Securing the "Bring Your Own Device" Paradigm

Alessandro Armando, *University of Genoa and Bruno Kessler Foundation*

Gabriele Costa and Luca Verderame, *University of Genoa*

Alessio Merlo, *eCampus University and University of Genoa*

The current mobile application distribution model cannot cope with the complex security requirements of the emerging "bring your own device" (BYOD) paradigm. The authors propose a secure metamarket architecture that supports the definition and enforcement of BYOD policies and discuss a promising prototype implementation tested under realistic conditions.

Two factors have contributed to the widespread proliferation of mobile devices. First, the emergence of modern mobile operating systems such as iOS, Android, and Windows Mobile has pushed development and commercialization of a wide range of powerful devices. Second, the market-based mobile application distribution model has spawned rich public stores in which users can browse through millions of apps and install those most suited to their needs.

During mobile application installation or execution, the user is normally required to grant some permissions, such as whether to allow location-based services or push notifications. This discretionary access control (DAC) of resources and services suffers from numerous security vulnerabilities,[1,2] yet these vulnerabilities are often eclipsed by the benefits of the app's flexibility and customizability, which are enabled by DAC policies.

In recent years, companies have shown increasing interest in and tolerance for employees using their own mobile devices at work. This trend has led to a new "bring your own device" (BYOD) paradigm in which employees are allowed access to sensitive resources through their personal devices as long as they agree to comply with the organization's security policy. However, the DAC policies encouraged by current mobile OSs sharply contrast with organizations' relatively strict mandatory access control (MAC) policies, which are designed to prevent damaging errors or fraudulent activity by users. Indeed, many third-party apps could expose company resources to great risk.

Researchers in academia and industry have proposed various novel mechanisms to enforce security policies on mobile devices. For Android applications, for instance, Aurasium is a prototype technology that enforces some predefined security policies through code instrumentation,[3] while SCanDroid is a tool that statically detects known, potentially dangerous data flows.[4] Although effective, these approaches target specific, user-centric security flaws and only consider a fixed policy model. Thus, they do not directly apply to the BYOD paradigm.

Commercially available security solutions from Apple (www.apple.com/ipad/business/it/byod.html), Samsung (www.samsung.com/global/business/mobile/solution/security/samsung-knox), Symantec (www. symantec.com/

Published by the IEEE Computer Society

mobile-management), Blackberry (http://it.blackberry.com/business/software/bes-10.html), and MobileIron (www.mobileiron.com/en/solutions), among others, support the BYOD paradigm. Some of them, such as Samsung Knox, provide an isolated environment for corporate apps, whereas others, including those offered by Symantec and MobileIron, restrict the privileges of apps not developed with a certain software development kit.

Most of these solutions provide a mobile device management (MDM) service that can block or even reset devices violating the security policy. The policy is often specified through application black- or whitelists, but with no behavioral analysis in place, these lists—and hence the policy—are often based on vague concepts. AppThority Risk Manager (www.appthority.com/products/app-policy-management) uses a different approach that assigns a TrustScore to apps based on common risk factors such as encryption methods, SMS access, or contacts, but the IT manager still must make the final, crucial decision about an app's trustworthiness.

Here, we describe a novel approach to securing the BYOD paradigm based on the concept of a *secure metamarket* (SMM).[5] An SMM mediates access to traditional application markets, keeps track of the apps installed in registered devices, and—as its core functionality—checks whether new apps respect a given organization's security policy. In principle, this last task is daunting as the problem is generally undecidable. Nevertheless, we have obtained practically viable solutions by leveraging a fruitful combination of state-of-the-art static analysis via model checking and runtime monitoring via code instrumentation. We have experimentally confirmed these results by running our prototype framework against hundreds of popular apps. To the best of our knowledge, no other framework—be it a research prototype or commercial solution—allows for flexibly defining the BYOD policy while automatically providing formal guarantees.

MOBILE APPLICATION MARKETS

Application markets such as Google Play, the Apple Store, and the Windows Store that let users browse through myriad apps and install them on their devices with "a single click" have unquestionable advantages. Finding and acquiring software is straightforward, and installation is standardized for all apps and requires little expertise.

However, application developers typically have no information about users' security requirements. Also, application markets' security concerns are usually limited to viruses and other common types of malware. Nevertheless, security violations involving nonmalicious software have been reported—for example, confused deputy attacks in which even trusted apps can become entry points for security breaches. Thus, usage control rules for private mobile devices in corporate environments should be evaluated against a device's specific application profile. (Hardware tampering and intentional system attacks by the device's owner or some other active adversary mostly involve intrusion detection and insider attack prevention, which we do not consider here.)

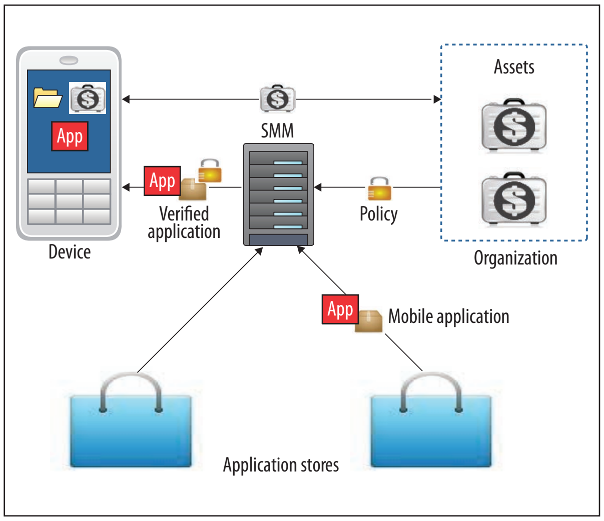

Figure 1. "Bring your own device" (BYOD) scenario with a secure metamarket (SMM). The SMM aims to guarantee that applications installed on a personal mobile device comply with an organization's security policy.

Being centralized repositories, application markets facilitate the deployment of security analysis tools and methodologies for the apps they host. For instance, the Apple Store checks all submitted apps before deciding whether or not to publish them, while Google Play employs Bouncer, an application scanning service, to mitigate malware diffusion.[6] Although helpful, these solutions suffer from numerous limitations. For instance, Tielei Wang and his colleagues showed how to forge a malicious application and publish it in the Apple Store,[7] while Yajin Zhou and his coauthors likewise reported a method for easily circumventing Google Bouncer.[8]

To effectively support the BYOD paradigm, application markets must specify security policies spanning multiple apps, check whether an app complies with a given security policy, and ensure that the app never violates that policy at runtime. Moreover, security solutions should be immediately applicable to existing mobile devices. Hence, deep OS customizations are out of scope, while application-level modifications are acceptable.

SECURE METAMARKET

An SMM aims to guarantee that a set of applications installed on a mobile device comply with a given security policy. Figure 1 depicts the expected scenario. Using its existing ICT infrastructure (dashed box on the right), an organization wants to enable use of personal devices. It defines a security policy, but personal devices are outside the ICT infrastructure's scope and thus cannot be controlled by traditional enforcement mechanisms. Moreover, devices can run third-party applications that could, intentionally or unintentionally, violate the policy. An ideal solution should ensure that all organization members' mobile devices comply with the policy, avoid leaving security-critical decisions to users, and preserve the devices' usability as much as possible.

Security by contract

Nicola Dragoni and his colleagues previously considered the formal verification and enforcement of fine-grained security policies with respect to mobile applications using their security-by-contract (SxC) model.[9] As Figure 2a shows, after creating an app, a *code producer* (developer) generates a contract (P.3) and attaches this to the software (P.8). The code producer also computes *evidence* (P.7) that the contract (P.5) properly describes the code (P.4) and includes this in the application package (P.9). A *code receiver* (mobile device) leverages the evidence to check contract correctness (C.0) and then uses the contract (C.1) to validate the app against its own security policy (C.3). If either contract (C.2) or policy (C.4) verification fails, the app receives limited or even no access privileges—for example, via security monitoring. If instead both succeed, the app can run unrestricted.

SxC provides an elegant solution to securing mobile code but, by decentralizing some of the security checks, suffers from several problems:

- Contract and policy verifications are computationally challenging tasks that might be too demanding for resource-constrained mobile devices.
- Developers generate contracts without knowing end-user policies and thus must provide large, inclusive descriptions of their apps. Behavioral annotations in contracts might be irrelevant for most policies.
- The SxC workflow demands that mobile devices carry out activities that could invalidate the requested app—for example, contract and policy verification. Instead, users should decide whether to download and install an app after considering its safety and the effect it will have on their device.
- Contract generation, evidence computation, and code instrumentation are nontrivial tasks that will increase development costs. Considering that many mobile apps are released by small software companies—sometimes consisting of a single developer—these costs could inhibit innovation.

For all of these reasons, we believe that SxC is not directly applicable to mobile application markets or BYOD environments.

SMM architecture

By centralizing the security checks on mobile devices, the SMM model avoids most of the problems associated with SxC: it relieves the burden on

- mobile devices from computationally expensive security analyses,
- users from security-critical decisions, and
- application developers from nonfunctional, security-related activities.

Our proposed model is optimized for the BYOD scenario in Figure 1. An organization deploys or subscribes to an SMM, which ensures that registered devices only run policy-compliant code. The SMM architecture consists of a client and server. The SMM client is installed on the device upon registration; it disables and replaces the clients of other stores to prevent insecure installations. In addition, once installed, the SMM client collects system details, such as installed applications and OS version, and asks the SMM server to verify the device's configuration. If the device complies with the organization's security policy, then it successfully joins the BYOD network. The SMM server is responsible for maintaining the configuration of all registered devices and mediating access to application markets, permitting installation only of policy-compliant apps.

SMM workflow

Figure 2b shows the SMM's workflow. The SMM server retrieves an application package from the public market and extracts its model (B.0), which describes all of the app's security-relevant behaviors. The server then validates the behavioral model (B.2) against the organization's security policy (B.3) via model checking (B.4). If verification succeeds, the server updates the device configuration and delivers the app to the SMM client for installation (B.6); if verification fails, the server tags the app for runtime monitoring (B.5) before delivering it. When the client receives the application package, it checks whether the app has been tagged for runtime monitoring (C.0). If this is the case, the client performs code instrumentation (C.1)—that is, it injects security checks into the application code—using information attached by the server; otherwise, it directly installs the app.

The server extracts application behavioral models by building a control-flow graph (CFG), in which nodes represent program blocks and edges represent control flow across those blocks. As models only contain security-relevant operations, a model's size directly depends on application interactions with the system and on the security policy.

Application security policies must be specified through an appropriate language that supports static as well as runtime verification techniques. Because such policies

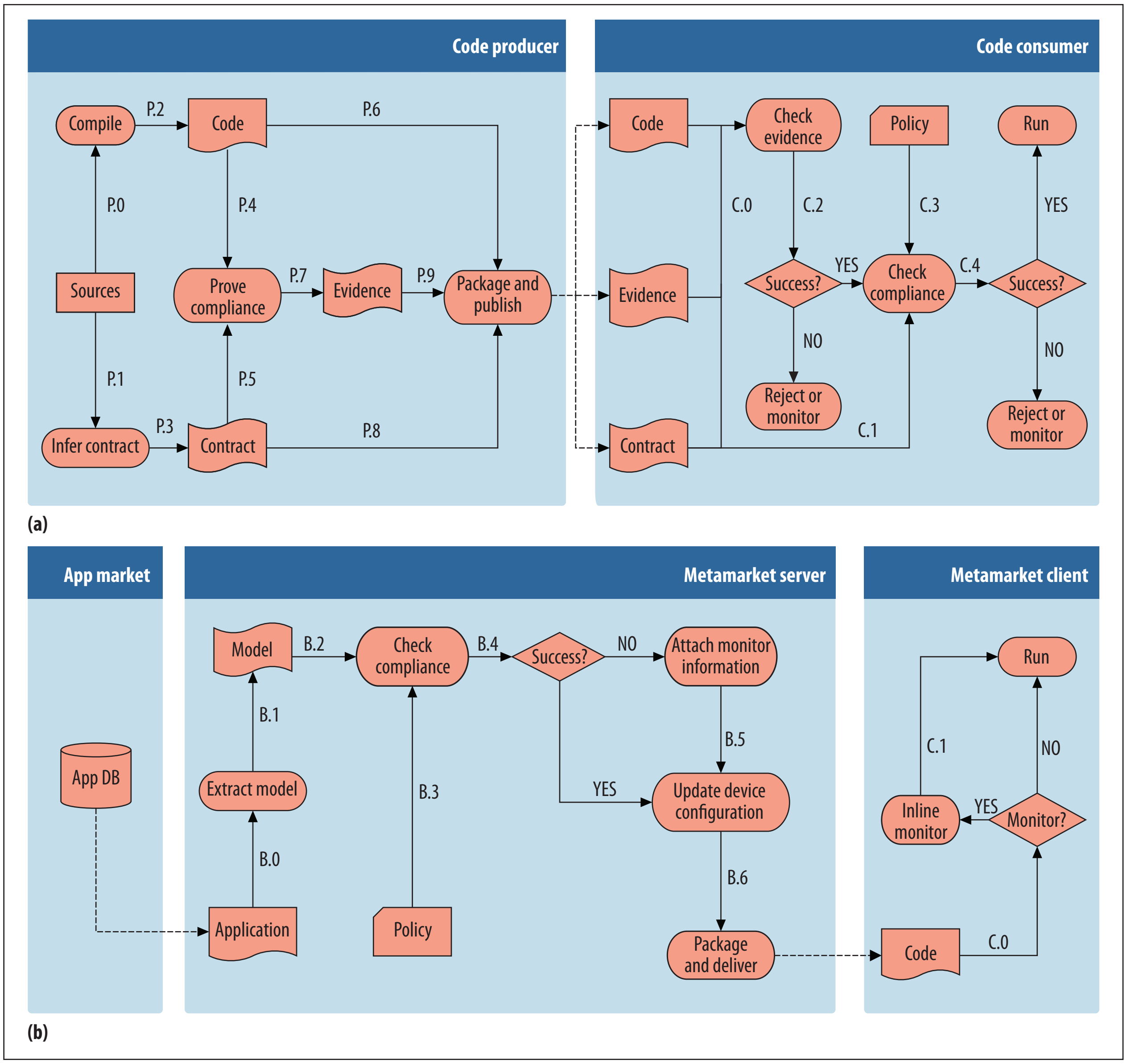

Figure 2. Workflows for two approaches to securing mobile applications: (a) security-by-contract (SxC) and (b) secure metamarket (SMM).

encompass several heterogeneous aspects, the language should make it easy to compose subpolicies. Eventually, it must also support specification of both allowed and unwanted behaviors.

Security policy verification involves checking that no execution trace of the application behavior model violates the policy. In general, a verification technique can return false positive and negative results. False negatives—dangerous apps that pass the verification process—can be avoided if the models are complete, that is, if they overapproximate actual behavior. In the case of SMM, the CFG extraction procedure guarantees this property.[10]

Applications that fail security policy verification must be controlled at runtime. Controls should be neither CPU nor memory intensive, and ideally the suspicious application and not the OS should sustain the computational overhead. In addition, runtime monitoring support must be noninvasive and require no OS customization.

Although the SMM model's possible outcomes are the same as those for SxC—an application executes with no security constraints if and only if it actually complies with the security policy—the SMM model offers some important advantages. The most important one is that only code instrumentation is carried out on the device itself. The verification process fully executes on the SMM server, thereby preserving one-click installation.

BYODROID

We have created a prototype SMM, BYODroid, that supports Android OS and BYOD security policies. It assumes the client to be an Android device and the application market to be Android compatible, such as Google Play. When possible, we used off-the-shelf technologies for the components that implement the workflow in Figure 2b.

CGF extraction

BYODroid extracts CFGs from an application's bytecode. Dalvik bytecode, typically obtained by compiling Java code, is the intermediate language interpreted by the Android virtual machine—namely, the Dalvik VM. BYODroid uses a customized version of Androguard (https://code.google.com/p/androguard), a state-of-the-art Android package analyzer, to generate CFGs that overapproximate the set of security-relevant execution paths.

Policy specification

For security policy specification, BYODroid uses ConSpec, which is amenable to both static verification and runtime monitoring.[11] A ConSpec specification consists of a set of rules, each describing how the security state changes when a certain action occurs. Policy actions often represent security-critical access operations that a program performs, such as reading a file or opening a connection. However, the ConSpec syntax is flexible enough to model other policy aspects such as obligations (for example, "the sales office must be copied on emails sent to customers") and accountability (for example, "access to corporate URLs requires the user's authorization").[11]

Model checking

To verify policy compliance, BYODroid leverages Spin (http://spinroot.com/spin/whatispin.html), a state-of-the-art model checker that has been successfully applied to numerous real-world problems. BYODroid translates both the application CFG and the ConSpec policy into Promela, Spin's specification language. Toward this end, we have adapted existing techniques applying to CFGs and ConSpec specifications for encoding finite state agents in Promela.[12] To prevent unbounded computations that are unacceptable in the SMM workflow, the model checker includes a timeout mechanism. Thus, the model checker produces three possible results:

- YES—the model complies with the policy;
- NO—the model violates the security policy; or
- TIMEOUT (TO)—the time threshold has been reached.

BYODroid's model-checking process does not allow false negatives to occur—that is, no policy-violating application can pass analysis.

Code instrumentation

Runtime monitoring is enabled through code instrumentation. The BYODroid client performs such instrumentation directly on the device, injecting security checks in the code of applications that failed the policy verification step—that is, model checking resulted in either NO or TO. The security checks wrap each of the application's policy-relevant instructions. If the execution flows through a security check, the ongoing operation is controlled. In the event of a policy violation, the security check raises an exception. In this way, all security policies are enforceable: instrumented applications can never violate them. (Liveness properties can also be enforced, though this aspect of code instrumentation is beyond the scope of this article.) BYODroid relies on Redexer (www.cs.umd.edu/projects/PL/redexer) for the instrumentation procedure. Redexer includes several features for manipulating Android apps, such as Dalvik bytecode parsing and rewriting operations.

The BYODroid client has been implemented as a market application that replaces the native Android apps distributed through stores such as Google Play (see www.ai-lab.it/byodroid for snapshots and a description of the client GUI). This minimally affects the application level of the Android OS architecture, with no customization of core components including the Dalvik VM and Linux kernel (www.android-app-market.com/android-architecture.html). Furthermore, our approach does not disable, but simply enhances, Android's native security mechanisms.

After installation, the BYODroid client's first task is to inspect the device configuration: the client collects data about all currently installed applications and queries the BYODroid server to check whether the device can safely join the BYOD network. For the device to be admitted, the owner might have to remove or update some of the apps. (Optionally, disclaimers, licenses, and contracts can be prompted during this process.)

The BYODroid client provides the user with security details about installable applications—for example, green labels identify secure, verified apps. It also carries out usual market client operations, such as purchase and refund procedures. When the user selects an app for installation, the client triggers the request for a software package and, consequently, the SMM workflow. The BYODroid server returns a package that, depending on the outcome of the policy verification step, might require code instrumentation. If so, the server provides the client with the original app along with information supporting the instrumentation process.

It is worth noting that code instrumentation invalidates the application's original signature. Still, a valid signature is necessary to install apps on Android devices. Since the code instrumentation is known to the BYODroid server, it computes a new signature using a *proxy key*—a private key generated by the server for each application producer.

```
SECURITY STATE
SESSION Str [11] agency_host = “agency.gov/”;
SESSION Obj agency_url = null;
SESSION Bool connected = false;

/*(R1) No download of business data on device*/
AFTER Obj url = java .net.URL.<init>(Str [64] spec) PERFORM
      (spec.contains(agency_host)) → {agency_url := url;}
      ELSE → {skip;}
AFTER Obj url = java.net.URL.<init >(Str [0] protocol, Str [64] host, Nat [0] port,
   Str [0] file) PERFORM
      (host.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(int i) PERFORM
      (!connected) → {skip;}

/*(R2) “When in Doubt, Delete it Out”*/
AFTER Obj file = java.io.File.createTempFile() PERFORM
      (true) → {file.deleteOnExit();}

/*(R3) No transfer of sensitive data to nonagency devices*/
BEFORE android.bluetooth.BluetoothSocket.getOutputStream() PERFORM
      (!connected) → {skip;}
```

Figure 3. ConSpec specification of the BYOD security policy used in the experimental assessment of BYODroid.

As part of the instrumentation support information, the server sends the new signature to the BYODroid client, which checks the original signature to verify the integrity of the code it is about to instrument. Finally, the client instruments the code, attaches the server-provided signature to the new application, and installs it.

EXPERIMENTAL ASSESSMENT

To assess our approach's effectiveness, we ran BYODroid against a test suite of 860 popular Android applications and a US government BYOD security policy (www.whitehouse.gov/digitalgov/bring-your-own-device). The apps—in categories ranging from business to entertainment to games—came from Google Play's “top free” list.

Figure 3 shows the ConSpec specification of the BYOD policy. Rule (R1) states that users cannot store sensitive data on their device. If an application creates a URL object that points to sensitive data sources, the app cannot write to local files anymore. Rule (R2) states that the device must delete temporary data, which could include sensitive information, at the end of each application session. Finally, rule (R3) prohibits data transfer via Bluetooth.

As indicated earlier, ConSpec also supports accountability policies. For instance, we can easily add a new clause to the end of R1's program block that logs the application's writes to local files instead of preventing them:

```
BEFORE java.io.FileOutputStream.write
   (Nat[0]i) PERFORM
       (!connected) → {skip;}
       (connected) → {Log.append(“Accessed”
          + agency_url);}
```

The experiments consisted of multiple application installation requests to the BYODroid server by the BYODroid client on an emulated device. For each app, we measured the execution time of each activity carried out by the server (model extraction, translation to Promela, and model checking) and client (code instrumentation). The time limit for model checking was fixed at 5 minutes. The server was deployed on a 3.40-GHz Intel quad-core i7-2770 with 16 Gbytes of RAM and a 1-Tbyte hard drive running Ubuntu Server 12.04 (64-bit version). The client ran on an emulated Nexus 4 device with a 1.5-GHz ARMv7 CPU and 2 Gbytes of RAM.

Figure 4 summarizes the results; a full report is available at www.ai-lab.it/byodroid/experiments.html. T[ext] indicates CFG extraction time, T[conv] Promela specification generation time, T[mc] model-checking time, and T[ins] code instrumentation time.

We clustered the applications according to the size of their models in terms of the number of CFG edges—for example, the cluster 380–400 includes all apps whose CFG has between 380 and 400 edges—and normalized them (calculated the average values in each cluster). Clusters contain various numbers of apps, and there is no direct relation between an application package's physical dimension and the size of its model. Clusters in the top part of the plot are significantly less dense (some are even empty), as most apps generate small models (the average size is 271). Although model size is relevant, other aspects also affect execution time. For instance, as model extraction involves unpacking and processing all application code, application size matters.

The experimental results are encouraging: the BYODroid server validated 89 percent of the apps with an average execution time of about 3 minutes (T[ext] + T[conv] + T[mc]). Even shorter execution times could be obtained through optimizations—for example, caching models according to request frequency—that we have not yet implemented. The remaining 11 percent of applications, although rejected or timed out by the server, were successfully instrumented by the client. The code instrumentation process took an average time (T[ins]) of 72 seconds and very slightly enlarged the application packages (less than 0.1 percent on average). Only those apps that failed model-checking analysis experienced the delay, which occurred only once during installation.

In general, larger models lead to longer computations. However, model size affects BYODroid activities in different ways. For instance, Promela conversion (T[conv]) is almost constant, whereas model extraction (T[ext]) tends to increase noticeably. However, most apps generate small models, and large models are extremely rare. A smaller model size also positively impacts model-checking activity (T[mc]), which successfully terminates in most cases.

Although promising, our experiments must be carried out on real devices, as we expect hardware configurations to impact execution times.

Our work demonstrates the feasibility of extending the mobile application distribution model to provide security guarantees for the BYOD paradigm by means of an SMM. Future research includes the incorporation of policy specification language in BYODroid to support context-dependent policies and an extensive assessment of the computational cost of code instrumentation on mobile devices with various hardware profiles.

We also plan to address security issues related to application configurations—in particular, we hope to create an SMM workflow to determine whether a new app, together with those already installed on the device, comply with a given security policy. To tackle the well-known problem of application collusion, the SMM analysis process would benefit from the addition of compositional reasoning techniques such as partial model checking, which allows for customizing a security policy by partially evaluating it against a model.[13] Iterating this process makes it possible to inject multiple models into a policy tailored to the applications hosted on a platform.

References

1. A. Armando et al., "Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures)," *Proc. 27th IFIP TC 11 Information Security and Privacy Conf.* (SEC 12), 2012, pp. 13–24.
2. E. Chin et al., "Analyzing Inter-Application Communication in Android," *Proc. 9th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 11), 2011, pp. 239–252.
3. R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," *Proc. 21st Usenix Security Symp.* (Security 12), 2012; www.usenix.org/system/files/conference/usenixsecurity12/sec12-final60.pdf.
4. A. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid: Automated Security Certification of Android Applications," unpublished manuscript, Univ. of Maryland, College Park, 2009; www.cs.umd.edu/~avik/projects/scandroidascaa.
5. A. Armando et al., "Bring Your Own Device, Securely," *Proc. 28th Ann. ACM Symp. Applied Computing* (SAC 13), 2013, pp. 1852–1858.
6. H. Lockheimer, "Android and Security," blog, 2 Feb. 2012; http://googlemobile.blogspot.it/2012/02/android-and-security.html.
7. T. Wang et al., "Jekyll on iOS: When Benign Apps Become Evil," *Proc. 22nd Usenix Security Symp.* (Security 13), 2013, pp. 559–572.

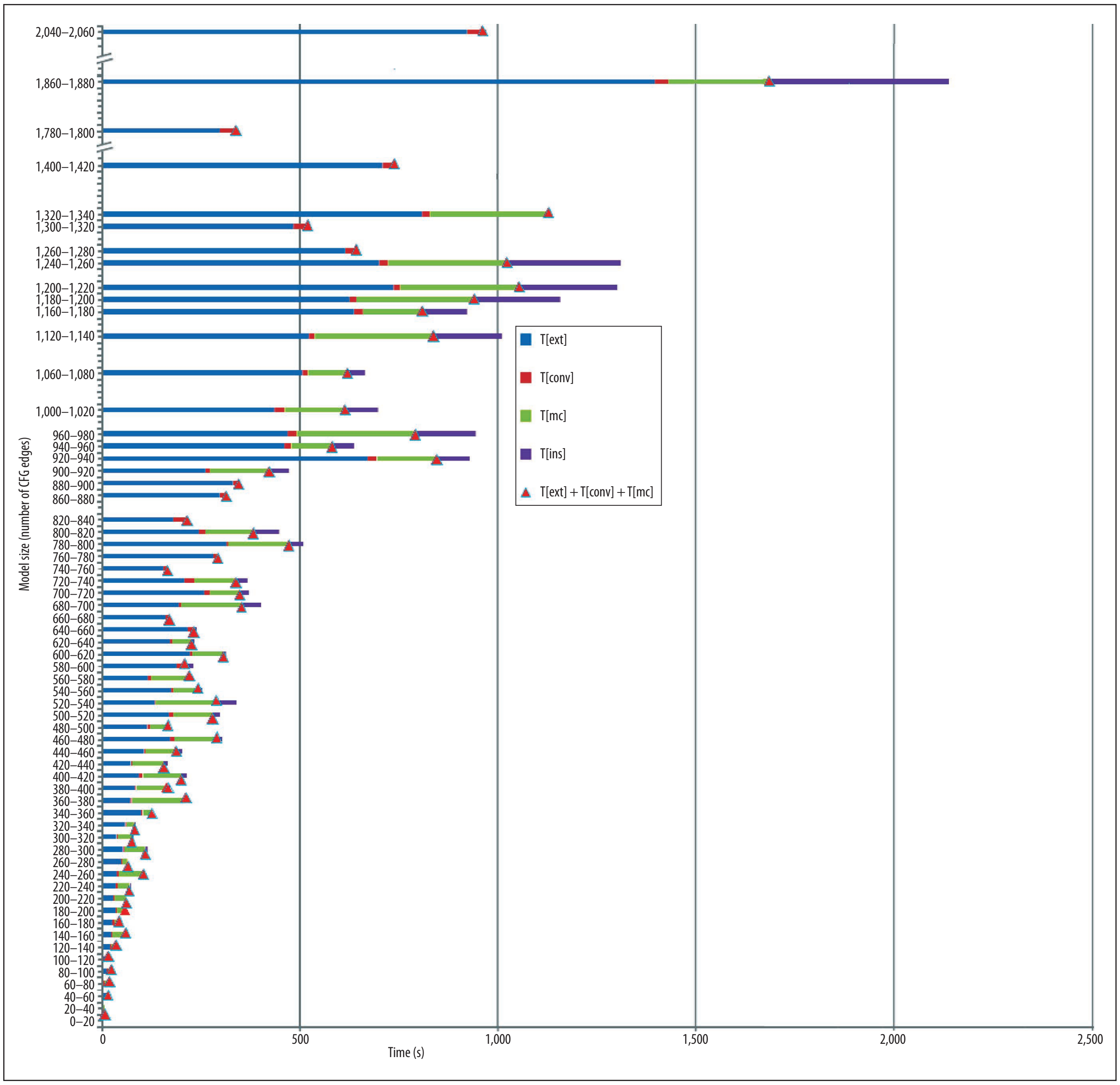

Figure 4. Execution time of application installation request activities performed by BYODroid against a test suite of 860 popular Android apps, clustered according to model size. T[ext]: control-flow graph (CFG) extraction time; T[conv]: Promela specification generation time; T[mc]: model-checking time; and T[ins]: instrumentation time.

8. Y. Zhou et al., "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," *Proc. 19th Ann. Symp. Network and Distributed System Security* (NDSS 12), 2012; www.internetsociety.org/hey-you-get-my-market-detecting-malicious-apps-official-and-alternative-android-markets.
9. N. Dragoni et al., "What the Heck Is This Application Doing?—A Security-by-Contract Architecture for Pervasive Services," *Computers & Security*, vol. 28, no. 7, 2009, pp. 566–577.
10. M. Bartoletti, "Securing Java with Local Policies," *J. Object Technology*, vol. 8, no. 4, 2009, pp. 5–32.
11. I. Aktug and K. Naliuka, "ConSpec—A Formal Language for Policy Specification," *Science of Computer Programming*, Dec. 2008, pp. 2–12.
12. R.V. Koskinen and J. Plosila, *Applications for the SPIN Model Checker—A Survey*, tech. report 782, Turku Centre for Computer Science, Finland, 2006.
13. H.R. Andersen, "Partial Model Checking," *Proc. 10th Ann. IEEE Symp. Logic in Computer Science* (LICS 95), 1995, pp. 398–407.

Alessandro Armando *is an associate professor as well as cofounder and leader of the Artificial Intelligence Laboratory (AI-Lab) in the Department of Informatics, Bioengineering, Robotics, and Systems Engineering (DIBRIS) at the University of Genoa, Italy. He also heads the Security and Trust research unit of the Center for Information Technologies at the Bruno Kessler Foundation in Trento, Italy. His research focuses on automated reasoning and its application to the modeling, design, and verification of security-critical systems. Armando received a PhD in electronic and computer engineering from the University of Genoa. He is a member of the IEEE Computer Society. Contact him at alessandro.armando@unige.it.*

Gabriele Costa *is a researcher in the DIBRIS AI-Lab at the University of Genoa. His research interests include language-based security, formal methods for security analysis, distributed systems security, and security verification and enforcement. Costa received a PhD in computer science from the University of Pisa, Italy. Contact him at gabriele.costa@unige.it.*

Luca Verderame *is a PhD student in the DIBRIS AI-Lab at the University of Genoa. His research focuses on information security in mobile devices. Contact him at luca.verderame@unige.it.*

Alessio Merlo *is an assistant professor at eCampus University in Novedrate, Italy, and an associate researcher in the DIBRIS AI-Lab at the University of Genoa. His research focuses on Web, distributed systems, and mobile security. Merlo received a PhD in computer science from the University of Genoa. He is a member of the IEEE Computer Society and ACM. Contact him at alessio.merlo@unige.it.*