Mobile Application Validation through Virtualization

Cyril Dumont[1] Steven Enten[2] Fabrice Mourlin[2] Laurent Nel[1]

(1) Research department
Leuville Objects
Versailles, France
email : {cyril.dumont, laurent.nel}@leuville.com

(2) LACL
UPEC University
Créteil, France
email : {steven.enten, fabrice.mourlin}@u-pec.fr

***Abstract*—When several business applications use low level libraries which are not installed in the firmware of the device; a solution consists in building of a new firmware with the well-chosen libraries. This afford to deploy only once for all the business mobile applications. The complete adaptation of an operating system to replace the Read-Only Memories (currently called ROM) manufacturer, fundamentally changes the kernel of an embedded system. This solution allows us to offer regular and frequent custom firmware updates that maintain business applications dedicated stability in time. By creation of firmware, we leave out the space consuming trial software. We may also leave out many of the included utilities, letting our users add them back only if they need them. Often we also strip out carrier specific versions of the launcher, replacing them with Google's original versions or a version we prefer. After customizing a firmware, we focus on adding business software in place to monitor the embedded device. Thru, we use this firmware to virtualize an embedded device. Thus, we collect information to determine whether the firmware can be deployed on devices. The collected data are about memory usages, threads, and resource access and energy consumption. So, this reporting step sums up the validation of our firmware, then they are validated to a deployment step on mobile devices. Reports are delivered about the behaviors of embedded software.**

Keywords-embedded device; firmware custom; monitoring; virtualization; state management.

I. Introduction

To cook a ROM (Read Only Memory) is the process of modifying a firmware of an embedded device. It can be seen as a kind of bridge between the applications and the actual hardware of a device. When business applications need low level libraries, the firmware has to be customized by the company. For companies which need specifically designed terminals to one or more trades, rhis new solution is called the Read-Only Memories (currently called ROM) cooking. Such approach is also useful when business applications have to be added into a new kind of embedded device. Starting from a base of operating system (such as Android) installed on a smartphone or tablet, firmware, or operating system is modified to fully meet demands without unnecessary applications. Another motivation of firmware update occurred when low level libraries have to be changed. There are plenty of examples in companies: for instance a TV application needs a specific library for video streaming; a network application which encrypts its messages needs also the use of a specific algorithm which is not necessary in a standard distribution of the framework.

The firmware cooking, namely the complete adaptation of an operating system to replace the firmware manufacturer, fundamentally changes the kernel, the Android framework and pre-installed applications for a completely clean system to the company. This solution has the merit of offering and customized the firmware of regular updates and more frequent than the manufacturer's updates. This solution provides better stability over time of business applications. Android firmware is particularly suitable for customization for several reasons. Embedded device manufacturers have a process to build their own firmware. It is time consuming and the update management is a difficulty regarding the set of potential devices.

As an open system, developers have the source code required to modify the Linux kernel. They rely for the rest on the binary components manufacturer if their source codes have not been distributed. These codes are commonly referred to as hardware abstraction modules, for example for the camera, Global Positioning System (GPS), sound and graphics acceleration. Android is supported by a broad spectrum of terminals, allowing a company to choose the device ergonomics best suited to business constraints.

Finally, users are not disturbed because the Android environment is familiar. A consumer product can perfectly meet a Business to Business (B2B) demand. Experience has shown that the porting of applications is now done without difficulty, regardless of the changes made by Google's Android versions, even a major evolution [1].

Some companies that have experience in customizing firmware chose to integrate their approach, the staff concerned. The selected spectrum of users participating in the experiment, allowing finely define requirements and content and gradually reflect on future developments as a dedicated applications market. These companies are also finding true motivation of their staff on these issues. Today, in terms of support, initiatives are numerous: note of Android Business Group, the sharing of experiences between large accounts [2], the Data Android User Group [3], a monthly meeting of developers, backed by Google and able to meet many contributors. This contributes to the development of firmware cooking into several companies.

The firmware cooking allows access to personal data essential to their work wherever the end users are and the company with materials and fully dedicated to their activities such as information environments, training, home automation, transportation, geolocation, security, etc.). When new software is installed, the consequences on these data are essential. Also, what are the disruptions caused by new software on the host platform. In order to do this study, it is important to be able to place a virtual machine under observation. The goal is to validate by suitable tests that use this virtual machine are safe. These observations are crucial because the next step is the deployment of virtual machine on mobile devices. And any error becomes serious consequences for the users and the publishers [4].

We have structured our paper to present our approach to firmware customization and also put in our firmware monitoring controllers. So, the section 2 deals with the monitoring of mobile applications, the nature of the data collected but also how to make the collections to minimize disruption to ongoing observations. In section 3, we discuss a case of firmware manufacturing dedicated to the study of a business application. We add measurement points dedicated to the type of mobile platform. Then, we perform the data collection and build the associated representations. In section 4, we explain the usefulness of such software architecture for gathering information. Data analysis is also detailed. Finally, we conclude on the implementation of our approach for customizing firmware before deployment.

II. Mobile application monitoring

Because new applications can involve conflicts between the previously installed applications, it is essential to observe the behavior of the mobile applications on a given device. This is particularly crucial with applications which need root permission or acquire some sensors such as a camera, etc. Mobile monitoring is become a key step in the lifecycle of a new mobile application. This step consists of looking at the behavior of an application on an embedded platform.

A. Embedded system monitoring

1) Basic mobile application monitoring:

Some fraudulent mobile applications are malware, which may capture personal information sent and received by the device or make phantom calls to premium phone numbers, while others may just be using a company name or logo without prior authorization. Regardless of their intent, these applications create a negative association in the mind of the user, which tarnishes the company’s good reputation. This type of bad behavior can be detected by a sufficiently long period of observation commissioning of future embedded platforms.

When a fraudulent application is detected, it has to be immediately reported to a log along with a full report [5]. This contains developer information, number of downloads, application screenshots, and a diagnostic as to why this mobile application is believed it to be fraudulent. It provides valuable intelligence data and can help support a criminal investigation.

Today, manufacturer services give security operations users an additional layer of protection, coupled with a new data stream that includes more contextual information about the specific and potential threats contained in fraudulent mobile applications. This approach can be completed by ad hoc tools, which collect data about runtime of applications under observation. We have decided to build a tool chain for building these data collections.

2) Adhoc monitoring.

Through the application monitoring feature a mobile application can be studied in depth if the monitoring task is developed in close relationship with the features of the given mobile application. For instance, when a mobile application uses Bluetooth protocol, then a monitoring task has to be configured to control the packets, which are transferred on this protocol, the collisions which occur, the availability of the sensor, etc.

Tools such as Systrace tool, helps us to analyze the performance of mobile applications by capturing and displaying execution times of these applications processes and other Android system processes [6]. This kind of tools combines data from the Android kernel, such as the Central Processing Unit (CPU) scheduler, disk activity, and application threads to generate an Hypertext Markup Language (HTML) report that shows an overall picture of an Android device’s system processes for a given period of time. Very often, such kind of tools is considered as debugging tools because they are particularly useful in diagnosing display problems where an application is slow to draw or stutters while displaying motion or animation. But a main drawback is the obligatory use of a USB debugging connection.

We needed a way to get periodic screenshots of a mobile device connected to a computer through a light protocol. On Android Platform Dalvik Debug Monitor Server (DDMS) has the ability to take screenshots on-demand, but not automatically. It provides port forwarding services, screen capture on the device, thread and heap information on the device, etc. but the documentation is so poor that source code of the library is the only information source. It uses an Android Debug Bridge called `adb`. It allows us to communicate with a connected device on the same WIFI network.

On Android, every application run in their own process, each of which runs in its own Virtual Machine (VM). Each VM exposes a unique port that a debugger can attach to. We have built a DDMS monitoring application for looking at the embedded runtime of business applications. When we start our DDMS application, it connects to `adb`. When a device is connected, a VM monitoring service is created between `adb` and our DDMS monitoring application, which notifies our DDMS application when a VM on the device is started or terminated. Once a VM is running, our DDMS application retrieves the VM's process ID (pid), via `adb`, and opens a connection to the VM, through the `adb` daemon (`adbd`) on the device. Our DDMS application can then talk to the VM using a custom wire protocol. The result is a data collection about the behavior of the embedded business application.

This strategy can be translated within a hypervisor like VirtualBox or VMWare. The `adb` daemon is called through a virtual network mapping between the host platform and the

Android virtual machine. The main advantage of this approach is to run the monitor on the host platform and the mobile application (under observation) on a virtual machine managed by a hypervisor. A second benefit is on the porting of application. The hardware architecture constraints are respected; only the configuration of our monitoring application is updated and its network mapping.

B. *Memory management*

Our DDMS application allows us to view how much heap memory a process is using. This information is useful in tracking heap usage at a certain point of time during the execution of business applications. Another feature of our DDMS application is to track objects that are being allocated to memory and to see which classes and threads are allocating the objects. This allows us to track, in real time, where objects are being allocated when we perform several actions in our application. This information is valuable for assessing memory usage that can affect application performance. This happens when an application shares preferences with another one.

The file system of the virtual machine is also an information source. It is useful in looking at files that are created by a mobile application or if we want to transfer files to and from the virtual machine. This is also useful when the size of the data collection is so large that it is suitable to filter a part of the data before the transfer. This case occurs when the mobile application uses the sensors such as the camera or the microphone. The output format and the recording involve often a large output file. Only a part of the data is useful for the analysis. Also, we filter locally to the device a subset of persistent data by the end of the monitoring scenario.

C. *Time profiling*

Method profiling is a means to track certain metrics about a method in a program, such as number of calls, execution time, and time spent executing the method. When we want more granular control over where profiling data is collected, it is possible to deep into the body of a method and to compute other measures.

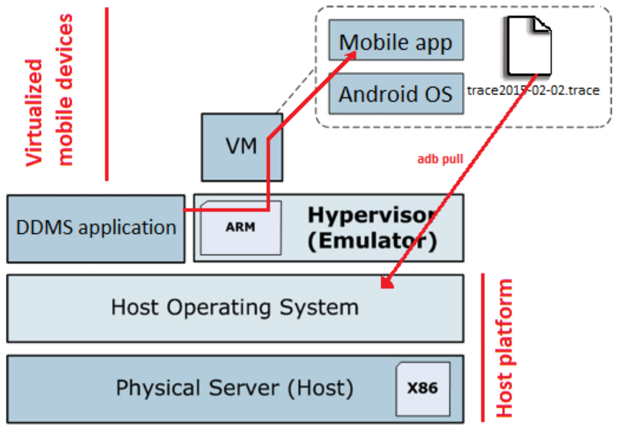

Figure 1. Monitoring architecture

A difficulty of embedded operating systems such as Android lies in its organizational changes between different versions of the same operating system. So, depending on the Android version, our DDMS application provides a summary of what happened inside a given method. Also, we need to generate log files containing the trace information we want to analyze. We use the `Debug` class in our code and call its methods such as `startMethodTracing()` and `stopMethodTracing()`, to start and stop logging of trace information to disk. This option is very precise because we can specify exactly where to start and stop logging trace data in our DDMS application. Our monitoring application has necessary the permission to write to external storage.

To create the trace files, we include the `Debug` class and we call one of the `startMethodTracing()` methods. In the call, we specify a base name for the trace files that the system generates. These methods start and stop method tracing across the entire virtual machine. For example, we could call `startMethodTracing()` in our activity's `onCreate()` method, and by the end of the monitoring stage, we call `stopMethodTracing()` in that activity's `onDestroy()` method. When our application calls `startMethodTracing()`, the system creates a file called "`trace2015-02-02`" trace. This contains the full method trace data and a mapping table with thread and method names (see figure 1).

The system then begins buffering the generated trace data, until our application calls `stopMethodTracing()`, at which time it writes the buffered data to the output file. If the system reaches the maximum buffer size before we call `stopMethodTracing()`, the system stops tracing and sends a notification to the console. This event can also trigger the pulling of the technical data from the mobile device to the workstation. We have also used the data exportation through the use of RESTful remote monitoring application.

D. *Profiling scenario*

After a mobile application has run and our DDMS application has created the trace files "`traceyyyy-MM-dd`".trace on the device, we have to copy those files to the host computer. We use `adb pull` to copy the files. As an example, we copy a trace file, `trace2015-02-02`, from the default location on the device to the `/tmp` directory on the host machine via:

```
   adb pull /sdcard/trace2015-02-02 /tmp
```

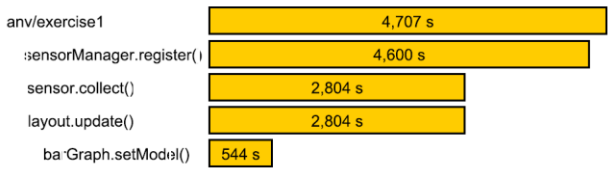

Figure 2. top 5 of costly methods

The format of the trace file is suitable to be parsed by `TraceView` or `TraceDump`. This tool uses the `Graphviz Dot` utility to create the graphical output, so we need to install `Graphviz` before running this command. But as we are going to explain in the fourth section, the data can also be sent to a monitoring server where services parse the collected data and compute metrics about the execution [7]. As an example, the Figure 2 shows the most costly methods into a bar graph representation.

III. CUSTOM STRATEGY FOR FIRMWARE COOKING

Firmware is the low level programming. They are also often called firmware; they contain the operating system and basic applications to make the phone work. For the iPhone and iPad those firmware come from Apple and can typically only be updated when Apple issues updates. But for Android devices there are literally hundreds of developers working on custom firmware for most common models of phones and tablets, which they are proud to share with the community. In our working context, we need to define firmware with additional software. First, we need to add our DDMS application for the future observations. This component is important for local monitoring. Secondly, we want to install business software which is under control during this validation step.

A. From source to Virtual Machine

For building a new firmware, several choices have to be done. So, it is essential to know the advantages or disadvantages of using an Android Open Source Project (AOSP) firmware versus a ROM stock. A ROM stock is the firmware that comes with a device; the device is stocked with that firmware by the manufacturer. Android is generally customized by the manufacturer to some degree; at minimum there needs to be device specific drivers for Android to work on a particular device. The customizations may include a custom theme, launcher, and default applications like monitoring control panel does.

1) A large set of acronyms

An AOSP firmware is a ROM based on the Android Open Source Project. In the purest sense, AOSP refers to unmodified ROMs or code from Google. The name is often co-opted for a custom firmware that is very close to the original AOSP, since these firmware still need to be customized; for example, we have downloaded and compiled the Android source code and run it on a Samsung Galaxy S5 with doing a whole lot of customizations. For example, monitoring libraries are installed with test suites. This means that we have added source projects with configuration files for building, testing.

Technically, ROM stocks are all AOSP firmware apart from the versions of Android that has not been released yet. Kitkat and Lollipop firmware are AOSP for a long time; the source code is available at Android Web site. In the next case studies, we will use Kitkat and Lollipop versions.

To further add to the confusion, a custom firmware does not refer to customized firmware in general. That term specifically refers to firmware that has been customized by engineers or researchers which are not the manufacturers or carriers. For example, CyanogenMod provides firmware [8] which is modified under the constraint of the open source community. Most AOSP firmware for a specific device is ROM stocks that have been customized to remove some of the manufacturer or carrier features and make them closer to the pure AOSP experience. As an example, we can disable PIE feature on any ROM stock or AOSP firmware. The option is not even available in most ROM stocks. So our solution is to modify the AOSP firmware source and then build them into an updated firmware.

2) The benefits of firmware cooking:

First of all, the main thing to know is that messing with the firmware of phone can be risky. We can potentially damage a mobile phone so that it won't be usable without some major low-level hacking. This reason involves our need to experiment new customized firmware behind a hypervisor.

The most basic benefit of custom firmware is getting rid of unstable software of malware, spy application and so on. These applications take up precious room on a mobile device. Beyond simple fixes, custom firmware can also open a whole world of new possibilities for new mobile devices. In many cases newer versions of Android are available for the devices as custom firmware, beyond what the carrier has released or is planning to release. Custom firmware can also include other pleasant features, like overclocking, themes, private browsing support, and so on.

Our current objective was to isolate the minimum Android operating system which can support our business mobile applications and our monitoring tools. The architecture of Android platform is defined to host new software. The source codes provided by AOSP are also organized to host new source projects, which have a predefined structure. Android has layer architecture (Figure 3). This structure is understandable easily when we explore the source code. For instance, we wanted to record, convert and stream audio and video. We observed that, the OpenGL library can be upgraded or completed by the add-on of `ffmpeg` library. Also, we have added the source code project with Android build file into the whole Android source files. When a new build is launched, then this new library is taken into account.

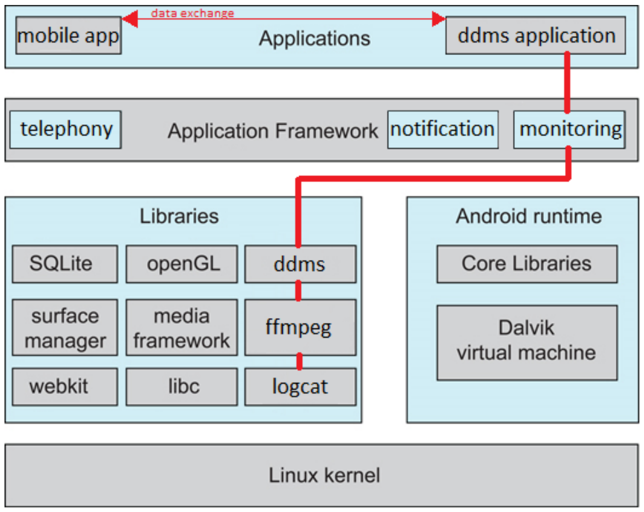

Figure 3. Android layer architecture

For our project, the main benefit was the enrichment of the Android source distribution with the source codes of all the boxes connected with the thick red line (Figure 3). The box called “mobile app” plays the role of the test suites or the mobile application we want to observe before validation.

Next, a benchmark step can start. Its duration depends on the features, we want to observe. Software stability is a key feature in the study of our business applications. Also, the meantime declared by the manufacturers are used as test limits

3) Our firmware cooking approach:

A firmware construction is a step by step process. First, we define the software basement. Then, we prepare the build of a new firmware, with the selection of the right add-ons. Controls are done about the version numbers, compatibility features, architecture definition and security permissions. We have developed an embedded monitoring application for our embedded devices that generates log files with technical data about the runtime. In the future, new tracking tools will be added about energy management, bandwidth network use. So, if we imagine those applications always having to write the machine code to get the monitoring to turn on. It would be a lot of code duplication and would make an application slow. Instead, for functions like the method profiler or thread manager, we have packaged our own monitoring library. These are pieces of codes that can be executed by calling them through a method call. These are already pre-written and ready to use. It saves a lot of coding work and keeps the source code small. On the Android device, we have particular libraries like `ddms` or `ffmpeg` libraries that can’t be absent, or else the firmware won’t even be used in our studies.

On Figure 3, the red line highlights the dependencies during a monitoring scenario. A mobile application is under observation. Also, by the use of our monitoring application (called `ddms` application), we collect behavioral data. They are computed through the use of `ddms` library and `logcat` library which already belongs to AOSP distribution. Depending on the application domain of the mobile application, the `ffmpeg` library plays a role of stream observer. For instance, it contains `ffprobe` tool which gathers information from multimedia streams and prints it in human- and machine-readable fashion. For example, we use it to check the format of the container used by a multimedia stream and the format and type of each media stream contained in it. We use `ffprobe` in combination with a textual filter, which performs more sophisticated processing, e.g., statistical processing and plotting. `ffprobe` output is easily parsable by a textual filter, and consists of one or more sections of a form defined by the selected writer, which is specified by the print format option. The sections contain other nested sections, and are identified by a unique name. The metadata tags stored in the container or in the streams are recognized and printed in the corresponding output section. This means that we control the properties of the video streams. Such abilities are essential in the case of audio communication application, such as encoder/decoder audio applications. Depending on the data size, the output files are saved into a specific folder. Each of them respects a block size limit.

B. Virtual Machines managed by hypervisor

The virtualization is a technology that allows resources to be shared by a variety of physical outlets. Today, virtualization is a topic, which is focused on the relatively new concept of server virtualization. In this context, multiple operating system and application sets are virtualized on a single server, allowing it to be more efficiently and cost effectively used. But, there are a multitude of virtualization schemes addressing a spectrum of applications. We have addressed new ideas around virtualization and applied their uses and advantages to the virtualization of mobile devices.

1) Virtualization of platform.

The virtualization is an abstraction over physical resources to make them shareable by a number of physical users. Platform virtualization is what enables both server and desktop virtualization. A platform in this context refers to the hardware platform and its various components. This includes not only the CPU, but also networking, storage and bus attachments such as USB and serial ports, but also sensor such as camera, microphone and even GPS or compass.

The key technology that makes this possible is called the hypervisor. The hypervisor is the component that virtualizes the platform, making the underlying physical resources shareable and implementing the policies for sharing among the multiple virtual machines. These ones can belong to embedded systems like Android or IOS. They are an aggregation of the operating system and application set which contains our mobile applications. The VMs are considered as a file in some format depending of the hypervisor. The virtual disk used by the VM is another file encapsulated within the VM.

An Android VM as a file in a host system like a hypervisor has some interesting benefits: we can back up the full virtual machine and its configuration in one hit rather than backing up at file level within the server. As a file, it's easy to manage a VM as a template. It's also simple to move a VM from one host system to another, as the process is nothing more than a file copy. As we expected with the virtualization concept, there are a variety of ways in which virtualization can be achieved. For platform virtualization, there are two primary models, called full virtualization and para-virtualization. Both are suitable depending on the kind of architecture of the mobile device ARM or x86. The last case corresponds to full virtualization; the former one exploits an intermediate layer.

2) Managed VM through a hypervisor .

The hardware of mobile devices has multiple features; this concerns not only the processor but also the sensors and even the pluggable add-ons. The usage of VM involves several approaches depending on the architecture of the physical server. In our project, we use AMD architecture servers (64 bits). This means that we apply the full virtualization when we test and validate software for AMD mobile devices. This kind of virtualization provides a sufficient emulation of the underlying platform that a guest operating system and application set can run unmodified and unaware that their platform is being virtualized. But, the most widespread kind of processor is ARM, a solution is to make the guest operating system aware that it's being virtualized. With this knowledge, the guest operating system can short circuit its drivers to minimize the overhead of communicating with physical devices.

We use Qemu, which emulates a full system, including a processor and various peripherals [9]. A number of specific emulator features are enabled in both the Android kernel and Android user space environment when run in an emulated environment. These features allow a smooth and complete user experience resembling using a real Android device, on laptop and desktop workstations. With the introduction of the ARMv8-A architecture and Android support for 64-bit ARM platforms, this need is more important than ever because it allows us to begin adapting our applications to an ARM 64-bit based mobile ecosystem prior to hardware being available. All of our tests are based on the use of Qemu and the exposition of our own custom firmware via a graphical interface.

C. *Testbed of mobile business applications*

Testing functionality is typically a matter of enumerating the functions that an application should support, then defining a set of tests that exercise those functions, with pass/fail results. Problems that we encounter running functional tests are input to evaluating usability. For usability, we want to have several mobile end users with different skill levels attempting to accomplish a given set of business goals, producing subjective ratings that indicate how easy or hard the task was. Performance test results are easier to quantify, but can be very difficult to interpret. For example, wireless throughput is always higher in the lab under ideal conditions than in real life, so be very careful about the conclusions we draw from performance tests.

To test failure modes in components, we enumerate a number of possible failure conditions and simulate them. We must also identify what we are measuring. For example, when measuring time to establish the connection in the event of network loss of signal, do us measure network connection resumption or mobile application connection resumption.

IV. CUSTOM APPROACH OF MOBILE APPLICATION MONITORING

Based on the build of our own kernel and the enrichment of the AOSP sourced, we have built our own custom firmware. Our results are presented as log reports and numerical measures.

A. *Monitoring architecture for mobile applications*

When a mobile business application is running under monitoring, all events are saved through the use of a local monitoring application (as explained previously). Huge amount of data, even for relatively small programs are recorded in the local file system. Then, these data are exported to a server. The main events are class load, or unload, compiled method load, and unload, GC start, finish, method entry and exit, thread start and end, etc. We assign IDs to objects, classes, methods, etc. And our monitoring application is responsible for keeping track of IDs. They are assigned through defining events (e.g., class load). As a small part of an example of output trace, the following sequence of event trace in table I.

TABLE I. EVENT TRACE OF METHOD CALLINTENT

```
public int callIntent(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

Other information about performance is also collected. They are about the method execution measure and also class loading and checking. As an example, the table II shows a first level of information about time measures. The size of these data depends on the number of samples per time unit.

TABLE II. DATA TIME COLLECTION

```
Capture.callIntent 2015-03-08 14:59:30.252,
Capture.update 2015-03-08 14:59:30.254,
Model.get 2015-03-08 14:59:30.255,
Capture.setProps 2015-03-08 14:59:30.258 …
```

All the timestamps allow tester to display the events of the garbage collector at runtime.

B. *Interaction between monitor and mobile application*

Our message exchange protocol is packet based and is not stateful. There are two basic packet types: command packets and reply packets. Command packets may be sent by either the ddms application or the target VM. They are used by the ddms application to request information from the target VM, or to control program execution. Command packets are sent by the target VM to notify the ddms application of some event in the target VM such as a breakpoint or exception. A reply packet is sent only in response to a command packet and always provides information success or failure of the command. Reply packets may also carry data requested in the command (for example, the value of a field or property). Currently, events sent from the target VM do not require a response packet from the ddms application.

Our monitoring protocol is asynchronous; multiple command packets may be sent before the first reply packet is received. The layout of each packet looks like in table III:

TABLE III. COMMAND PACKET LAYOUT

```
Header
   length (4 bytes)
   id (4 bytes)
   flags (1 byte)
   command set (1 byte)
   command (1 byte)
```

```
data (Variable)
```

All fields and data sent via our monitoring protocol should be in big-endian format. It means the big-end is first. In other words, we store the most significant byte in the smallest address

TABLE IV. REPLY PACKET LAYOUT

```
Header
   length (4 bytes)
   id (4 bytes)
   flags (1 byte)
   error code (2 bytes)
data (Variable)
```

The length field is the size, in bytes, of the entire packet, including the length field in table IV. The id field is used to uniquely identify each packet command/reply pair. Flags are used to alter how any command is queued and processed and to tag command packets that originate from the target VM. The command set is useful as a means for grouping commands in a meaningful way. The error code field is used to indicate if the command packet that is being replied too was successfully processed.

This command field identifies a particular command in a command set. This field, together with the command set field, is used to indicate how the command packet should be processed. The data field of a command or reply packet is an abstraction of a group of multiple fields that define the command or reply data. As an example of data type: threadID uniquely identifies an object in the target VM that is known to be a thread. Another example is methodID, which must uniquely identify the method within its class/interface or any of its subclasses. A methodID is not necessarily unique on its own; it is always paired with a referenceTypeID to uniquely identify one method. The referenceTypeID can identify either the declaring type of the method or a subtype.

C. Validation process of mobile application

The list of all the data types is not exhaustive here but all element of a runtime program can be referenced and tracked. So, based on these results, we are able to decide whether the source codes of mobile applications can be added into the source.

The validation process takes into account response time of applications and our monitoring protocol helps us to collect internal data from each application under test. For instance, when response time is greater than 20% of the expected response time, we can conclude that there are perturbations from the runtime context towards the application sunder test.

Another test case is about the management of the memory by the virtual machine which runs a business application. We can compare the used memory with the first benchmarks of the applications under test. When the difference exceeds 25% then if means that the application cannot be deployed on the future firmware.

If we do not respect such rules, we could build unstable firmware and the consequences will be more serious. For instance, after flashing the firmware a phone works fine for one or two weeks. But soon this phone starts crashing more often. The phone starts rebooting every now and then it becomes useless. The most difficult point is the loss of working time. Also, by applying the reference measurement definition tested is crucial for our validation process

The validation by the use of virtualization has the advantage of using virtual devices instead of concrete smart device. The flash of firmware is a dangerous operation for the hardware and we preserve the hardware by previously testing our custom firmware. Another approach need a deployment on a smart device with a rooted firmware.

In the opposite side, the build of firmware involves new drawbacks after the deployment step. A first one is a legal issue. This means that the manufacturer guarantee is cancelled when a free firmware is installed. A second one is about the telecom provider checks. When several tools of a given telecom provider are already installed, then exceptions are raised by these applications when the underlying firmware is changed. Also, it is often useful to change a whole toolkit of software when new firmware is built by ourselves.

V. CONCLUSION

As we explained in the first section, we need to prepare our own firmware because of the change of libraries which are essential for our business applications. Also, the choice of validation before deployment explains our use of virtual machine. In this paper, we have presented our approach of the monitoring of mobile business applications. It is based on a perfect configuration of all the ROM stock and its build. We have shown that it is preferable to have a local monitoring instead of a remote monitoring application. The impact of its actions is less in the case of embedded systems.

We have described briefly our stateless protocol between the VM of the business application and our monitoring embedded application. We want to enrich this protocol and then observe new kinds of property.

The data collected are exported to a server where they can be parsed and aggregated with other simulations. Next new reports can be built and published onto a Web server if the monitoring data are public. We think that our approach is an adaptation of a monitoring strategy from Web domain into the domain of mobile applications. We consider our pragmatic study as a validation of our concepts and our next step will be to automatize as much as possible all the steps described in the document. And so, our experience could be transferred to other development teams.

Our approach reduces the effort of deployment on a large set of devices. Moreover, we reduce also the number of anomalies by increasing the observation time when some expected benchmarks are not achieved

REFERENCES

[1] “The Android Source Code: Governance Philosophy”, source.android.com, January 25, 2015.

[2] M. Isacc, “A deep-dive tour of Ice Cream Sandwich with Android's chief engineer”, Ars Technica, September 15, 2012.

[3] A. Shah, “Google's Android 4.0 ported to x86 processors”, Computerworld, International Data Group, February 20, 2012.

[4] R. Whitwam, “HTC Posts Android 4.4 Kernel Source And Framework Files For One Google Play Edition, OTA Update Can't Be Far Off”, androidpolice.com, December 2, 2013.

[5] “Exclusive: Inside Android 4.2's powerful new security system | Computerworld Blogs”, Blogs.computerworld.com, November 9, 2012.

[6] “AppAnalysis.org: Real Time Privacy Monitoring on Smartphones”, February 21, 2012.

[7] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo, “A technique for drawing directed graphs”, IEEE-TSE, March 1993.

[8] E. Tyler and W. Verduzco, “XDA Developers: Android Hacker's Toolkit: The Complete Guide to Rooting, ROMs and Theming”, Wiley edition, May 2012.

[9] R. Warnke and T. Ritzau, “Qemu”, Paperback, March 10, 2009.