AVP
An Android Virtual Playground

François Gagnon[1], Frédéric Lafrance[1], Simon Frenette[1] and Simon Hallé[2]
[1]*Cégep Sainte-Foy, Québec, Canada*
[2]*Thales Research and Technology, Québec, Canada*
frgagnon@cegep-ste-foy.qc.ca

Keywords: Android, Virtualization, Test-bed, Network Experiment, Automated.

Abstract: This paper presents a virtual test-bed for the Android platform named AVP - Android Virtual Playground. The focus of AVP is the automatization of the manipulations required to perform a network experiment, in a very broad sense, involving Android virtual devices. A central objective of AVP is data collection during experiments. Together with describing the different steps of using AVP, from the specification of the experiment to the visualization of the results, the paper presents the current capabilities of AVP.

1 INTRODUCTION

In the last decade, mobile devices such as smartphones have penetrated the consumer world at an incredibly fast pace. 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. 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 a fully automated test-bed for the Android platform. Our test-bed aims to be generic enough to be used for many different purposes. Be it to test a mobile application on multiple hardware platforms for compatibility (both at the library level and the graphical level), or to inspect the security aspect of an application. The test-bed is large in scope to take advantage of the different modules already available instead of focussing on a particular aspect and digging deep. The 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).

We strongly believe automated test-beds for mobile platforms are an important component for speeding up the research related to this field. This has been the case with automated test-beds for malware analysis in the PC world (Massicotte and Couture, 2011).

The paper is structured as follows. First, section 2 places AVP in context with existing approaches for Android virtualization and experimentation. Section 3 gives a brief overview of AVP. The core of test-bed is described in Sections 4 and 5, respectively its capabilities and how it is used. Then, Section 6 concludes with final remarks and Section 7 discusses avenues for future work.

2 RELATED WORK

Our work can be related to two different fields: Android emulation solutions (see Section 2.1) and Android experimentation frameworks (see Section 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[1] emulator (Bellard, 2005). It can run native Android apps under the x86, ARM

[1]http://qemu.org

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 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.

- **Genymotion.** is a French company distributing a virtualization solution for Android. It relies on the well-known VirtualBox[2] hypervisor (Watson, 2008) and on the compatible images that can be built from the Android source code. Those images are 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.** [3] 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 virtulization options.

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

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

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.

- **TaintDroid.** (Enck et al., 2010) 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[4].
- **AASandbox.** (Blasing et al., 2010) is an Android malware sandbox implemented at kernel-level. It is entirely automated but focuses only on system call tracking.
- **DroidScope.** (Yan and Yin, 2012) 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 (Lockheimer, 2012; Percoco and Schulte, 2012). 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.**[5] is another proprietary solution for Android malware analysis. It offers a fully automated platform, but one which is focussing entirely on malware analysis.
- **Monkeyrunner.**[6] is Google's API to automate the control of their modified QEMU emulator. This project does not focus on malware analysis

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

[5]http://www.joesecurity.org

[6]http://developer.android.com/tools/help/monkeyrunner_concepts.html

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 be included in an experiment system like 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 preform 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 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).

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 on the host?)
- Semantic validation of the action sequence (e.g., can an action really be executed in the current experiment context?)

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 later 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 point, AVP instruments an Android virtualization technology in order to manipulate AVDs (e.g., start an AVD); it executes actions on 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 CAPABILITIES

This section deals with the core of our test-bed: the actions it can perform automatically (Section 4.2) and the data it can collect (Section 4.3). But first, the virtualization solutions supported in AVP are presented (Section 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 identical. For instance, Google's modified[7] 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* [8] (Formerly *AndroVM*/*BuildDroid*[9]), based on *VirtualBox*, provides a more fluid emulation targeting Android apps developers requiring only limited functionality

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

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

[9] http://androvm.org

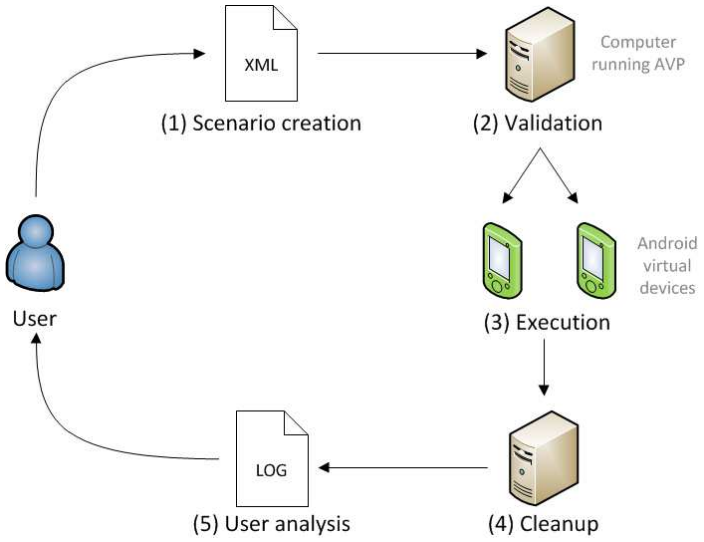

Figure 1: Android Virtual Playground Workflow.

(e.g., no support for SMS) but a more responsive 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 Section 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]
- A real physical device

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 two categories: lab actions and device actions. A lab action (see Section 4.2.1) is an action that is performed by our test-bed software (possibly on a set of AVDs), while a device action (see Section 4.2.2) is an action that is executed by an AVD (as instructed by AVP).

4.2.1 Lab Actions

4.2.1.1 AVD Manipulation. AVD manipulation actions are actions that aim to manipulate AVDs. Examples of such actions are: *create*/*deleteAVD*, *start*/*stopAVD*, *push*/*pullFile*, *unlockAVD*.

Most actions are self-explanatory, however, some have details of interest. Starting and stopping AVDs can 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. 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.

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 three action related to manipulating apps: *InstallApp*, *StartApp*, and *StartMonkey*. The *StartMonkey* action will run the *monkey*[10] application exerciser tool in the context of the specified app. This is very interesting to perform an experiment simulating a user interacting with an application. The data collection can then provide some behavioral information regarding the tested app.

[10]http://developer.android.com/tools/help/monkey.html

4.2.1.3 Lab Manipulation. Lab manipulations are actions that are handled by the test-bed. 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. Both the the above actions 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[11]. 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.

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.

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 AVD 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 simply by adding new capabilities to our *commandCatcher* library. So far, the following device actions are implemented:

- *Call*: the device will initiate a phone call to a specified number.
- *SendSMS*: the device will send the specified SMS message to the specified number.
- *Navigate*: the device will open a web browser to the specified URL.

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

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

4.3 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. Here is the list of information that AVP can collect at this moment and their corresponding supported actions (further details are provided below):

- SMS Activity: *Start/StopRecordSMS*
- Phone Call Activity: *Start/StopRecordCalls*
- Network Traffic: *Start/StopRecordNetworkTraffic*
- Running Processes: *StartRecordProcesses*
- System Calls Activity: *Start/StopStrace*
- Taint Propagation Activity: *Start/StopRecordTaintDroid*

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[12] can be obtained. this list is automatically refreshed each time there is a change in process activity. This provides us with the evolution of processes during an experiment.

System calls are monitored through the *strace*[13] utility. This provides the list of system calls for a much deeper analysis.

Finally, through the use of *TaintDroid* (Enck et al., 2010), AVP is able to track sensitive data leaking from an AVD. This, however, requires a specific AVD configuration (through the use of particular image files).

[12] process ID, owner ID, process Name, etc.

[13] http://sourceforge.net/projects/strace/

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*[14] 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 Section 7).

5 USING AVP

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

5.1 Scenario

A scenario is made of two sections: devices and actions. The first section specifies which Android devices will be used in the experiment. For each device, the user must provide an ID, the underlying technology and the device name. The ID 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 refers to the AVD filename on the host.

The second section 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, let's consider an experiment for the behavioral analysis of an application. Such an experiment can be specified with 3 stages of actions: setup, start collecting information, application execution.

In the setup phase, the lab will poweron (*StartAVD*) the necessary AVDs. We can suppose only one AVD "A" to host the appliction. In order to maintain this AVD clean, we could 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*).

[14]The Android logging system: http://developer.android.com/tools/help/logcat.html

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

Now the application is ready to come in play. The app 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 a specified number of random actions in the application context (*StartMonkey*). It would also be possible to interact with the App in a more precise manner (e.g., clicking on specific buttons in a specific sequence); however, this requires a very App-specific sequence to be specified.

Finally, it is possible, but not mandatory, to stops the data collection mechanisms and the AVD. Upon reaching the end of the scenario, the lab will 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 that 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 has its own set of VirtualBox-based 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. Similarly, starting and stopping physical devices is not supported.
- 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 file specifically related to the nature of the data, then log 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. 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 CONCLUSION

AVP (Android Virtual Playground) is a virtual testbed for the Android platform. The main objective of AVP is to collect data during network experiment (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 to 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.

7 FUTURE WORK

AVP 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, it would be interesting to provide a graphical user interface as an alternative to the XML scenario writing step.

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 and integrating existing data collection solutions (such as TaintDroid).
- Taking AVD screenshots during an experiment.
- 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 the integration of a physical device (e.g., can we achieve something similar to snapshots of AVDs with respect to cleaning the device after an experiment).

From a networking point of view, we are interested in deploying a single experiment on several different physical host in such a way that all the AVDs remain connected with one another, see (Gagnon et al., 2010). For instance, even if AVDs *A* and *B* actually run on different physical machine, they should be able to communicate through SMS.

Our experiment specification language will evolve (e.g., conditional execution and loops) with new utilization cases of AVP that require more expressive constructs. Moreover, to effectively support batch execution of similar experiment, we plan to generalize our scenario concept to introduce the notion of templates. For instance, a scenario template describing an experiment could be written without including the details on the application to use (for actions such as: *InstallApp*, *WaitForAppInstall*, *StartApp*, and *StartMonkey*). Then, the scenario template could be reused (and specialized) for each application in a given dataset.

Regarding scenario validation (see Section 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 inc. for their 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.

REFERENCES

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track - Abstract*, pages 41–46.

Blasing, T., Batyuk, L., and Schmidt, A.-D. (2010). An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). 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)*.

Gagnon, F., Esfandiari, B., and Dej, T. (2010). Network in a box. In *Proceedings of the 2010 International Conference on Data Communication Networking (DCNET'10)*.

Lockheimer, H. (2012). Android and security. http://googlemobile.blogspot.ca/2012/02/android-and-security.html.

Massicotte, F. and Couture, M. (2011). 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)*, pages 19–28.

Percoco, N. J. and Schulte, S. (2012). Adventures in bouncerland - failures of automated malware detection within mobile application markets. BlackHat USA.

Watson, J. (2008). Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166).

Yan, L. K. and Yin, H. (2012). Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *21st USENIX conference on Security (Security'12)*, pages 29–44.