Blueprints of an Automated Android Test-bed

François Gagnon[1,⋆], Jérémie Poisson[1], Simon Frenette[1], Frédéric Lafrance[1], Simon Hallé[2], and Frédéric Michaud[2]

[1] Cégep Sainte-Foy, Québec, Canada
[2] Thales Research and Technology, Canada

Abstract. This paper discusses the automation of experiments on the Android platform. The most obvious choice for such a test-bed is virtualization as it provides an easy solution to several challenges, e.g., configuration, automation, clean up. However, virtualization sometimes imposes limitations, for instance, with respect to a realistic environment. Although this paper focusses mainly on our virtual test-bed for Android (named AVP for Android Virtual Playground) it also explores a solution for a physical test-bed. Both test-beds were built with the primary concern of being able to control (as much as possible) the devices participating in the experiment. Moreover, the virtual test-bed provides a wide variety of data collection possibilities while the physical one has a leaner design allowing to perform experiments in a more ad hoc way (with the devices available in a room).

Keywords: Android, Virtualization, Test-Bed, Network Experiment, Automated

1 Introduction

In the last decade, mobile devices such as smartphones have penetrated the consumer world at a speed no other technology ever has. Nowadays, most people use mobile devices for various tasks: communications, banking, shopping, etc. Several actors are taking advantage of this massive acceptance of mobile technology. Application developers try hard to obtain a market share with a specific idea/feature. On the other hands, malicious hackers now have access to a very interesting tool, providing entry into various networks and access to an abundance of personal (and often sensitive) information.

In this paper, we share our ideas and experiences in developing two fully automated test-beds for the Android platform. Our test-beds aim to be generic enough to be used for many different purposes. Be it to test a mobile application on multiple devices for compatibility (both at the library level and the graphical level), or to inspect the security aspect of an application. The test-beds are large in scope to take advantage of the different modules already available instead of focussing on a particular aspect. Our first (and main) solution is named Android Virtual Playground (AVP), pointing to the fact that it is a framework allowing us to perform several different activities (hence the term playground). Our second solution explores the possibilities of performing experiments on physical devices in a more ad-hoc way (for instance, by recruiting the willing devices available during a meeting).

⋆ Corresponding author: frgagnon@cegep-ste-foy.qc.ca

We strongly believe automated test-beds for mobile platforms are an important component for speeding up the research related to mobile applications. This has been the case with automated test-beds for malware analysis in the PC world [8].

The paper is structured as follows. First, Sect. 2 discusses existing approaches for Android virtualization and experimentation. Sections 3 to 5 present different aspects of our Android Virtual Playground tool (Sect. 3 gives a brief overview, Sects. 4 and 5 detail the core of AVP in terms of capabilities and usability). Then, our physical test-bed is presented in Sect. 6. Finally, Sect. 7 concludes with final remarks and Sect. 8 discusses avenues for future work.

2 Related Work

Our work can be related to two different fields: Android emulation solutions (see Sect. 2.1) and Android experimentation frameworks (see Sect. 2.2). AVP aims to instrument the various virtualization methods available for Android and provide automated control of user defined experiments with a rich set of functionalities. This section presents projects related to AVP.

2.1 Virtualization Solutions

Below is an overview of the main solutions used for Android virtualization. Through AVP, we aim to harness as many of those as possible.

- **Google** distributes a modified version of the open-source QEMU[3] emulator [1]. It can run native Android apps under the x86, ARM and MIPS architectures. Its purpose is to allow Android developers to test their apps on a "real" Android system. This emulator has several advantages over other emulators. It is supported by Google, and as such offers features that many other emulators do not provide. For example, it allows two emulator instances to communicate with each other (e.g., SMS, calls) or with the host machine using documented mechanisms. On the other hand, it is known to be slow, particularly when running under a non-x86 architecture.
- **Microsoft** just released [9] an Android emulator with it's Visual Studio development environment. Their emulator is based on Google's open source emulator so it should support most features available in a real mobile device. Apparently, the Microsoft version is an improvement over the Google one, possibly solving the poor user experience problem encountered with the Google version. Microsoft emulator was not available when we started our project, so it was not considered as a virtualization solution. However, we are planing to include it in AVP quite soon.
- **Genymotion** is a French company distributing a virtualization solution for Android. It relies on the well-known VirtualBox[4] hypervisor [12] and on the compatible images that can be built from the Android source code. Those images are

[3] http://qemu.org

[4] https://www.virtualbox.org/

then modified to add Genymotion's own apps and services that can be used to control the virtual machine remotely. They offer a different set of capabilities than the Google emulator. In Genymotion, the virtual machines are much more responsive since they take advantage of virtualization features offered by VirtualBox (instead of being emulated).
- **AndroX86** [5] is an open source project offering ports of the Android OS for different x86 platforms. As a result, Android can be run in conventional hypervisors such as *VMWare Workstation* as well as bare metal on some supported hardware configurations (e.g., Lenovo Ideapad S10-3T). This solution provides great performance at the cost of functionalities inherent to a mobile device (e.g., no support for SMS).
- **Manymo** is a software as a service solution for Android virtualization. It allows one to launch Android virtual machines through a web interface. These machines can then be controlled through a command-line interface and run Android apps. The main limitation of this platform is the number of launches and concurrent devices available, which are both very small for non-paying users.

AVP differs from these projects as it is not meant to provide new ways of virtualizing the Android environment. Instead, AVP builds on top of existing virtualization solutions to take advantage of their strengths. AVP is designed to support multiple underlying virtualization options.

2.2 Android Experimentation Frameworks

Several tools to perform experiments in Android exist; mainly focussing on android sandboxing for dynamic (behavioral) malware analysis. Some are presented below to contrast them with AVP.

- **MegaDroid** [11] is a project by Sandia National Laboratories to simulate a large amount (i.e., 300,000) of Android devices using virtualization on a cluster of PCs. Little technical information is known about Megadroid except the fact that it can manipulate GPS information to simulate device movements and can be used for security purposes. In essence, MegaDroid is probably the closest to AVP, although a comparison is difficult since almost no technical information is available regarding their project.
- **TaintDroid** [3] is a modification of the Android API to allow tracking of sensitive data through tainting. Tainting is a technique in which data from sensitive sources is tainted. As the data moves around the system (through variable assignments, files and inter-process communication), the taint is propagated. If tainted data escapes the system (through an internet connection, SMS, etc.), a notification is made system-wide, so that monitoring tools can detect it. TaintDroid focuses entirely on taint propagation and does not provide automation[6].
- **AASandbox** [2] is an Android malware sandbox implemented at kernel-level. It is entirely automated but focuses only on system call tracking.

[5] http://www.android-x86.org/

[6] AVP can be used to automate the experiment flow of TaintDroid

– **DroidScope** [13] is another Android malware sandbox located at hypervisor level (a modified version of QEMU). DroidScope offers an interesting level of automation. However it focuses entirely on malware analysis; providing information on taint propagation, memory state and process state.
– **Bouncer** is Google's own solution designed to prevent malicious applications from being added to the Play Store. Since it is proprietary software, little is known about its inner workings [7,10]. It seems to run Android applications under Google's emulator for a few minutes and observe their behavior. Upon suspicious behavior, control can be transferred to a human for further analysis.
– **JOESandboxMobile**[7] is another proprietary solution for Android malware analysis. It offers a fully automated platform, but one which is focussing entirely on malware analysis.
– **monkeyrunner**[8] is Google's API to automate the control of their modified QEMU emulator. This project does not focus on malware analysis and do provide a more generic capability of creating experiments. However, the API is limited to one emulator while AVP can integrate several.

Most of the tools available for Android application analysis focus on a very narrow view of behavior (e.g., taint tracking, malware behavior). In contrast, AVP aims to provide the widest possible behavioral view. In particular, by integrating several existing and custom tools as AVP modules, it is possible to gather more information. AVP targets a wide range of experiments, not only those related to malware analysis.

Static solutions to malware analysis (code-based analysis) are not discussed here since their offline nature sets a very different scope from AVP. However, static analysis solutions can easily be included in any experiment system, including AVP.

AVP distinguishes itself from existing solutions in three ways. First, it is platform-agnostic: it can leverage several different virtualization technologies, and has been built with expansion possibilities in mind. Second, it fully automates the process of experimentation in an Android environment; allowing to perform controlled, custom and repeatable experiments. Finally, it offers a wider scope than current solutions by focussing not only on malware analysis, but on a more general concept of experiment execution.

3 AVP in a Nutshell

This section provides an overview of the test-bed. More details are provided in the following sections. AVP was designed in order to perform different types of experiments with Android devices in a virtual environment. The main focus was automating the test-bed and collecting data during the experiments.

Figure 1 presents the high-level workflow of AVP.

First, the user specifies an experiment, both in terms of the Android virtual devices (AVDs) to use and the actions to perform. The experiment specification (what we call a *scenario*) is given to the system through an XML file (1).

[7] http://www.joesecurity.org
[8] http://developer.android.com/tools/help/monkeyrunner _concepts.html

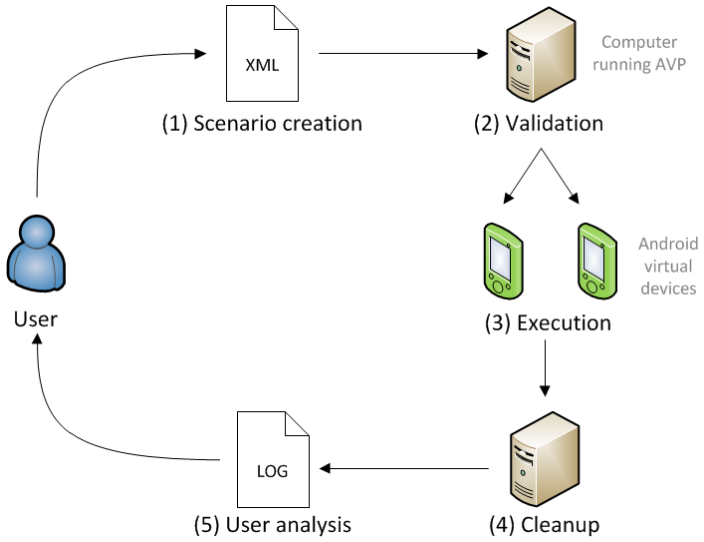

Fig. 1. AVP workflow

Next, the system validates that the scenario is correct (2). Validation is performed at different levels:

- Syntactic validation of the XML document (e.g., are the commands understood?)
- Resources validation (e.g., are all the required AVDs available?)
- Semantic validation of the action sequence (e.g., can an action really be executed in the current experiment state?)

Whenever validation fails, the program terminates with an error message. AVP also has a set of softer validation rules that are considered best practice but can be violated (e.g., timer issues, resources liberation). In the latter case, the system will provide warnings but still perform the experiment.

Third is the execution phase (3), which represents the heart of the test-bed. At this step, AVP manipulates AVDs through the instrumentation of various Android virtualization technologies. it executes the AVDs and their applications; and then collects experiment data (which is why the test-bed was developed in the first place). Most collected data is linked to the behavior of the AVD or one of its applications. Section 4 will provides details regarding the possible "actions" and the data collection capabilities of AVP.

At the end of the execution phase, the test-bed automatically performs a clean-up phase (4). The inner workings are actually quite similar to those of the execution phase, but we keep it conceptually distinct because it is out of the user's control.

Finally, once the experiment is over, the user can visualize and/or process the data collected during the experiment (5). Data visualization can also be done in real time during the experiment.

4 AVP Capabilities

This section deals with the core of our test-bed: the actions it can perform automatically and the data it can collect (Sect. 4.2). But first, the virtualization solutions supported in AVP are presented (Sect. 4.1).

4.1 Virtualization

When building AVP, we did not want to create our own virtualization environment. Instead, we wanted to take advantage of the best existing solutions to emulate Android devices. While studying existing virtualization technologies, it quickly became apparent that their objectives are not the identical. For instance, Google's modified[9] version of the QEMU emulator aims to provide an experience close to a physical Android device in terms of functionalities (e.g., SMS and calls). On the other hand, *Genymotion* [10] (Formerly *AndroVM*/*BuildDroid*[11]), based on *VirtualBox*, provides a more fluid emulation targeting Android apps developers requiring only limited functionality (e.g., no support for SMS) but a more realistic user experience.

Instead of choosing one technology, AVP was designed in such a way that it could use different underlying technologies. As a consequence, the user has access to a wider range of functionalities. As Sect. 5.1.1.3 will show, this choice has an impact on the semantic validation of a scenario.

AVP currently supports four Android engines:

- Google's modified QEMU emulator [Google emulator]
- Genymotion's free version emulator [Genymotion emulator]
- Our own custom modified Google emulator [Modified Google emulator]
- VMWare Workstation with x86 ports of Android

All of the above rely on ADB (Android Debug Bridge) which provides a common mechanism to interact with all of them. However, they all have particularities that need to be instrumented differently by AVP. In the rest of the paper, unless explicitly mentioned, we focus on the Google emulator as it is the most complete in terms of functionalities.

4.2 Actions

The set of actions supported by our test-bed can be divided into four categories:

- Lab actions (see Sect. 4.2.1). Actions performed by our test-bed software (possibly on a set of AVDs).
- Device actions (see Sect. 4.2.2). Actions executed by an AVD (as instructed by AVP).
- Data collection actions (see Sect. 4.2.3). Actions allowing to gather (behavioral) data of an application or device.
- User simulation action (see Sect. 4.2.4). Actions providing simulation of user interaction.

[9] Available through the ADT (Android Development Tools) environment.

[10] http://www.genymotion.com

[11] http://androvm.org

4.2.1 Lab Actions. Actions performed by the virtual lab manager are listed in Table 1 together with their mandatory and optional arguments (resp. parameters and options) and discussed in the following subsections.

Table 1. Lab actions

Type	Action	Parameters	Options
AVD Manipulation	CreateAVD	dev, kernel, cpu, ram, snapshot	sdCard, height, width
	DeleteAVD	dev	
	StartAVD	dev	snapshot
	StopAVD	dev	
	UnlockAVD	dev	
	Push/PullFile	dev, labPath, devPath	
	ShellCommand	dev, cmd	
App Manipulation	InstallApp	dev, appPath	reinstall
	StartApp	dev, appPath	
Lab Manipulation	Wait	time	
	WaitForBoot	dev	timeout, interval
	WaitForInstall	dev, appPath	timeout, interval
	WaitForUser		
	CallDevice	dev, src	
	SendSMSToDevice	dev, msg, src	

4.2.1.1 AVD Manipulation. AVD manipulation actions are actions that aim to manipulate AVDs.

Most actions are self-explanatory, however, some have details of interest. The creation of an AVD requires several parameters: the device name (*dev*), the Android version to use (*kernel*), the *cpu* (x86 or arm) and so on. Starting AVDs can optionally be modified through the use of snapshots. AVP allows the user to either boot (or not) from a snapshot and save (or not) to a snapshot when closing (snapshot configuration must be specified when starting the device). The support for snapshots provides important advantages:

- It significantly speeds up the booting process, which is interesting in the context of performing a large amount of experiments in batch.
- It allows containment of an experiment, that is, the effect of an experiment has no repercussions on the subsequent experiments (e.g., leaving an application installed). This is of tremendous importance for data collection experiments.

With the Google emulator, there is at most one snapshot for each AVP, so there is no choice when loading/saving a snapshot. VMWare Workstation, however, supports multiple snapshots through ID manipulation. At the moment, AVP does not support IDs for snapshots. Hence, even for VMWare Workstation, only the current snapshot is "visible" inside AVP.

4.2.1.2 App Manipulation. One of the key element with mobile devices is the manipulation of applications, referred to as "apps" on mobile platforms. AVP supports two actions related to manipulating apps: *InstallApp* and *StartApp*. When installing an application, the user must provide the path to the apk file (*appPath* parameter); optionally, if the application is already installed on the device, a reinstall can be forced (*reinstall* option).

In general, when manipulating apps (e.g., install, start), the path to the apk file is needed. Often, it is simply to retrieve and parse the manifest in order to extract the necessary information to perform the action (e.g., full package name).

4.2.1.3 Lab Manipulation. Lab manipulations are actions that are handled by the testbed. The *Wait* action allows the user to specify an amount of time for the lab to pause before executing the next action. A similar action, *WaitForUser*, can be used to pause the lab until the user perform a specific action. The two actions above are not dependant on any virtualization technology. Hence, they are always available.

Starting an AVD (boot) is a process that can take a long time[12]. Therefore, it is important for the lab to wait until the device is booted before proceeding further. An additional action called *WaitForAVDBoot* achieves this.

Similarly, the installation of an application is not instantaneous. AVP can wait until an application has been installed through the *WaitForAppInstall* action.

Note that since the "wait" actions are blocking by nature (while all other actions are non-blocking) they come with an associated timeout threshold. When the threshold is reached, the corresponding action is considered a failure and the experiment is aborted. Some wait actions also come with a an interval defining the frequency at which the verification should be performed.

Finally, two actions can be performed by the lab that will directly affect the AVDs: *SendSMSToAVD* and *CallAVD*. This will result in sending an SMS (resp. call) to a specific AVD as if it would have been sent by another Android device. Both action require the spoofed source (*src*) of the communication and, for SMS, the actual message (*msg*) to be sent.

4.2.2 Device Actions. The last type of actions supported by AVP are device actions. These are actions that will be executed by the AVDs themselves. To achieve this, we developed an Android app (named *commandCatcher*) that the lab can install on an AVD and then invoke to request specific tasks to be executed by that AVD. The invocation is done through the intent broadcasting mechanism which can be triggered from outside the AVD.

The list of device actions can be extended quite easily by adding new capabilities to our *commandCatcher* library. Table 2 provides the list of device actions currently available in AVP.

If the *Call* or *SendSMS* action targets a device also running inside AVP (*dst*), the two devices will actually communicate with one another. On the other hand, specifying

[12] From a few seconds to a few minutes depending on the physical machine and the virtualization acceleration.

Table 2. Device actions

Action	Parameters	Options
SendSMSToNumber	dev, msg, dst	
SendSMSToDevice	dev, msg, dst	
CallNumber	dev, dst	
CallDevice	dev, dst	
NavigateToURL	dev, url	

a phone number not inside AVP will result in only the first half of the communication establishment taking place (nobody is at the receiving end).

4.2.3 Data Collection. While executing an experiment, the main objective of AVP is to collect data. To this end, we have identified some information worth having and developed the corresponding mechanisms to collect it. Table 3 list the information that AVP can collect and their corresponding supported actions (further details are provided below).

Table 3. Data collection actions

Information	Action	Parameters	Options
SMS Activity	*Start/StopRecordSMS*	dev	
Call Activity	*Start/StopRecordCalls*	dev	
Network Traffic	*Start/StopRecordNetworkTraffic*	dev	interface
Running Processes	*Start/StopRecordProcesses*	dev	
System Calls Activity	*Start/StopStrace*	dev, process	
TaintDroid Activity	*Start/StopRecordTaintDroid*	dev	
Visualization	*TakeScreenShot*	dev	interval

AVP can gather SMS and call activities (both outgoing and incoming). In doing so, it gathers the sender and recipient of the exchange, and for SMS the actual data exchanged. For the Google emulator, this is done by installing an app inside the AVD that will hijack SMS and call activities. However, we noticed that it is possible (and quite easy) for a malicious app to bypass this monitoring mechanism by avoiding the built-in functionalities for SMS and call. To circumvent this limitation, we slightly modified the Google emulator to have a lower level access to SMS and call activities, which is much more difficult to bypass. This is the main difference between the stock Google emulator and our modified version.

Network activity is recorded in a pcap file to allow further forensic of Internet access. Of particular interest is the IP visited (e.g., publicity sites or botnet Command and Control centers) as well as data downloaded (e.g., malicious payloads).

The list of running processes plus their usual information[13] can be obtained. this list is automatically refreshed each time there is a change in process activity. This provides the evolution of processes during an experiment.

System calls are monitored through the *strace*[14] utility. This provides the list of system calls for a given process. System calls can then be analysed to detect anomalous sequences [5].

AVP can take screenshots of an AVD. This can be useful in determining whether the experiment was successful. The *interval* option allows to take multiple screenshots at the given interval. No interval means only one screenshots.

Finally, through the use of *TaintDroid* [3], AVP is able to track sensitive data (e.g., phone number, device unique ID) leaking from an AVD. This, however, requires a specific AVD configuration (through the use of particular image files). Thus, AVP treats *TaintDroid* as its own virtualization technology (similar to Google emulator except it supports the TaintDroid data collection process and instruments the start mechanism in a different way).

Other pieces of data are always collected by AVP; this information serves mainly as debug information. More specifically, the *logcat*[15] stream and system logs from the scenario execution (executed actions and issued commands) are collected.

In the future we plan to integrate new data collection capabilities to AVP. In particular, by adding support for tools similar to *TaintDroid* (see Sect. 8).

4.2.4 User Simulation. Table 4 lists a few AVP actions allowing to simulate user interaction, a crucial element to gather behavioral information regarding the tested app. Two avenues are covered: random simulation through the *monkey*[16] application exerciser tool and human-specified action through recording and replaying action sequences. The later requires human interaction during an experiment to record a sequence of actions; afterwards, the action sequences can be automatically replayed in several experiments. Although AVP is able to replay an action sequence perfectly well with the Google emulator it is not yet stable with VMWare Workstation (action coordinates are sometimes shifted on the device screen).

Table 4. User simulation actions

Action	Parameters	Options
RunMonkey	dev, nbEvents	appPath, package
StartRecordUserInteraction	dev	
StopRecordUserInteraction	dev	file
ReplayUserInteraction	dev, file	

[13] process ID, owner ID, process Name, etc.
[14] http://sourceforge.net/projects/strace/
[15] The Android logging system: http://developer.android.com/tools/help/logcat.html
[16] http://developer.android.com/tools/help/monkey.html

monkey needs the number of events to generate. Optionally it is possible to restrict the events to one application (by providing directly the package name or the path to the apk file where the package name can be extracted).

Recording user interaction will store the action sequence in a file that can be used to replay the action sequence.

5 Using AVP

As shown in Fig. 1, the user interacts with AVP at two moments. Initially, the user must create an XML scenario describing an experiment (see Sect. 5.1); this is the input to AVP. Then, after the experiment has been executed, the user can visualize the data collected (see Sect. 5.2).

5.1 Scenario

A scenario is made of two parts: devices and actions. Table 6 (see Appendix A) provides an example of a scenario file which is used throughout this section. The first part specifies which Android devices will be used in the experiment (only 1 in the example of Appendix A). For each device, the user must provide an ID, the underlying technology and the device name. The ID (e.g., *A*) is used through the scenario to identify this device. The technology refers to the virtualization technology used to power the device (e.g., *taintdroid* meaning the Google emulator with the TaintDroid images is used). Finally, the name (e.g., *testAVD1*) refers to the AVD filename on the host.

The second part is the sequential list of actions. All actions are non-blocking (that is AVP will go to the next action right after having issued the command to execute the specified action) with the exception of *wait* actions.

For instance, the scenario of Appendix A illustrates an experiment for the behavioral analysis of an application. It contains four stages of actions: setup, start collecting information, application execution, clean up.

In the setup phase, the lab will power on (*StartAVD*) the necessary AVDs (e.g., device A as defined above). In order to maintain this AVD clean, we use the option to boot from an existing snapshot but not persist the changes, hence leaving the snapshot unmodified after the experiment. After waiting for device "A" to boot (*WaitForAVDBoot*), the unlock command is issued (*UnlockAVD*).

Then, data collection mechanisms are activated. Depending on what we expect to observe, we can decide what data to collect. In the running example, we want to observe the communication behavior of an App, hence the following are interesting: SMS (*StartRecordSMS*), calls (*StartRecordCalls*), network traffic (*StartRecordNetworkTraffic*) and tainted data (*StartRecordTaintDroid*).

Now the application is ready to come in play. The app (located at X:/test.apk) must first be installed (*InstallApp*). Once the installation is over (*WaitForAppInstall*), the app can be launched (*StartApp*). Then, we can use the *Monkey* tool to perform 5,000 random actions in the application context (*StartMonkey*). Then, we wait an arbitrary 5 minutes to record the application behavior.

Finally, it is possible, but not mandatory, to stops the data collection mechanisms and the AVD. In the current example, it is done explicitly in the scenario. However, upon reaching the end of the scenario, the lab will automatically close each open handle in a specific way (e.g., open AVD are turned off, data collection feeds are stopped).

5.1.1 Scenario Validation. Before the execution of an experiment begins, the scenario must be validated. Validation occurs at three different levels: syntactic, resources, and semantic.

5.1.1.1 Syntactic Validation. The syntactic validation is straightforward enough in XML. First, the document structure is tested (e.g., is there a list of devices before the list of actions?). Then, each action is validated by first making sure the action exists and then verifying that the parameters to those actions are correct, that required parameters are present, and that they all have meaningful values. A last syntactic check is performed to make sure all the devices referenced in the actions belong to the list of devices defined in the scenario.

5.1.1.2 Resources Validation. Resources are validated using a simple validation process which makes sure the devices listed in the scenario exist and that they can be used by the specified technology. The Google emulator, Modified Google emulator, and TaintDroid emulator can all support the same set of QEMU-based AVDs. However, Genymotion and VMWare Workstation each have their own set of AVDs. In addition, the validation process verifies that the applications used in the different actions exist in their given locations.

5.1.1.3 Semantic Validation. The semantic validation is much more complex. AVP currently covers only a small portion of the possible semantic rules. Here are a few examples of semantic validation rules:

- Most actions cannot be performed on an AVD that is not running.
- Some actions can only be executed if the AVD involved is of a specific technology. For instance, actions related to calls and SMS are not applicable to a *Genymotion* AVD. Table 7 (see Appendix B) provides a complete list of supported actions by virtualization technology.
- Some actions are strongly dependent on one another. For instance, *StartApp* should always be preceded by a corresponding *InstallApp*.

AVP also produces warnings that will not prevent the execution but indicate a potential mistake in the scenario for the following reasons:

- When a wait action is usually necessary but omitted in the scenario (e.g., between starting an AVD and executing an action on that AVD).
- When a stop action occurs without a preceding corresponding start action.
- When a start action has no following corresponding stop action. Although the lab will compensate during the clean-up phase, this might indicate an error on the user's part.

5.2 Data Analysis

AVP will collect a lot of information during an experiment. Moreover, the type of information collected will vary from one experiment to the other (depending on which data collection mechanisms are used in each experiment and the data actually generated by the AVDs/apps). Since data collection is at the core of AVP, the log format is easily searchable and AVP comes with visualization tool for the logs. Each piece of data collected during an experiment is stored in a text file specifically related to the nature of the data, then text files are aggregated in a more generic file based on the similarity of their content. Finally, all the information is aggregated into a master file.

Each entry of each log file contains five elements:

- *Time*: the timestamp of the event.
- *Thread*: the identification of the AVP thread that generated the event.
- *Location*: the origin (e.g., the method, class, library) of the event.
- *Level*: the importance of the logged event (*Trace*, *Debug*, *Info*, *Warning*, *Error*).
- *Message*: a message describing the event.

5.2.1 Log Files. For each device, the information related to the lab execution flow for that specific device is stored in a file (then, this information is aggregated in common file for all devices). The following log files are generated:

- Output/Error information of the processes used by AVP (e.g., *adb.exe*, *aapt.exe*) in one file per process (and again aggregate all processes information in a single file).
- Sent/Received SMS are stored in a per-device basis (and SMS activities for all devices are aggregated in a single file.) The same occurs for calls, system calls, running processes, and tainted data.
- *Logcat* output per device (and aggregated).
- A number of logging files related to the lab execution (e.g., debug, warnings, errors).
- *Pcap* files containing network traffic logs on a per device basis only.

5.2.2 Data Visualization. To facilitate the visualization of collected data, AVP includes a generic log viewer (see Fig. 2). The log viewer will automatically create one view per log file. This feature allows us to add the capabilities of logging new information (i.e., creating new log files) without the need to modify the log viewer. The log viewer allows for searching and filtering of data.

6 Physical Test-Bed

Although virtualization technologies allow us to build a flexible and efficient test-bed for Android experiment, it has some limitations. Among those limitations, we encountered to following:

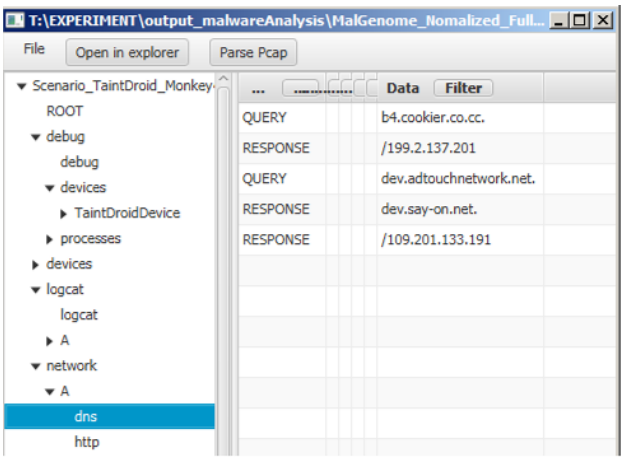

Fig. 2. AVP log viewer

Sandbox-evading malware: malware authors are working hard to prevent the reverse-engineering of their creations. One way to do so is to prevent the malware to run normally inside an emulated or virtualized environment, which is often a sign that the malware is being observed by security researchers. The malware simply detects the presence of the emulated or virtualized environment (by looking for known signatures in the device configuration, e.g., ANDROID_ID, or performing timing analysis) and plays dead.

Low-fidelity execution: Emulators are not perfect and can have different behaviors than physical devices, especially for corner cases on complex platforms. For instance, the ARM processor used in almost every Android smartphone has many instruction modes (i.e. ARM, THUMB, ThumbEE, Jazelle, etc.) and switching from a mode to another has complex effects on the state of the processor. Emulators such as the ubiquitous QEMU do not replicate these effects perfectly. Emulators are also often incomplete. For instance, the baseband processor (i.e. radio) inside Android smartphones could be an attack vector [6] but it is not supported at all by QEMU.

Device specific vulnerabilities: Many smartphone vendors heavily customize the base Android OS to add functionalities and differentiate themselves from the competition. These customizations are often called "skins" and vary a lot from a vendor to another. For instance, Samsung phones provide TouchWiz and HTC phones use Sense. These skins come with a lot of additional software that may contain specific vulnerabilities and their popularity marks them as interesting targets for malware. Emulators are not yet adapted to those different skins.

Using a physical test-bed is essential in order to address those limitations. We considered two approaches for a physical test-bed: Sect. 6.1 proposes an extension to AVP to control physical devices connected to the lab (connected mode), while Sect. 6.2 discusses an agent-based approach where devices are not, a priori, controlled/configured by the experiment manager (disconnected mode).

6.1 Connected Mode (AVP extension)

The easiest way to provide a test-bed with physical Android devices is to extend AVP, adding the ability to control physical devices. AVP sees physical devices as a specific

"virtualization" technology. To be included in an experiment, a physical device must be reachable through ADB from the experiment server (i.e., connected via USB debugging mode).

However, AVP lacks the flexibility to build an experiment using dynamically available resources. We are targeting the Android platform and most people carry such a device on themselves at all time. Hence, it would be interesting if a test-bed could be built on the spot with whatever devices are available in a room (e.g., a corporate meeting) without having to preconfigure all the devices. Our second approach tackles that problem and is described in details in Sect. 6.2.

6.2 Disconnected Mode (Agent-based approach)

The disconnected mode consists of an agent that can be easily deployed on Android devices. This grants us the ability to establish an experimentation network on the fly using one pre-configured device as the experiment server and other random devices as clients. There is no computer nor internet connection needed, the only requirements are that every device participating to the experiment must:

- have a web browser;
- be able to open a network connection with the experiment server;
- allow installation of applications from untrusted sources.

The experiment manager need to have the server application installed on his device and must distribute a given HTTP URL to the clients. Clients will then be able to join the experiment by connecting to the given URL.

Deploying and running an experiment can be done in eight steps (see Fig. 3). The following subsections detail each step.

The main advantage of using the disconnected mode over other techniques is the fast deployment of an experimentation environment with less installation and configuration overhead. One important thing to note is that the use of the disconnected mode is not limited to any specific version of Android and can be done using any Android device, even an average person's phone and can be setup in minutes.

6.2.1 Launch the server application.

It serves both as a web server for clients to connect, and a dashboard to control the experiment. After starting the server, the experiment manager loads an experiment scenario. This is step 1 in Fig. 3.

As it was the case with AVP, an experiment is described by a "scenario" file (an example is given in Table 5). The scenario, which is made up of two parts, describes the information to gather during the experiment (capture section) and the sequence of actions assigned to the devices (tasks section).

Capture section : Used to select which logs are being recorded and dumped into a file during the experiment execution, allowing us to perform log analysis after the experiment. This is simply a list of XML tags with options. Each tag represents an information to capture. It is possible to specify which devices should capture a specific type of data; but, by default, all devices will capture all the data specified in the experiment.

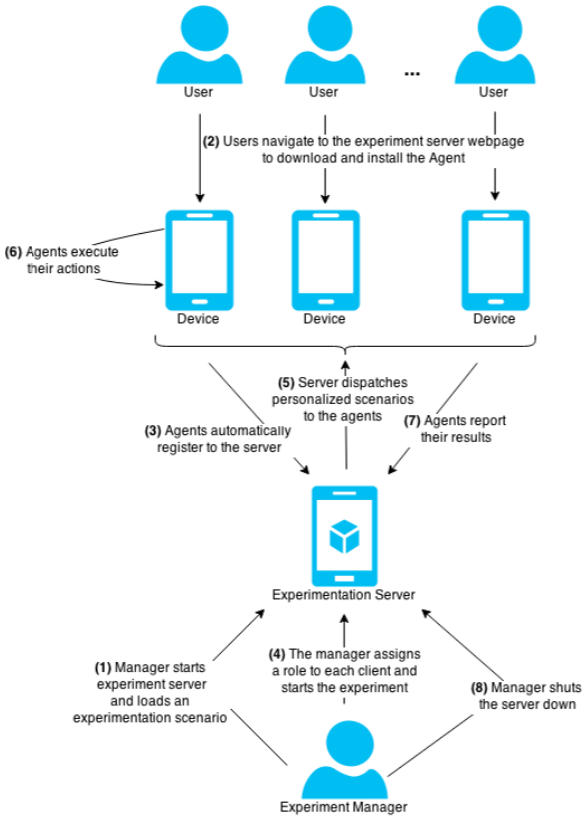

Fig. 3. Physical Test-Bed Workflow

Tasks section : Here, tasks are being assigned to devices that are part of the experiment. The task section contains multiple *activity* tags describing every single actions that must be performed during the experiment. Each activity contains a *device*, an *action* and a *data* parameter. The device parameter indicate which device will perform this action, a device is referred to by its abstract ID (see Sects. 6.2.4 and 6.2.5). The action parameter defines the actual action to be executed by the agent, the easiest way to do this is to rely on the Android intent mechanism. The data parameter allows to further specify the actions by providing extra information to the intent. We are taking advantage of the flexibility and the openness of the Android Intent structure and, since it is at the core of every Android application communication, it allows us to perform actions either at the operation system level or the applicative level.

6.2.2 Install Agent. The users must install a software agent on their device to participate in an experiment. This is done by clicking a download link on the server webpage

Table 5. Generic Experiment Scenario

```
<experimentScenario>
    <capture> <logCat/> </capture>
     <tasks>
         <activity device="A" action="android.intent.action.VIEW" data=
         "geo:46.802659,-71.324712?q=46.790893,-71.284103 (Cégep Sainte-Foy)"/>
         <activity device="B" action="android.intent.action.VIEW" data=
         "http://www.cegep-ste-foy.qc.ca/cybersecurite"/>
    </tasks>
</experimentScenario>
```

(see step 2 in Fig. 3). Agent installation will start automatically once the download is complete; all the user has to do is confirm he wishes to install the application.

6.2.3 Agent Registration. Once the agent installation completes, the user is automatically taken to another server webpage with a special link bounded with the agent. When the user clicks the link the agent application is launched on his device with a special intent that automatically register the client to the experiment manager (Fig. 3 step 3).

6.2.4 Role Assignation. As devices register themselves to the server, the experiment manager sees them in the server dashboard. The dashboard (see Fig. 4) allows the manager to assign a role to each device. Given an experiment scenario, the list of possible roles is the set of all abstract device IDs used in that scenario plus the spectator role (which never execute any action). By default, all devices receive the spectator role (denoted by the ID "-"), but the manager can reassign the roles through the dashboard.

Once role assignation is completed (each role must be taken by exactly one device), the manager can start the execution of the experiment.

6.2.5 Personalized Scenarios. The first step to execute the experiment is to generate a personalized scenario for each registered device (see step 5 of Fig. 3). A device scenario is generated by copying all the information of the related experiment scenario and adding an identification section at the beginning. The identification section simply indicate what is the role (abstract ID) of the device reading this scenario. For instance, to personalize the experiment scenario of Table 5 using role A, the following line must be added before the *capture* section:

<identification role="A"/>

6.2.6 Experiment Execution. Upon receiving a scenario, the agent will inform the server it is ready to proceed with the experiment. When all non-spectator devices are ready, the server gives the start signal. Then, each agent will go through the action list in the scenario; executing those for his role and skipping the others (step 6 of Fig. 3). One

Fig. 4. Role Assignation in Experiment Manager

of the main drawback of our current test-bed is the lack of flexibility regarding the synchronization of action execution. Currently, a scenario action is executed (or skipped) every 10 seconds. As future work, we plan to provide a choice to the experiment manager regarding the strategy to handle synchronization.

6.2.7 Reports. When an agent has finished executing his actions, he bundles the recorded information in an archive file and transfer that file to the experiment server (Fig. 3 step 7).

6.2.8 Cleanup. Finally, when the experiment manager closes the server, a last command is sent to the agents to uninstall themselves from the device for the cleanup phase (see step 8 of Fig. 3).

7 Conclusion

AVP (Android Virtual Playground) is a virtual test-bed for the Android platform. The main objective of AVP is to collect data during network experiments (on Android), while its main strength is the high level of automation it provides (from creating and launching Android AVDs, to forcing AVDs to perform specific tasks, while collecting data from several viewpoints).

Some solutions already exist for Android virtualization/experimentation. However, they are either extremely limited in terms of capabilities (e.g., actions to perform, data to collect) or they provide little (or no) automation mechanisms. One interesting feature of AVP (when compared to other existing solutions) is the ability to use multiple virtualization technologies. As a result AVP can harness the individual strengths of each technologies.

The main challenge while developing AVP was the lack of maturity of the existing virtualization solutions for Android, especially when compared with their PC counterparts. Hence, properly instrumenting the virtualization environments and stabilizing the execution of experiments proved to be more difficult than expected.

In order to circumvent some limitations of a virtual test-bed, we proposed a solution relying entirely on physical devices. This options provides an interesting level of flexibility to setup ad hoc experiments. However, since we have not been able to implement a fully automated snapshot-like functionality, our physical test-bed lacks a dependable cleanup mechanism.

8 Future Work

Our work can be extended in several ways. Below are some of the avenues we are currently working on, or expect to work on in the near future.

At the user-level, our next addition will be the ability to perform the same experiment (with slight variation) in batch on multiple computers (each running an instance of AVP) with a central controller (experiment server). For instance, replicate the same experiment but each time with a different application. To achieve this, we are working on template scenarios in which some elements are left unspecified and filled at run time (e.g., the application to use).

At the core of AVP lies the actions it can perform during an experiment. This is where most of the future work resides. Possible extensions are:

- Supporting new virtualization technologies (the first target is Microsoft Android Emulator part of Visual Studio 2015) and integrating existing data collection solutions (such as TaintDroid).
- Analyzing in more details the logcat output in order to keep a more accurate representation of an AVD's current state (e.g., detect the classical case where an app stops working).
- Providing more support for physical device. For instance, can we achieve something similar to snapshots of AVDs with respect to cleaning the device after an experiment.
- Enhancing the ability of AVP to record/replay user actions. We are working on a generalization where an action sequence recorded on one device could be replayed on another device (under the same, and eventually, a different virtualization technology).

From a networking point of view, we are interested in deploying a single experiment on several different physical hosts in such a way that all the AVDs remain connected with one another, see [4]. For instance, even if AVDs A and B actually run on different physical machine, they should be able to communicate through SMS.

Regarding scenario validation (see Sect. 5.1.1) significant work remains to be done, in particular to augment our semantic validation capabilities. Moreover, it would be interesting to provide a continuous runtime monitoring engine that would detect (and ideally try to recover from) problems occurring at runtime. An example is the failure to install (or start) an application in an Android device (possibly due to a malformed apk/manifest).

Acknowledgements

We would like to thank Thales Canada for their help and support through this project. This work is funded by the National Sciences and Engineering Research Council of Canada through grants RDA1-447989-13 and RDA2-452896-13. We are in the process of releasing AVP as open source, for more information, contact the corresponding author frgagnon@cegep-ste-foy.qc.ca.

References

1. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX 2005 Annual Technical Conference, FREENIX Track - Abstract. pp. 41–46 (2005)
2. Blasing, T., Batyuk, L., Schmidt, A.D.: An android application sandbox system for suspicious software detection. In: 5th International Conference on Malicious and Unwanted Software (MALWARE). pp. 55–62 (2010)
3. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2010)
4. Gagnon, F., Esfandiari, B., Dej, T.: Network in a box. In: Proceedings of the 2010 International Conference on Data Communication Networking (DCNET'10) (2010)
5. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
6. Kocialkowski, P.: Replicant developers find and close samsung galaxy backdoor. http://www.androidpolice.com/2014/03/13/security-researcher-dan-rosenberg-calls-bullshit-on-samsung-backdoor-vulnerability-published-by-fsf/ (March 2014 (Accessed 2015-01-15)
7. Lockheimer, H.: Android and security. http://googlemobile.blogspot.ca/2012/02/android-and-security.html (Feb 2012 (Accessed 2014-12-15))
8. Massicotte, F., Couture, M.: Blueprints of a lightweight automated experimentation system: a building block towards experimental cyber security. In: Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS '11). pp. 19–28 (2011)
9. Moth, D.: Microsoft visual studio's android emulator. http://blogs.msdn.com/b/visualstudioalm/archive/2014/11/12/introducing-visual-studio-s-emulator-for-android.aspx (Nov 2014 (Accessed 2014-12-15))
10. Percoco, N.J., Schulte, S.: Adventures in bouncerland - failures of automated malware detection within mobile application markets. BlackHat USA (2012)
11. Sandia Labs: Sandia builds self-contained, android-based network to study cyber disruptions and help secure hand-held devices. https://share.sandia.gov/news/resources/news_releases/sandia-builds-self-contained-android-based-network-to-study-cyber-disruptions-and-help-secure-hand-held-devices/#.VI9CYSvF-So (Oct 2012 (Accessed 2014-12-15))
12. Watson, J.: Virtualbox: Bits and bytes masquerading as machines. Linux J. 2008(166) (Feb 2008)
13. Yan, L.K., Yin, H.: Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: 21st USENIX conference on Security (Security'12). pp. 29–44 (2012)

Appendix A

Table 6. Example scenario file

```
<scenario>
    <devices>
        <device id="A" technology="taintdroid" name="testAVD1" />
    </devices>

    <actions>
        <StartAVD devices="A" snapshot="START-BUT-DONT-SAVE"/>
        <WaitForBoot devices="A" />
        <UnlockAVD devices="A"/>

        <StartRecordSMS devices="A" />
        <StartRecordCalls devices="A" />
        <StartRecordNetworkTraffic devices="A" />
        <StartRecordTaintDroid devices="A" />

        <InstallApp devices="A" appPath="X:/test.apk" />
        <WaitForInstall devices="A" appPath="X:/test.apk" />
        <StartApp devices="A" appPath="X:/test.apk" />
        <RunMonkey devices="A" nbEvents="5000" appPath="X:/test.apk" />
        <Wait time="5m" />

        <StopRecordTaintDroid devices="A" />
        <StopRecordSMS devices="A" />
        <StopRecordCalls devices="A" />
        <StopRecordNetworkTraffic devices="A" />
        <StopAVD devices="A" />
    </actions>
</scenario>
```

Appendix B

Table 7. Supported actions by technology

Actions		Virtualization Technologies				
Type	**Name**	**Go**	**MG**	**TD**	**Ge**	**Wo**
Device	*Call*	Yes	Yes	Yes	No	No
Device	*Navigate*	Yes	Yes	Yes	Yes	Yes
Device	*SendSMS*	Yes	Yes	Yes	No	No
Lab	*Push/PullFile*	Yes	Yes	Yes	Yes	Yes
Lab	*SendSMSToAVD*	Yes	Yes	Yes	No	No
Lab	*ShellCommand*	Yes	Yes	Yes	Yes	Yes
Lab	*StartApp*	Yes	Yes	Yes	Yes	Yes
Lab	*Start/StopAVD*	Yes	Yes	Yes	Yes	Yes
	snapshots	Yes	Yes	No	No	Yes*
Lab	*Start/StopRecordCalls*	Yes	Yes	Yes	No	Yes
Lab	*Start/StopRecordNetworkTraffic*	Yes	Yes	Yes	No	Yes
Lab	*StartRecordProcesses*	Yes	Yes	Yes	Yes	Yes
Lab	*Start/StopRecordSMS*	Yes	Yes	Yes	No	No
Lab	*Start/StopRecordTaintDroid*	No	No	Yes	No	No
Lab	*Start/StopStrace*	Yes	Yes	Yes	Yes	Yes
Lab	*StartMonkey*	Yes	Yes	Yes	Yes	Yes
Lab	*WaitForAVDBoot*	Yes	Yes	Yes	Yes	Yes
Lab	*CallAVD*	Yes	Yes	Yes	No	No
Lab	*InstallApp*	Yes	Yes	Yes	Yes	Yes
Lab	*WaitForAppInstall*	Yes	Yes	Yes	Yes	Yes
Lab	*UnlockAVD*	Yes	Yes	Yes	Yes	Yes
Lab	*Create/DeleteAVD*	Yes	Yes	Yes	No	No
Lab	*Start/StopRecordUserInteraction*	Yes	Yes	Yes	No	No
Lab	*ReplayUserInteraction*	Yes	Yes	Yes	No	No†

Go = Google emulator
MG = Modified Google emulator
TD = TaintDroid emulator
Ge = Genymotion emulator
Wo = VMWare Workstation

*Currently, only the latest snapshot is accessible through AVP.
†Actions are replayed, but precision is not yet adequate.