);
```

PDP holds the security policy and the current security state, the PEPs control invocations to critical operations. When an application attempts to perform a security-relevant access, a corresponding PEP is activated. The PEP triggers the PDP which checks the security state and policy and grants or rejects the permission. Finally, the PEP enforces the PDP's response by allowing or blocking the requested access.
Monitoring environments may differ for the PEPs deployment strategy. Traditionally, two strategies exist, i.e., customizing the execution environment and instrumenting the application code. In both cases, the PDP is installed as a background service. For our purposes, PEP instrumentation is the best option. As a matter of fact, platform customization requires heavy modification of the mobile devices in order to replace/rewrite system components. For instance, for virtual machine-based systems, e.g., Java/Android, the customization consists in replacing the standard VM with a new one including the PEPs. On the contrary, under the same assumptions, the application instrumentation is much less invasive. Indeed, it only requires to slightly modify the intermediate code, i.e., the VM instructions, of the application. For a more detailed discussion about the PEP inlining for mobile applications see [14].
We implement instrumentation directly on the mobile device. As described in Section 4.1, mobile devices may receive applications to be instrumented. In this case, the instrumentation procedure injects instructions implementing the PEPs. These instructions wrap security-relevant invocations. In Listing 1 we show a code fragment obtained by instrumenting the instruction `Url url = new URL(addr)`. Methods `checkBefore`, `checkException` and `checkAfter` implement the PEP logic by requesting authorizations to the PDP. Briefly, they interrupt the code execution and alert the PDP about the ongoing operation and its parameters. The PDP evaluates the action according to the current security state. The evaluation changes the security state and produces a return value for the PEP, i.e., *allow* or *deny*. If the PEP receives an *allow* signal, the execution is resumed, otherwise it throws a security exception. We refer the interested reader to [34] for further details.
Clearly, application instrumentation invalidates the original software signature and, since most mobile OSes, e.g., Android and iOS, do not allow for the installation of unsigned contents, a new, valid one must be computed. For this reason, the application package to be instrumented contains the original code together with an *instrumentation signature*. The signature is computed by the meta-market and only applies if the application is modified in the right way. After the

instrumentation, the signature is attached and the application installed. Moreover, the signature prevents from illegal instrumentation which could modify the code in arbitrary ways.

5. META-MARKET IMPLEMENTATION

In this section we present a prototype implementation of the meta-market. The current implementation of the meta-market is meant to interact with the Android software development and distribution framework and adopts all techniques previously discussed. In the following we assume the code receiver to be an Android device and the application market to be Google Play.

5.1 Model Extraction

Models are extracted directly from the application bytecode. The (Dalvik) bytecode is the intermediate language interpreted by the standard Android virtual machine, namely the Dalvik Virtual Machine (DVM). The application code is typically written in Java and then compiled in Dalvik bytecode. Actually, application packages consist of compressed Dalvik bytecode and resources, like audio files and pictures. Occasionally, application packages may also carry native code, i.e., libraries of machine-executable procedures. Although we do not present it below, the same approach can be applied for that kind of code (e.g., see [13]).

Basically, the model extraction process consists in building a CFG representing the behavior of the application. The CFG generation is a widely adopted technique [3] and several tools support the CFG extraction for bytecode. To this aim, we adopt Androguard[11], a state-of-the-art Android package analyzer and CFG extractor. Unfortunately, Androguard extracts large and non optimized CFGs and we had to customize the tool for our specific purpose. Briefly, we implemented ad-hoc procedures for optimizing the CFG construction in terms of generated nodes and transitions. Part of this activity consists in pruning those parts of a CFG which do not refer to any security-relevant method.

5.2 Model Checking

Application validation is carried out through model checking. Although several model checkers exist, we opted for Spin [24] since it offers some practical advantages. Spin has been applied to the verification of a plethora of practical case studies in several different contexts (see [27] for a survey). In terms of reliability, Spin has been included in industrial verification activities as reported, for instance, in [28]. Moreover, Spin includes features for optimizing the use of CPU and memory. For instance, since version 5 Spin supports multicore computation [25]. Pragmatically, this allows Spin to perform better when running on powerful hardware, as we assume for a server hosting the meta-market.

The specification language for Spin is Promela. Hence, we implemented a translator to convert the CFGs extracted from application packages into corresponding Promela specifications. The definition of Promela specifications from transition systems or state machines is a quite consolidated technique, see [27]. However, in our context two aspects require more attention, i.e., IPC facilities and policy encoding. Roughly, each component of an application results in a subgraph of the overall CFG. Since components pass each other the control state through internal IPC messages, their subgraphs should not be considered as concurrent agents. In fact, we model special IPC channels in the Promela specification for simulating these interactions. Briefly, this approach prevents many states, representing impossible configurations, from being considered by the model checker. Moreover, since IPC can also occur among different applications, we apply a similar reasoning to interface components, i.e., those that can be invoked by other applications. Such components are known through the application interface definition contained in its manifest file.

Usually, Promela specifications contain both the system and the property for the model checker. For instance, a LTL formula defined over the agents' states can be (automatically) translated into an invariant, i.e., a *never* claim, and applied to the Promela agents. However, the properties we want to verify here are related to the messages sent from applications to the platform, i.e., the system calls. Hence, we are interested in evaluating sequences of messages, rather than the states of the Promela agents. For this reason, we directly encode the policy as an extra Promela agent. Clearly, this step relies on the formal definition of the transitional semantics of the ConSpec policy language.

The Promela agent obtained from the policy represents an abstract controller reading the abstract traces generated by the model. If the application model carries some policy-violating behavior, the policy controller reaches a faulty state causing Spin to report an error. In that case, the output contains the trace of transitions which can be mapped back to the application code in order to locate the illegal flow.

5.3 Policy Management and Partial Evaluation

In Section 4.2 we highlighted the advantages of applying partial model checking to the partial evaluation of security policies. Also, we described the importance of an efficient organization of mobile devices configurations. Below, we provide technical details about these two aspects.

Partial policy evaluation.

We implemented a partial model checking tool from scratch. Indeed, at the best of our knowledge, no implementations of the algorithm presented in [4] are publicly available. Beside the mere implementation details, two aspects must be considered: transition system and policy representations. As a matter of fact, the procedure defined in [4] uses finite transition systems as modeling formalism. Our approach directly generates transition systems from CFGs. In fact, CFGs correspond to transition systems having a fixed, maximum branching degree, i.e., two.

Policy conversion is slightly more complex. Indeed, partial model checking deals with equational μ-calculus specifications. However, since the μ-calculus is extremely expressive, it is possible to express ConSpec policies in that formalism. These aspects required us to develop proper compatibility components, but they lay out of the scope of this work.

The partial model checking mainly applies to the policy management process. This part is not included in our current experimental activity. As a matter of fact, it heavily depends on the specific application scenario and it can only be evaluated against a real BYOD environment or via its application usage statistics. Future developments include setting up this kind of environment.

[11] `https://code.google.com/p/androguard/`

Database design and management.

The secure meta-market uses a database for storing data on the security configuration of the registered devices. Mainly, installation and removal of applications are the two activities impacting the security state of a device. In general, we expect the meta-market to handle an arbitrary amount of devices, each of them running a set of applications. The number of devices may vary according to several factors, e.g., the size and purpose of the organization adopting the meta-market. In addition, the number and the kind of mobile applications changes from device to device. According to recent statistics, it seems reasonable to assume that devices install dozens of applications[12].

Considering the situation depicted in Figure 2, we have that node N is represented by a corresponding row in a table called *policy_instance*. The entry for N refers to a *green_list* table which includes A_4. Similarly, A_2 and A_3 belong to the *green_list* tables associated to other two entries of *policy_instance*. These two represent nodes in which the installation of A_2 (A_3, respectively) is legal and leads to the configuration of node N.

The dimension of the configurations DAG and database can rapidly increase with the number of devices and applications. This might negatively impact the scalability of the meta-market. For coping with this issue, we introduces simplification techniques based on heuristics. For the time being, we included two of them in our prototype, i.e., equivalence-based and frequency-based node elimination. The former consists of an asynchronous process which visits the DAG looking for pairs of nodes referring to equivalent policies, i.e., such that they are satisfied (violated, resp.) by exactly the same traces. If such a pair exists, the process starts a procedure for collapsing them in a single record. Furthermore, the frequency-based elimination looks for nodes which are rarely used, e.g., because some of the applications they refer to no longer exist. Again, when one of these nodes is discovered, the process decides whether to remove it and, eventually, reorganize part of the DAG accordingly. Other heuristics can be also considered and included (notice that some could also be context-dependent).

5.4 Client-side components

Below we discuss the components and technologies hosted by the mobile devices registered to the meta-market.

Meta-market client application.

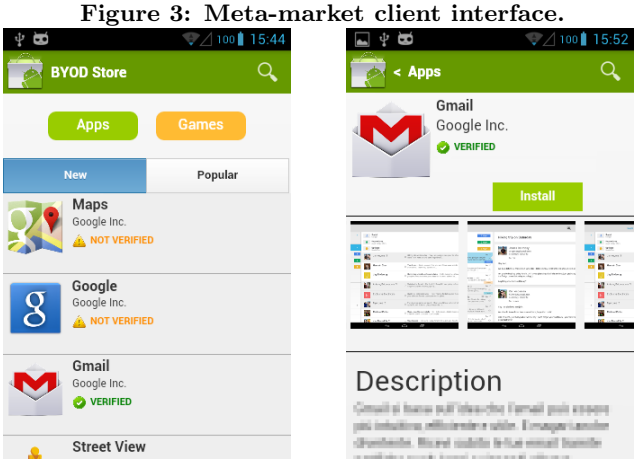
The meta-market client application provides access to the application installation/removal operations through a user interface which mostly resembles those of standard market client applications. Figure 3 depicts the client GUI in two different activities: applications browsing and application description. Briefly, the client is responsible for providing the user with details about the security of the available applications, e.g., green, VERIFIED labels identify secure applications. Also, it mediates the operations usually carried out by the application market clients, e.g., contents purchasing and refunding procedures[13].

[12] `http://www.phonearena.com/news/The-average-global-smartphone-user-has-downloaded-26-apps_id47160`

[13] Clearly, some of these tasks could need to be slightly reconsidered before integration in the meta-market. For the time being, our prototype does not support purchase refunding.

When the user selects an application for installation, the meta-market client triggers the request and, consequently, the meta-market workflow. Eventually, the meta-market returns a software package which, depending on the verification outcome, can require security instrumentation and monitoring. We describe these two activities of the client application in the next paragraph.

Figure 3: Meta-market client interface.



Application instrumentation and monitoring.

As discussed in Section 4.2, PEP inlining is a well known approach for enabling the security monitoring of mobile applications. Hence, some directions for implementing the instrumentation of Android application bytecode are viable. The current implementation of the meta-market relies on a modified version of *Redexer* [34] for instrumentation. Redexer is an OCaML implemented suite including several features for the assessment of some specific aspects of the Android application security, e.g., privilege exploitation analysis. Also, Redexer includes Dalvik bytecode parsing and rewriting facilities which we isolated and imported in our prototype. Although not designed for running on Android platforms, it can be ported by means of the Java native interface[14]. Briefly, the instrumentation is carried out by defining a *visitor* which inspects the bytecode and injects PEP instructions as described in Section 4.2. Eventually, the instrumented application is repackaged for installation. The PDP component is implemented as a background Android service running a policy interpreter. The service receives messages from the PEP instrumented in running applications. These messages are converted into events feeding the policy interpreter. Then, the interpreter simulates a corresponding step in the policy and returns a value representing the policy state, i.e., either *allow* or *deny*, as discussed in Section 4. An implementation of a ConSpec policy interpreter for mobile devices can be found in [14].

6. EXPERIMENTAL RESULTS

We applied the implementation of the meta-market to test the compliance of a large number of real-world applications against an instance policy, extracted from the US Government BYOD security guidelines presented in [17].

[14] See `https://sites.google.com/site/keigoattic/ocaml-on-android` for a technical explanation.

Roughly, experiments consist in performing multiple installation requests from the client application running on an emulated device. The meta-market reacts by running its workflow for achieving a policy-compliant installation. We evaluate the performance of each component taking part in the meta-market activity.
In the remaining of this section we discuss the experimental setup and test cases (Section 6.1), we provide results (Section 6.2) and, eventually, we critically discuss them (Section 6.3).

6.1 Experimental setup

The meta-market has been deployed on an Intel Quad-Core i7-2770 @ 3.40 GHz architecture, with 16 GB RAM and 1TB of hard drive, running Ubuntu Server 12.04 64bit. The server is also equipped with Java JDK 1.7.0_40 and Androguard 1.3 we customized for the model extraction. The meta-market client runs on an emulated Nexus 4 device mounting an ARM v7 CPU @ 1.5 GHz and 2 GB RAM.
We used the meta-market to analyze a group of 860 Android applications taken from the *"top selling free"* chart of the Google Play store. They include the most downloaded applications belonging to heterogeneous categories, e.g., business, entertainment and games.
In our experiments we considered the BYOD security policy given in Section 3 Also, we applied a time threshold of 300 seconds to the execution of each instance of the model checker. .

6.2 Results

We measure the time needed for each operation carried out by the meta-market server and client. Table 1 shows an excerpt of the experimental results[15]. A detailed explanation of each experimental phase is provided below.

Model extraction.

At first, the meta-market extracts the model of each application, using our customization of the Androguard tool. The CFG is generated parametrically w.r.t. the system action expressed in the security policy and all inter-application communications. Then, the server performs an optimization step consisting in simplifying the CFG by pruning unreachable nodes. Column T_{ext} reports the time needed (in seconds) for the generation of the final CFG while the size of the model is given in terms of number of nodes (column N) and edges (column E).

Promela encoding and model checking.

As explained in Section 5.2, the use of the Spin model checker passes through two consecutive phases, i.e., Promela encoding and verification. The duration of the encoding phase for each application is listed under T_{enc} in Table 1. The duration of the model checking verification step is reported in column T_{mc}. Moreover, we provide in column *Valid* the outcome of the model checker. We write YES to denote that the application successfully passed the verification, NO for the applications failing the verification, and Time Out (T.O.) for analysis that reached the time threshold.

[15]The complete results can be accessed online at `http://www.ai-lab.it/byodroid/experiments.html`.

Application instrumentation.

Application instrumentation is performed when the model checking fails in validating the application. The instrumentation execution times are reported under T_{ins} in Table 1. Moreover, since instrumentation may enlarge the original dimension of the software packages, we also report the package size change (column G), in percentage.

6.3 Discussion

The global values in Table 1 summarizes the overall statistics of our experiments. Although preliminary, the experimental results are promising. As a matter of fact, the meta-market validates more than 89% (column Y/N/TO) of applications within an average time of about 3 minutes ($\mu T_{ext} + \mu T_{enc} + \mu T_{mc}$). The remaining 11% (i.e. 94 applications), being rejected or leading to a time out, was successfully instrumented. The instrumentation process took an average time of 72 seconds (column μT_{ins}) and slightly enlarges the application packages by less than 0,1% on average (column μG).
It is worth noticing that our experiments have been carried out on a prototype implementation which can be further optimized. For instance, the definition of an optimized model extraction tool (different from Androguard) may reduce the extraction time; moreover, memory caching or task concurrency can improve the global performance.
Furthermore, consider that the meta-market validation is immediate when an application has been already checked in the current configuration. Hence, if devices share significant set of applications, we expect to observe a drastic increment of the skipped analysis phases.

7. CONCLUSION

We have presented a mobile application distribution system based on the notion of meta-market. The meta-market lays between standard, public application markets and the mobile devices. In practice, the meta-market guarantees that applications deployed on mobile devices do not violate a given security policy. This result is obtained by implementing a security workflow including model extraction and formal verification steps. When the verification rejects the application, the deployment process leads to the instrumentation of the potentially dangerous code. Instrumented code is then monitored for avoiding illegal behavior.
The whole system has been designed to be transparent to the users, the application developers, and the application markets. We have experimentally assessed the feasibility of the approach by testing the performance of an implementation of the meta-market on a large number of popular Android applications for both leisure and work. For the experimentation, a real world BYOD security policy taken from the US Government guidelines has been modeled. Eventually, we showed that the meta-market succeeds in detecting applications which can violate the security policy and in sanitizing them. All the activities carried out by the meta-market have been measured in terms of time and resources in order to witness the feasibility of our approach under realistic assumptions.
Future directions of this research include the extension of the experimental activity. Indeed, a crucial aspect of the meta-market is its capability to scale over a large number of devices and applications. In particular, we need to define new tests for showing the effectiveness of the PMC compo-

Table 1: Computation times and outcomes.

Application	Size(Mb)	T_{ext}(s)	N	E	T_{enc}(s)	T_{mc}(s)	Valid	T_{ins}(s)	G(%)
Adobe Connect Mobile	11,71	1,5	11	63	0,1	2,1	YES	-	-
Adobe Reader	6,63	11,5	36	157	0,6	2,1	YES	-	-
AndrOpen office	49,59	20,2	39	164	0,4	2,3	YES	-	-
Angry Birds	33,9	106,7	93	367	1,7	300	T.O.	38,7	0.14
Angry Birds Rio	32,61	106,6	93	367	1,7	300	T.O.	38,2	0.14
Angry Birds Seasons	42,27	112,3	94	373	1,7	300	T.O.	43,2	0.11
Candy Crush Saga	38,45	8,3	41	169	21	3,1	YES	-	-
Dropbox	5,59	32	33	168	11,1	4,9	YES	-	-
Extreme racing	8,02	62,5	112	403	2,1	2,4	YES	-	-
Facebook	15	30,2	25	112	0,2	2,3	YES	-	-
FB Messenger	12	314,3	41	180	12	4,3	YES	-	-
Fruit Ninja free	18,34	33,8	78	287	1,1	300	T.O.	29	<0,01
Gems Journey	11,19	53,6	51	201	0,8	300	T.O.	33	0,23
Gmail	3,57	23	36	157	0,4	2,1	YES	-	-
Google Chrome	24,6	36,3	54	230	10,1	2,3	YES	-	-
Google Play Music	7,53	68,1	64	276	1	2,1	YES	-	-
Google Street View	0,25	1,8	13	57	0,1	2,1	YES	-	-
Instagram	14,87	93,1	60	244	4	3,8	YES	-	-
LinkedIn	6,56	65,1	69	292	4,2	2,1	YES	-	-
Microsoft Lync 2010	3,61	3,5	8	41	0,1	2,1	YES	-	-
Microsoft Remote D.	4,50	16,622	15	64	0,1	2,1	YES	-	-
Mozilla Firefox	23,13	36,7	60	226	1,1	2,2	YES	-	-
OpenDocument Reader	1,61	45,8	44	188	0,6	3,0	YES	-	-
PocketCloud Remote	11,25	73,2	198	660	4,8	2,4	YES	-	-
Tiny Flashlight	1,28	16,4	61	223	0,8	3,2	YES	-	-
T.N.T. Italy	5,54	77,7	54	209	0,7	2,9	YES	-	-
TripAdvisor	6,31	24,1	74	269	1,1	2,1	YES	-	-
TuneIn Radio	6,58	154,4	174	649	8,3	13,3	NO	66	0,71
Twitter	5,18	42,7	62	256	10,7	2,3	YES	-	-
SMS backup & restore	1,33	35,5	74	298	1,3	2,9	YES	-	-
Skype	14,73	56	34	147	2,8	5,5	YES	-	-
Splashtop 2 Remote D.	17,78	58,46	90	357	1,7	2,2	YES	-	-
Spotify	3,65	8,1	18	79	2,3	2,1	YES	-	-
WhatsApp	9,75	369,4	223	715	8,8	2,1	YES	-	-
Global values									
# Applications	μSize	μT_{ext}	μN	μE	μT_{enc}	μT_{mc}	Y/N/TO	μT_{ins}(94)	μG(94)
860	11,53	69	71	271	4,8	114,9	89/0,2/10,8	72,6	0,08

nent in improving the verification and management of large, i.e., involving several applications, device configurations. Moreover, we want to apply the meta-market to real BYOD scenarios. As a matter of fact, the behavior and performances of some meta-market components, e.g., the partial model checker, can depend on the specific environment it has to deal with. For instance, the average number of applications per device and their occurrences may affect the database management. Moreover, we plan to investigate different instances of BYOD policies involving various aspects of the security of mobile devices. Among them, contextual policies, e.g., those referring to the geographical location of the device, need further investigation. Indeed, they may be useful for some kind of organizations.

Furthermore, we plan to research on the possibility of including novel optimization heuristics for the policy database management. We already investigated double implication and low frequency. Other approaches might include policy inclusion/implication, i.e., using a more restrictive policy which, for some reason, is more convenient or easier to handle. We believe that this represents a problem of general interest and it could be interesting to consider "garbage collection" techniques for this kind of policy management systems.

8. REFERENCES

[1] M. Abadi and C. Fournet. Access Control based on Execution History. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, 2003.

[2] I. Aktug and K. Naliuka. ConSpec – A formal language for policy specification. *Science of Computer Programming*, 74(1-2):2–12, Dec. 2008.

[3] A. Amighi, P. de Carvalho Gomes, and M. Huisman. Provably Correct Control-Flow Graphs from Java Programs with Exceptions. In *Formal Verification of Object-Oriented Software*, volume 26 of *Karlsruhe Reports in Informatics*, pages 31–48, Karlsruhe,

Germany, October 2011. Karlsruhe Institute of Technology.

[4] H. R. Andersen. Partial Model Checking (Extended Abstract). In *In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[5] Apple Inc. Secure Coding Guide, 2012. `https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf`, accessed on 4-Dec-2013.

[6] A. Armando, G. Costa, and A. Merlo. Formal modeling and reasoning about the android security framework. In C. Palamidessi and M. Ryan, editors, *Trustworthy Global Computing*, volume 8191 of *Lecture Notes in Computer Science*, pages 64–81. Springer Berlin Heidelberg, 2013.

[7] A. Armando, G. Costa, A. Merlo, and L. Verderame. Bring your own device, securely. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1852–1858, New York, NY, USA, 2013. ACM.

[8] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Breaking and fixing the android launching flow. *Computers & Security*, 39, Part A(0):104 – 115, 2013. 27th IFIP International Information Security Conference.

[9] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino. Securing Java with Local Policies. *Journal of Object Technology*, 8(4):5–32, June 2009.

[10] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. Comput. Secur.*, 9(3):217–250, Jan. 2001.

[11] T. Blasing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak. An Android Application Sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62, 2010.

[12] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[13] K. D. Cooper, T. J. Harvey, and T. Waterman. Building a control-flow graph from scheduled assembly code. Technical report, Department of Computer Science, Rice University, 2002.

[14] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter. Runtime monitoring for next generation Java ME platform. *Computers & Security*, 29(1):74–87, 2010.

[15] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS .NET Run Time Monitor. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, Dec. 2009.

[16] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, Jan. 2008.

[17] 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. Technical report, White House, August 2013. Available at `http://www.whitehouse.gov/digitalgov/bring-your-own-device`.

[18] N. Dragoni, F. Massacci, T. Walter, and C. Schaefer. What the heck is this application doing? - A security-by-contract architecture for pervasive services. *Computers & Security*, 28(7):566 – 577, 2009.

[19] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[20] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[22] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 53–65, London, UK, UK, 2001. Springer-Verlag.

[23] N. Hardy. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[24] G. Holzmann. *The Spin model checker*. Addison-Wesley Professional, first edition, 2003.

[25] G. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[26] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[27] R. V. Koskinen and J. Plosila. Applications for the SPIN Model Checker – A Survey. Technical Report 782, Turku Centre for Computer Science, Lemminkäisenkatu 14 A, 20520 Turku, Finland, September 2006.

[28] B. Long, J. Dingel, and T. N. Graham. Experience Applying the SPIN Model Checker to an Industrial Telecommunications System. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 693–702, New York, NY, USA, 2008. ACM.

[29] M. Nauman, S. Khan, and 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, pages 328–332, New York, NY, USA, 2010. ACM.

[30] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[31] J. Oberheide and C. Miller. Dissecting the Android Bouncer. In *SummerCon*, June 2012. `http://jon.oberheide.org/research/`.

[32] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[33] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[34] N. Reddy, J. Jeon, J. A. Vaughan, T. Millstein, and J. S. Foster. Application-centric security policies on unmodified Android. Technical Report UCLA TR 110017, University of California, Los Angeles, Computer Science Department, July 2011.

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

[36] Ph. Schnoebelen. The Complexity of Temporal Logic Model Checking. In Ph. Balbiani, N.-Y. Suzuki, F. Wolter, and M. Zakharyaschev, editors, *Proceedings of the 4th Workshop on Advances in Modal Logic (AIML'02)*, pages 481–517. King's College Publications, 2003.

[37] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *SIGOPS Operating Systems Review*, 37(5):15–28, Oct. 2003.

[38] 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, held in Dagstuhl, September 1996*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.

[39] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22nd USENIX Security Symposium*, pages 559–572, 2013.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, Nov. 2006.

[41] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[42] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on Android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, 2011.