DroidSand - A modular dynamic analysis platform for Android

Tobias Kirsch
Saarland University
Saarbrücken, Germany
s8tskirs@stud.uni-saarland.de

Jannik Pfeifer
Saarland University
Saarbrücken, Germany
s8japfei@stud.uni-saarland.de

***Abstract*—DroidSand is a dynamic analysis platform for Android which consists of several different analysis modules that serve different purposes. Its' capabilities are not only pure analysis techniques like internal method tracing, network traffic inspection and syscall tracing but also the ability to trigger as much functionality of the app under analysis as possible by intelligent GUI exploration combined with static analysis to examine the possible entry points. Finally, DroidSand uses an approach called event stimulation to uncover possible functionalities which are triggered by special system events.**
DroidSand is originally designed to be integrated into the Sandnet malware analysis platform [RDB+11] but it also can be used as a standalone environment for app analysis. Additionally, DroidSand is completely modular such that each of the functionalities can be activated and deactivated at runtime or new functionality can be integrated quickly. This documentation describes the complete structure, functionality and usage of the modules and gives a detailed overview on the requirements and building process such that the reader is able to rebuild the whole system from scratch. We also describe the problems we faced while setting up the system and the results of evaluating and testing the final product. Furthermore we give an overview on additional features that we want to integrate in the future.

I. Environment

While the later sections focus on the development of the single modules we start the documentation by giving a description on how we came to our final environment. Of course we ran into several problems while testing the single modules which led to changes in the whole environment, so there are some overlaps between the module and the environment development but we tried to separate the two topics as far as possible. In the following we give a detailed description of our timeline during environment development and describe the final system backend as a separated unit. The second part of this section then gives insights in our control scripts and a deeper understanding of the overall analysis run.

A. Development & Structure

When starting with designing a dynamic analysis platform for Android of course the first thing one does is to build the AOSP. Unfortunately this works not really well on the operating system we used (Ubuntu 16.04 LTS). Readers who want to completely comprehend our steps should just simply try to build Android on Ubuntu 16 and will notice that despite fullfilling the build requirements as stated by Google [AND] one has to apply a patch for *platform/build/core/clang/HOST_ x86_ common.mk* which we wanted to mention here for completeness reasons. [BP].
With a completely built AOSP and the corresponding emulator we were able to start testing the ARTist tool which is adapted to one of our core modules but directly faced the first problem: ARTist needs root privileges to execute a few shell commands (more information on that in section Internal Method Tracing). We had two options, either we could try to modify the ARTist code such that the needed shell commands are executed by our own scripts or we could install the superuser app on our emulator. Since superuser access seems to be dangerous when executing malware we tried the first strategy. The results are described in detail in the ARTist section, at this point it is enough to mention that this strategy turned out to be successless. So we decided to continue by using the superuser app but for security reasons we created a different plan. Due to the fact that superuser privileges are only needed during the ARTist compilation that creates *oat* files which then are placed in the directory of the app we did set up two emulators, one with superuser installed to recompile the app and another one on which the final analysis executions take place and on which we replace the *oat* files manually after compilation finished on the first one. Unfortunately pushing a new superuser binary is not completely easy, since the emulator has a read-only filesystem, but some internet searching was enough to find solutions. The bigger problem we faced was that giving artist superuser access with the superuser app via a script would have been really difficult, since the superuser brings up several pop ups that have to be confirmed by the user in a non-deterministic way, which led us to another problem that existed since the beginning of our project.
One of the most important requirements of an analysis platform is the possibility to reset the system into a clean state after executing malware suspects. In theory, the Android emulator which is completely based on *QEMU* [Q], provides the perfect solution for this problem: snapshots. Snapshots also would have been the perfect solution for our superuser problem, we simply could have loaded a state in which superuser had already been granted to ARTist. But as powerful as the QEMU eumlator is, as buggy is Google's adaption of it for Android. When we started with setting up our system we faced a lot of problems caused by using snapshots which delayed our timeline and previous plans a lot and finally led us to a completely new design of our platform environment.
Since we followed a simple try and error strategy when setting up our system we can describe our perspective when facing the problem for the first time in a simple kind of way: The emulator hung up and rebooted whenever we tried to load a

snapshot. Digging deeper into the logs we found out that the kernel triggers a kernel panic when loading a snapshot. The detailed kernel log can be found in appendix B: Kernel log when loading snapshots. As it seemed that the problem was caused by the kernel we decided to manually build our own kernel from sources, which was not that hard but quite time expensive since the Goole docs for building a kernel do not give detailed descriptions for all kernel versions. Anyway we managed to build a working goldfish kernel but that did not solve our problem.

Meanwhile we faced another problem that we completely missed before but turned out to be enormously huge during our module tests: The ARM emulator is really slow. Recompiling bigger apps with ARTist took at least 30 minutes on our test machine and sometimes the recompilation even ran into a timeout which clearly showed us the need of using the x86 emulator, which unfortunately also has problems with the snapshots. But now that we managed out how to build kernels, why not trying to use the x86 kernel? Although successfully building the x86 kernel, we faced the new problem that the kernel version that we used somehow triggered an infinite loop at startup. It seems that this bug is restricted to only some specific versions of the kernel [BUGb] but by switching to other versions we were not able to solve this issue and we finally stopped digging deeper into the problem since we wanted to follow another approach based on our interpretation of the kernel log.

The fact that the kernel did panic when loading a snapshot obviously pointed to a low-level problem so we focussed on changes when loading a snapshot on the kernel layer. Clearly the hardware configuration could not have changed, but maybe the kernel trigered a panic because of changes in the memory, so we focussed on making the memory at the point of creating the snapshot persistant. The most obvious naive solution was to copy all memory images when creating the snapshot (which by the way is not as easy as it seems, due to some locking issues) but as many naive approaches it did not lead to any success, so we focussed on the copy-on-write images, QEMU is able to use. Although copy-on-write approaches are a well studied and widespread feature, especially when facing concurrency issues, their configuration in QEMU is not completely intuitive and easily achieveable since the *qcow*[1] format used by QEMU exists in several versions with different compatibility settings for each of them. Finally we managed to find the right compatibility and size settings for the images used by the Android emulator. But again, the Android emulator did not use the whole power provided by the QEMU backend since it seemed to us that the emulator simply did not care about the kind of image we gave him. No matter which images we replaced, the timestamps of them revealed that the emulator used them like standard images.

Finally we stopped all of our investigations since we found a new fact that completely changed our view on the snapshot error. Of course both of us have the standard Android developer tools (AndroidStudio) installed and some internet search brought us to many forum entries where snapshots are described from the developers' view. Basically it should be possible to enable or disable snapshot usage when creating a new virtual device but we did not found this option when searching for it. So we tried to start the default emulator with the command line options for snapshots and got one message that destroyed all our plans and strategies as well as the time we invested until that point, namely that the Android emulator does not support snapshots anymore.

As we now knew that we would not be successful with bringing the snapshots to work, we searched for alternative solutions. Since time was running out we first switched to a virtual machine in which we had the two instances of the emulator running such that we could use the VM for creating snapshots and have a working setting for our work on the modules. Although this setting provided a really poor performance, caused by a very slow internet connection in the VM and since we had to use the slow ARM emulator in the VM because of the fact that the fast x86 emulator requires KVM acceleration which the VM does not provide, it gave us the possibility to work on our modules and for the first time establishing working solutions and scripts. But it was clear to us that we had to switch the setting to face all the performance issues which brought us to a new alternative: The Android x86 project. [x86]. Basically the x86 project is a modified AOSP which is based on the x86 architecture and is delivered as a VirtualBox VM. When starting this VM for the first time we directly noticed many advantages compared to the standard emulator, namely:

- The x86 Project is incredibly fast and solved all our performance issues like running into timeouts during ARTist compilation.
- The internet connection is stable and fast.
- The VM has a built-in superuser functionality which removed the need for our two-emulators-and-copy solution.
- Not a real advantage but very important was that the x86 Project was fully compatible with our Droidmate module.

Of course the biggest disadvantages of using the project was that we would rely on a modified system, but since the x86 Project also provides a possibility to enable native code we decided to focus our work on this setting, which of course brought us to new problems.

The first thing we had to ensure was compatibility of all our modules with the new emulator. While all other modules worked better than before due to the increased performance we had to establish compatibility between ARTist and the VM, on which a more detailed description is given in a later section: Internal Method Tracing.

Another problem we faced was again related to the internet connection. Since our network monitoring module consists of two parts, one outside the emulator and another one inside, we needed a bidirectional connection between the host system and the VM. Simultaneously, the VM needs an internet connection for the app under analysis. Normally, setting the VM network connection to bridged fulfills both of these requirements, but the problem with a bridged network is that the VM represents an own participant in the network which gets its own dynamic IP address which our modules then would have to find out since they establish socket connections to the VM. But now think of the following scenario: When starting the VM it gets a unknown IP address assigned by DHCP, so we have no possibility to send any commands to it. And since the VM

[1] QEMU copy-on-write

also does not know the IP of its host, establishing a connection between both of them would be really difficult. So we relied on the other two network settings, namely NAT, which is responsible for the internet connection and Host-only, which is responsible for the connection to the host. This basically means that we establish two different networks, which also brings a kind of compartmentalization advantage.
Now that the network issue was solved we faced another problem: When loading snapshots of the VM, adb crashed and the possibility to send commands to the emulator in the Android-fashioned style was lost. This turned out to be only a minor error since restarting adb solves the issue, such that we only had to write a simple shell daemon, that restarts adb after a fixed timespan. Then we use the reactivated adb to kill the daemon. Although a little bit hackish, this solution works completely fine, since this daemon is running on the initially loaded snapshot.
Alltogether we now had established a setting with good performance and reliable functionality around which we implemented our control scripts. The overall structure of a analysis run follows the fixed procedure

1) Load the clean snapshot of our x86 Project VM.
2) Continuously try to receive an answer of adb, when succesful use it to kill the adb restarting daemon.
3) Get the package name of the app with Androguard (described in section Component Exploration).
4) Install the app under analysis.
5) Create the output path.
6) Launch the desired modules as Subprocesses such that modules can be added or removed at runtime.
7) Collect all module logs and order them according to the time they were triggered.
8) Revert the VM back to its initial state.

B. Control

When Droidsand is started according to the descriptions in section Usage it follows a fixed order. The main entry point for the analysis is the *run.py* which is responsible for controlling the environment. Following our modular structure we created different python scripts for controlling the backend system and the modules. After entering the main *run* method, the clean snapshot of our VM is restored and loaded, a tunnel interface for the VPN connection (cf. Network Monitoring) is set up and the *ANDROID_ HOME* environment variable is checked. The parsed arguments that define the modules to activate are given to a *function selector* that hands over the control to the script *missioncontrol.py*, which is responsible for setting up the modules. At the beginning, missioncontrol executes some initialization instructions too, e.g. it installs the target app on the emulator and uses Androguard to get the package name of the app. The main part of the analysis is also triggered by missioncontrol. It starts the passive modules that collect the information (strace, network monitoring, artist) as subprocesses and then hands over control to the active component (either the Component Exploration or the Event Stimulation) which then is executed as the main process. As a result **no** additional modules can be activated while exploring the app under analysis directly. After the active module finished its job, the *stop* method of missioncontrol collects the PID's of the app under analysis which is necessary for the Strace module, sorts the logs according to their timestamps and terminates the subprocesses of the passive modules. Finally missioncontrol returns and run terminates the emulator. If the –*snapshot* option is enabled, the control script then also creates a new snapshot of the emulator state such that the user can inspect the emulator after the analysis.

II. Modules

A. Strace

When starting to think about requirements for a dynamic analysis tool or platform one of the first things that came to our mind was syscall monitoring. Syscalls are one of the most important parts of dynamic program behaviour as they representate each and every kernel invocation. And what should come to the mind of every computer scientist when there is a need of monitoring syscalls is the magical swiss army knife strace. Of course Google also noticed the need for such a utility in Android and adapted strace for their mobile operating system... at least in theory. To cite the developer's manual all we had to do is

1) Find out the package name of our app (easily doable using Androguard as described in II-D).
2) "Set up a directory for strace logs:

```
adb shell setenforce 0
adb shell mkdir /data/local/tmp/strace
adb shell chmod 777 /data/local/tmp/strace
```

3) Choose the process to trace before launching it:

```
adb shell setprop
        wrap.com.google.android.browser
        "logwrapper strace -f
        -o /data/local/tmp/strace/
        strace.com.google.android.browser.txt
        "
```

4) Launch the process normally"

[DEV] If it worked this solution would have been perfect for us. All we would have to do is invoking Androguard to get the package and execute a single shell command. With the -f option strace also would collect the syscalls of forked child processes of our app under analysis. Unfortunately the functionality seems to be broken, as testing these steps always led to a reboot of the whole emulator. Trying to fix this problem we found this bug report [BUGa] on Google's issuetracker. At the time this document has been created[2] it seems that there has nothing been fixed yet and that there is no huge interest from Googles' side to change this.
If developers face an unsolved issue that they are not able to fix, what do thay do? Right, they invent some hackish workaround that increases the complexity of their programm exponentially, which is exactly what we did to solve our strace problem. Since calling strace in a shell on the target system seemed to work perfectly we had given the basic functionality. And of course, what we were able to do was executing strace with the package name as argument while our app under analysis was running. The -f flag gave us also the opportunity to monitor forked child processes but we still had a huge problem: If we explore the app and have to reset

[2] 2017-09-28

it (like our final setting actually does), we need to kill the process of the app and restart it, which means that strace will also terminate. We solved this problem by starting strace on zygote with the -f option. Since all other processes are forked from zygote we basically monitor the syscalls of each and every process in the system. But we still had to solve the problem with the different processes after resetting the app for which we invented a not quite simple solution. We identified the Component Exploration as **the only** modul that actively controls starts and stops of the app under analysis, which gave us the possiblity to collect all process IDs of the started instances of the app. This data is returned as a list of process IDs to the controlling script which then simply hands these PIDs to the strace control script *(cf. <Droidsand directory>/analysis_ utils/strace.py)* where a simple 'grep' is executed to filter the complete syscall monitoring for the desired PIDs.
Another problem caused by our workaround was the fact that depending on the app architecture we had to decide whether we should monitor zygote or zygote64. We handled this issue by simply checking if the *oat* files of the app under analysis are placed in a *x86* or a *x86_ 64* folder.
The rest of the strace script is mostly subprocess handling like synchronization solutions that are needed for the user-mode of our platform. Additionally the script deploys some instructions to harden the strace subprocess against bugs and generally make the whole module more robust, like scanning the strace output for errors and then sending feedback to the main control.

B. Network Monitoring

Monitoring the network traffic of the emulator is easy in principle, since the emulator has a built-in tcpdump. The only question we had to solve was how we would be able to filter the traffic such that we only inspect the traffic of the app under analysis. An easy answer is using a VPN tunnel that only allows the app under analysis to access it. For this goal we were able to adapt one of Google's sample apps, namely the *ToyVpn* app. This app runs on our emulator and sets up a vpn connection to a VPN server on the host. We adapted the ToyVpn app in such a way that it only allows the target app to use the connection. Per default the system now directs all network traffic of the target app through the VPN tunnel and blocks all other apps if they try to use the connection. Since the respective VPN server runs on our host we are able to intercept all network traffic with a tcpdump on the tunnel interface and log it as a pcap file.

C. Internal Method Tracing

Assume you have a compiled app you want to analyze. Wouldn't it be nice to see the declared names of all methods at runtime? That is exactly the capability of our *Internal Method Tracing* module which is an adapted version of the *ARTist Project* [ART]. ARTist is "a compiler-based application instrumentation solution for Android that does not depend on operating system modifications and solely operates on the application layer" [BBS+17]. In a nutshell, ARTist operates as a compiler optimization algorithm for the *ART* compiler and hooks the methods of the target app during runtime compilation.
More detailed, ARTist consists of two main components: The *ArtistGui* and the *CodeLib* which are both apk files. The CodeLib defines the hooks to insert and where the target app should be hooked. Then it is packed as an apk such that the defined hooks are in the *dex* binary format used by Android. The actual hooking is executed by the ArtistGui, which is installed as a standard Android app and also can be controlled manually by the user. This also means that our adaption of the ArtistGui can easily be deployed as a compiled APK. For hooking the target app the ArtistGui contains the CodeLib as an asset together with several system libraries that are used during the compilation process and are adapted in such a way that they execute the ARTist "optimization" algorithm which in reality only inserts the hooks.
When the compilation process is started, ARTist takes the dex files from the app to hook and merges the headers of the declared hooks which are contained in the dex files of the CodeLib asset into the original dex files. Then the CodeLib dex file is added to the targets' dex files such that the hooks also can be executed. In the following an intermediate apk is built on which the actual optimization process runs. This optimization process is normally executed by Androids' *dex2oat* tool of which the ArtistGui also has a modified, ARTist considering version in its' assets. The *dex2artist* tool now "optimizes" the app and creates *oat* files, the file format for Androids' *ahead-of-time* compiler. Finally ARTist replaces the original oat files (if existing) with the hooked oat files.
What we did to adapt ARTist to a module of our platform was developing a CodeLib that logs the names of the calling and the called methods (which was fortunately part of the default CodeLib) and used Androids' monkeyrunner tool [MON] to automate the process of triggering the recompilation in the ArtistGui app. During this process we faced a big amount of problems which are described in the following.

1) Development: As usual we started by testing the available version of the tool we wanted to adapt, namely the ARTist app and contacted the developer to get some information on how to use and deploy the tool[3] and directly faced the first problem: ARTist needs root privileges. Since we planned to inspect possible malware we were not really happy about this fact and investigated the reasons for the needed privileges further. By inspecting the logs of the ARTist run we noticed that there are several shell commands which are executed by ARTist during the recompilation and need the privileges. A good example for these commands is *chown*. When the actual recompilation has finished and ARTist has placed the oat files in the oat folder of the target app, it executes a chown on those oat files such that the target app considers them as its' own oat files. Since all other system commands executed during the run were of the same kind of manner, we thought of the possibility to manually execute them in our control scripts and then continuing the ARTist compilation process. Since all our tests were successless, we finally decided to install the superuser app and grant root privileges to ARTist. After switching to the x86 project (as described in section Environment) we decided to completely ignore this issue, since root access controlled by the superuser app is a built-in feature of the x86 project and it would have been difficult to remove this feature.
After this successless try of modifying ARTist, we went back

[3] *At this point we want to thank Oliver Schranz for answering each and every email when we were stuck.*

to testing the tool according to the given manuals on the homepage and indeed we got noticable results. All method names were logged correctly according to the calling order and we also succesfully could distinguish between callers and callees, except on the lowest layer. When testing a simple app we wrote before, which executed a callchain when a special button is pressed, it seemed that if we went down to the lowest methods that were called, ARTist completely ignored them. Investigating the "error" further we finally found out that the root of it was only the simplicity of our testapp, since the lowest methods only consisted of one single instruction which was optimized away by the compiler. After adding more instructions to those methods everything worked as expected.
The next error we faced was one that accompanied us from the beginning of our tests. Despite solely using our testapp, we also used the kicker[4] app for more extensive tests and received always the same error in the logs:

```
07-19 04:12:07.571  3298  3355 E ArtistImpl:
com2.android.dex.DexIndexOverflowException:
Cannot merge new index
65564 into a non-jumbo instruction!
```

Researching the ominous term "non-jumbo instruction" we found out that a standard dex file is only able to hold a predefined number of methods (16bit). If this space is not enough for the developer and if he is unable (or too lazy) to split his methods over several activities, there are two possibilities to increase the space. Either one can use the *multidex* feature or switch to *jumbo instructions*. Clearly this was the cause of our "jumbo-error", the few methods that were added by ARTist to the dex file exceeded the limit. But since the dex file already was compiled we had no possibilty to switch to the jumbo mode so we had to invent another solution. An optimal solution would have been rewriting the dex2oat tool such that it distributes the additional methods defined in the CodeLib evenly but like those additions exceeded the method limit, rewriting the dex2oat tool would exceed our time limit, so we went to a less optimal solution. As already mentioned, the CodeLib provided by the ARTist developers provides a little more functionality than only logging the method names. The logging hooks actually needed by us only consist of two additional methods, so we simply removed all other hooks and accepted the risk of exceeding the limit with a very low probability.
We were now at the point in our timeline were we tried to switch to the x86 project for the first time due to the snapshot issues with the original emulator. And as positively surprised when starting the x86 project VM for the first time, we were quickly disappointed again since seemingly ARTist was not compatible with the VM. To continue the development we switched to a emulator-in-a-vm solution (VMception). But since the VMception was completely inefficient we decided to use it only for further development of the other modules and dig deeper into the compatibility problem. It seemed that when starting the recompiled app under analysis, dex2oat was called again and the oat files were overwritten. This restart was caused by the fact that we still used the libraries for the ARM architecture as assets. After building the libraries as explained on the ARTist homepage for the x86 architecture we came a little bit more far but still found the issue that the compiler options of dex2oat have to fit the system architecture. After changing the code of the ArtistGui such that it starts dex2oat with the right compiler options as shown in Code Adaptions in the ArtistGui and compiling the libraries with the x86 AOSP we were able to successfully start the process. [5]
While now continuing our work on the Method Tracing module we quickly got a new strange error message when the ArtistGui tried to merge the CodeLib into the target app:

```
09-29 10:25:08.927  2276  2348 E
      ArtistImpl: Artist Run() FAILED:
09-29 10:25:08.927  2276  2348 E
      ArtistImpl: java.lang.
      IllegalArgumentException:
      Bad size: -1
09-29 10:25:08.927  2276  2348 E
      ArtistImpl:
      at java.util.zip.ZipEntry.
      setSize(ZipEntry.java:277)
09-29 10:25:08.927  2276  2348 E
      ArtistImpl:
      at saarland.cispa.dexterous.
      Dexterous.buildApk(Dexterous.java:185)
09-29 10:25:08.927  2276  2348 E
      ArtistImpl:
      at saarland.cispa.artist.artistgui.
      compilation.ArtistImpl.
      mergeCodeLib(ArtistImpl.java:135)
09-29 10:25:08.927  2276  2348 E
       ArtistImpl:
       at saarland.cispa.artist.artistgui.
       compilation.ArtistImpl.
       Run(ArtistImpl.java:226)
```

Since the ARTist developer was not able to reproduce this error on ARM we believe that it is also related to the x86 architecture. The main root of the error is located after the merging process, when ARTist tries to add the assets the app had before to the intermediate APK it creates. When inspecting those assets, ARTist checks them for compression, but since it seems that on ARM some assets are always uncompressed, it also uses a kind of whitelist for special file endings (like *png*). Unfortunately at least on the x86 architecture it seems that those filetypes may also be compressed (in fact we found compressed png files in the assets), so using this whitelist leads to dangerous results. This observation lead us to our solution of the bug, concretely a simple additional check if the asset is compressed (see Code Adaptions in Dexterous). [6]
Continuing with preparing ARTist for the final product we ran into the most difficult error we had to solve during the ARTist development. While recompiling the app with dex2oat, we suddenly faced a kernel error declared as a "Broken Pipe" which had its root in a Segmentation Fault on the kernel layer as we found out after long investigations. Investing more time into the problem we could delimit the amount of apps that raise the error to those who use more than one dex file. ust in the time of our investigations the ARTist developers released a new update of the tool and suddenly we got this assertion error instead of the SegFault:

```
09-04 12:58:06.786  2423  2423
      I dex2artist: START ARTIST SETUP (dex2oat)
```

[4]German soccer newspaper

[5]Adaptions were made in the original ArtistGui code in method *ArtistImpl.SetupDex2OatCommand()*

[6]Adaptions were made in the original ArtistGui code in method *Dexterous.buildApk()*

Listing 1. Code Adaptions in the ArtistGui

```
if (config.app_oat_architecture.contains("x86_64")) {
        cmd_dex2oat_compile += " --instruction-set=x86_64";
        cmd_dex2oat_compile += " --instruction-set-features=smp,ssse3,sse4.1,sse4.2,-avx,-avx2";
        cmd_dex2oat_compile += " --instruction-set-variant=x86_64";
        cmd_dex2oat_compile += " --instruction-set-features=default";
} else {
        if (config.app_oat_architecture.contains("x86")) {
                cmd_dex2oat_compile += " --instruction-set=x86";
                cmd_dex2oat_compile += " --instruction-set-features=smp,ssse3,sse4.1,sse4.2,-avx,-avx2";
                cmd_dex2oat_compile += " --instruction-set-variant=x86";
                cmd_dex2oat_compile += " --instruction-set-features=default";
        }
}
```

Listing 2. Code Adaptions in Dexterous

```
final ZipEntry zipEntry = new ZipEntry(fileName);

if ((Config.NO_COMPRESS_EXTENSIONS.contains(fileExtension))
        && (apkContent.getMethod() == ZipEntry.STORED)) {

        Log.d(TAG, String.format(Locale.getDefault(), "> No Compression: %s " +
                "[Method: %d] Size: %d Compressed: %d",
                fileName,
                apkContent.getMethod(),
                apkContent.getSize(),
                apkContent.getCompressedSize()));
                zipEntry.setMethod(ZipEntry.STORED);
                zipEntry.setSize(apkContent.getSize());
```

```
09-04 12:58:06.985  2423  2423
        I dex2artist: ERROR:
        Aborting compilation due to error in ARTist:
        Could not find
        typeLsaarland/cispa/artist/codelib/CodeLib;

09-04 12:58:06.985  2423  2423
        E libc++abi: terminating

09-04 12:58:06.985  2423  2423
        F libc    : Fatal signal 6 (SIGABRT),
        code -6 in tid 2423 (main)
```

Again this error was not reproducable by the developers of ARTist on ARM so we had to solve it on our own and extended our investigations. By inspecting the merged app with Androguard, we noticed that only the first dex file had been correctly merged. This brought us to our productive solution, in which we scan the ARTist log for this specificly known error and activate our second backup mechanism if it occurs. We were able to extract the *dexmerger* tool and create an executable *jar* file of it. Since executing jar files on the emulator is not quite easy we use it only as a backup when the actual error happens. Before the normal compilation run we store backups of the dex files in their original state such that we can pull them if necessary from the emulator, merge our own CodeLib with the jar and then push them again to the emulator.
Last but not least we noticed that some of the system properties of the x86 project were not set correctly and we had to correct them such that they fit the requirements of ARTist.

```
original:
[dalvik.vm.isa.x86.variant]:
        [dalvik.vm.isa.x86.features=default]

required:

[dalvik.vm.isa.x86.features]: [default]
[dalvik.vm.isa.x86.variant]: [x86]
```

2) Integration: After facing all the errors described before, we finally managed to create our own working ArtistGui which is the core of the Internal Method Tracing module. It contains our own CodeLib with exactly the logging functionality that we want. Automating the recompilation process was not hard mostly because the ARTist developers already automated the whole process while testing their tool using the already mentioned *monkeyrunner*. Since monkeyrunner is python based we were easily able to adapt the available functionalities to our own requirements, so the most extensive part we had left was implementing synchronization and control mechanisms together with some error handling specific to our setting (for example handling the assertion error as mentioned above). As a result the ARTist method tracing is completely a standalone module which could also be used on its own and therefore started as a subprocess.

D. Component Exploration

A huge problem in dynamic anaysis especially of apps is the sheer amount of possible triggers, especially considering a reactive system such as Android. Malicous behaviour can be hidden in every activity, triggered by an arbitrary series of UI interactions or inter process communication or arbirtrary

intents received by possibly anonymous broadcast receivers registered at runtime. [RAHB15] To cover as much of these triggers at possible we combine a static analysis approach that we call Component Exploration together with several other modules that will be explained in subsequent sections.

1) Structure & Functionality: In a nutshell the Component Exploration simply extracts all statically declared components of the app and starts them via intents sent over the adb shell. Of course this simple functionality won't impress anybody, so we combined this module with the Droidmate module which we will explain later. The Component Exploration relies heavily on the Androguard Project. [ANG] Androguard is a powerful collection of tools that provide a lot more functionality than actually used by us, like analyzing resources, disassembling DEX or ODEX bytecodes and decompiling DEX or ODEX files. From this huge collection of analysis tools we picked the core functionality of analyzing the manifest of an APK, which provides (together with other functionalities) the ability to extract all components which are statically declared in the manifest by the app.
The entry point for the Component Exploration is the explore method *(cf. <Droidsand directory>/analysis_ utils/androguard.py)*. When it gets called by the controlling script it sets up a Droidmate instance, which means that it simply starts the Droidmate gradle daemon as a subprocess. Droidmate is an automated GUI-exploration tool, which is mostly written in Groovy and Kotlin. Since our scripts are completely in Python, we adapted the tool such that it is able to execute the exploration in a controlled manner. That control is realized by a socket connection which our Python modules use to synchronize their own execution with Droidmate and vice-versa. So far, this basic explanation of its functionality should be sufficient, a detailed explanation follows in the section II-E. Since the Component Exploration as already mentioned is not a very powerful exploration module on its' own, we trigger the Droidmate module by default when the exploration module is activated. After setting up Droidmate, we start the actual Androguard invocation:

```
androlyzed_apk = apk.APK(self.input)
#self.input is the path to the target apk
```

Here we create an instance of an APK-object on which we can invoke several methods to get the entry points for Droidmate. The first information that we extract from the analyzed APK is the package name, which we need later to construct commands that we can pass to adb shell for starting the components. While developing the module, we noticed that we can divide the components that we extract into two categories: visible and invisible components. The visible components are Activities and the invisible components consist of all the background tasks that are executed by the app, namely Services, BroadcastReceivers and ContentProviders. But if we want to trigger functionality that is hidden behind these passive components we need to start an active process, concretely an activity. This is exactly the reason why the explore method starts the main activity of the app at this point with the command

```
adb shell 'am start -n
        "<package_name>/<main_activity>"'
```

We mentioned this command here explicitly because we wanted to clarify the basic command that we use to start every component. It is constructed in the same manner for Activities, Services and with additional extensions also for BroadcastReceivers. Via adb shell we invoke the ActivityManagerService to start the component specified by the given name.
After starting the main activity, we use Androguard to obtain a list of declared services. For each of these services the explore method calls the explore_ service submethod which simply invokes the ActivityManagerService to start the service in the same way as described above. Then it invokes the Droidmate connection to send a "go"-signal via the socket that we established before to start GUI-exploration with Droidmate. The reason why we also invoke Droidmate when triggering passive components is simply a matter of time. If we did not activate Droidmate we would have to sleep for a fixed amount of time, since we want to wait a few moments such that we are really able to log behaviour triggered by the activation of the components. By additionally executing the GUI-exploration we are not only able to use this time but also to cover more GUI-interactions since Droidmate not always chooses the same GUI-elements when exploring the same activity. Since we noticed during our implementation that not all components can be triggered safely without crashing the app, we reset the app each time after starting a component. To give an example, an app may declare old components that were used during some special time frames but not anymore. Imagine a newspaper app that declared a special topic related activity for the elections several months ago, but now those activities are not used anymore and their code has been removed but the programmer forgot to remove their entries in the manifest. If we try to start such an activity, it is no big surprise that the app crashes.
The next components started by the explore method are the declared BroadcastReceivers. As already hinted, the command that we use to start the components by name is not sufficient for BroadcastReceivers. Instead we need the command

```
adb shell 'am broadcast -n
        "<package_name>/<receiver>
        -a <action> [-c <category>]
        [-d <data>]"'
```

Fortunately, Androguard also provides the feature to extract the intent-filter declared within a receiver tag which contains all necessary information to construct the command. All in all, exploring the receivers is the same as when exploring the services: extract the necessary information with Androguard from the APK, activate the receiver with adb shell, start Droidmate exploration and then reset the app.
Although Androguard also delivers us the names of the ContentProviders we needed to follow a different strategy than before when querying them, because we encountered two problems when we tried to construct the corresponding adb shell command:

1) The ActivityManagerService cannot be used to query providers, and
2) We do not know the exact scheme used for querying the provider

Since Androguard also is not able to extract the exact scheme we decided to follow a better-than-nothing strategy and simply query the path of the provider itself as root by executing

```
adb shell 'content query
        --uri content://<provider>
```

—user 0'

hoping that we may see some results. Future work could refine this strategy to get an insight into the saved data at runtime.
The last block of the explore method simply starts all declared activities in the same manner that we used for the services: extract the activity name with Androguard from the APK, start the activity via adb shell, send a "go"-signal to Droidmate and then reset the app. Of course here also may occur system errors caused by an unexpected activity, but resetting the app efficiently handles them.

More danger lies in a runtime error that is thrown by the Droidmate control script. This error indicates that for some reason the Droidmate execution encountered a problem before Droidmate was ready to listen for the incoming connection on the socket, which results in a build error of the gradle process. This build error is recognized by the Droidmate connection which then simply throws the runtime error that is handled according to our defined Error Handling. A lot more severe error can occur when Droidmate wants to terminate its execution because of any problem after initialization. Due to our changes in Droidmate, the program is not allowed to terminate on its own (more on that later) but only when *droidmate.py* sends a "stop" signal. As a result of this construction a circular wait in which Droidmate waits for a signal from the Component Exploration and vice-versa may arise. In this case we have no other possibility than waiting for the socket to throw a TimeoutException which then must be handled by the control script.

In every case the Component Exploration needs to return some information to the control script, namely a list of process IDs. The explanation for this at first glance strange behaviour is the fact that as already explained we need to grep our strace output for the right process. But if the Component Exploration resets the app each time when a new component is started, we also reset the process. By maintaining state of all PIDs we are able to combine those two modules.

E. Droidmate

"Droidmate is a fully automated GUI execution generator for Android apps." [JZ16] When we read this first sentence in the introduction of he Droidmate paper it was directly clear to us that such a functionality is a must have for any dynamic analysis tool. By automating the process of touching and activating the UI-elements in an app it is possible for us to increase the chance of triggering malicious behaviour by an enormously huge factor. To give the reader a better understanding we will now give an in-depth introduction to the functionality and capabilities of Droidmate together with the adaptions we needed to apply for integrating the tool into our own platform.

1) Building Droidmate: Building Droidmate may be the cause of some struggles, as Droidmate requires exactly specified versions of the Android build-tools, platform-tools and sdk-tools. We managed to find these required (older) versions and to save the reader the time of searching in the depths of the incountable download repositories of Google we added them into our repository and give a detailed description in the section Building where to place them such that Droidmate finally starts without blaming the developer for a wrong adb version or any other strange error.

2) Functionality & Adaptions: [7] Since Droidmate is completely written in Groovy and Kotlin we recommend the reader to import it in IntelliJ IDEA to follow our referencings of the classes. In a nutshell, Droidmate delivers two core functionalities:

1) Sensitive API-monitoring by inlining the target app
2) GUI exploration

Clearly the first capability is unnecessary for us because of the inlining functionality of ARTist which delivers inlining at a layer where it cannot be detected by the app under analysis. Droidmate instead performs the inlining on an layer where it may be influenced by the malicous app itself. For this reason the following introduction will only relate to the exploration functionality of Droidmate.
The UI-exploration heavily relies on the *UI Automator* tool that is integrated in Android itself. According to the developers guide, the UI automator "provides a set of APIs to build UI tests that perform interactions on user apps and system apps." [UI]. Droidmate uses the UI Automator to extract the current state of the GUI as an *xml* file, analyzes this state and decides which action to perform, according to a defined "exploration strategy". This action then is performed on the actual device or emulator by invoking the UI automator again via adb. This process is simply repeated until the exploration meets a termination criterion either according to the strategy or as a manifestation of some unexpected behaviour like a crash of the app.
To dig deeper into the functionality we now describe a complete run of Droidmate in its original form from sources, so we start with the preparation. The app under analysis must be placed under *<ROOT_ DIR>/droidmate/apks (where in our case ROOT_ DIR is the analysis_ utils folder)* and the arguments for the run need to be placed in *<ROOT_ DIR>/droidmate/args.txt*. Of course for our platform fixed arguments are needed which we evaluated in advance:

- -deployRawApks **true** and -runOnNotInlined:
  These arguments tell Droidmate to deactivate the inlining functionality and only explore the GUI of the app under analysis.
- -uninstallApk **false**:
  This flag prevents Droidmate from uninstalling the APK after the exploration process (which would be deadly for our other modules).

The main entry class of Droidmate is placed in *Droidmate-Frontend.groovy*[8]. What Droidmate does within this class gives us a complete overview of the tool design: After creating an instance of the class *Configuration* [9] where it simply sets up all

[7] The whole Droidmate code can be found in the GitHub repository of Konrad Jamrozik [REP] *(at this point many thanks to him for his introduction to Droidmate he gave us)* under GPL.

[8] *<ROOT_ DIR>/droidmate/projects/command/src/main/groovy/org/droidmate/frontend*

[9] *<ROOT_ DIR>/droidmate/projects/core/src/main/groovy/org/droidmate/configuration*

parameters according to the given configuration in the *args.txt* it continues with heavy use of the command design pattern. If we have a closer look at the different kinds of commands[10] we can classify them as *Explore Commands, Inline Commands* and *Report Commands*. According to our requirements we do not have to inspect Inline Commands and since we also are not interested in any kind of Droidmate reports we can focus on the Explore Commands.
When looking a the internal fields of an Explore Command we can identify some further setup requirements of Droidmate:

```
private final IApksProvider             apksProvider
private final IAndroidDeviceDeployer    deviceDeployer
private final IApkDeployer              apkDeployer
private final IExploration              exploration
private final IStorage2                 storage2
```

We will now continue with our investigation of the exploration process by having a close look at those fields, how they are set up and executed (except the Storage2 object which we can skip because it simply provides some methods to control and manage the directories and files). By looking at the **build** method of the Explore Command we obtain the right entry points for those components.

a) ApksProvider: [11] As the name of the class indicates the ApksProvider class simply provides a possibility to access the APK under analysis in a programmatical way. By using a java representation of the Android tool *aapt* (contained in *DeviceTools.groovy*) it is able to decompose the app into its single components.

b) AndroidDeviceDeployer: [12] Like the ApksProvider provides programmatical access to the APK, the AndroidDeviceDeployer does for the device, may it be a real device or the emulator. Besides providing a wrapper for using adb, the DeviceDeployer contains the possibility for accessing the Logcat, the uiautomator via TCP and some statical information on the device model within an instance of the class AndroidDevice[13]. When having a closer look at this class one can inspect all those functionalities.

c) ApkDeployer: [14] The ApkDeployer simply supplies some basic actions to execute with the APK on the device for the Explore Command like installing and uninstalling the app on the device. Here we encountered a first problem when integrating Droidmate into our platform: Droidmate always reinstalls the app when starting the exploration. This is done for robustness reasons (to make sure that the app is installed) but also since the app of course must be installed again when using the inlining capabilities. For our other modules this reinstallation would be a worst case scenario when Droidmate is activated at runtime so we had to remove it.

d) Exploration: [15] Last but not least we will have a closer look at the most important class for us, the Exploration class, which we modified most heavily. We can ignore the Configuration and ITimeProvider but what has a direct influence on the exploration process is the *StrategyProvider* where the strategy which the exploration should follow is defined. At the moment we rely on the default strategy with which Droidmate performs up to ten actions if it does not meet an unexpected termination criterion and performs a depth-oriented approach. At the point the code in the Exploration class is executed, Droidmate already has set up an instance of the class *RobustDevice* which holds methods for programmatical access to all needed actions on the device. Here we already faced the first difficulty of the integration. Since the setup and initialization of an instance of the class RobustDevice needs all described steps above, it takes a non-negligible amount of time. If we also consider the fact that our plan was to invoke Droidmate everytime when starting a component with the ComponentExploration and that the initialization phase takes about 30 seconds we would have end up with a needed time of *n*30+x+init* seconds where *n* is the number of components, *x* the time Droidmate needs to execute the actions on the device and *init* the time that our scripts need to set up the whole environment, not considering synchronization overhead and the time our other scripts need. Clearly this amount of time was completely inacceptable for us, so we had to invent a way of reducing the time needed by Droidmate. That was the point where we wanted to identify the point where the real exploration happens. Clearly the first instructions

```
try
{
  tryDeviceHasPackageInstalled(device,
                    app.packageName)
  tryWarnDeviceDisplaysHomeScreen(device,
                    app.fileName)
} catch (DeviceException e)
{
  return new Failable<IApkExplorationOutput2,
                    DeviceException>(null, e)
}
```

are only part of the preparation process and only need to be executed once in our setting. With these method calls Droidmate simply ensures that the app is indeed installed and that the device displays the home screen, which is due to the fact that Droidmate assumes that its exploration is always started within the MainActivity of the app. This indicated another problem for our setting but fortunately we could ignore this call since it only displays a warning in the log which tells that Droidmate is "Continuing the exploration nevertheless, hoping that the first reset app exploration action will force the device into the home screen."
After those preparation steps Droidmate now finally is ready to execute the real exploration in the method *explorationLoop*. In this loop Droidmate creates instances of the classes *RunnableExplorationAction, RunnableResetAppExplorationAction, RunnableTerminateExplorationAction, RunnableWidgetExplorationAction* and *PressBackExplorationAction*[16]. We won't give a detailed explanation of them, as their names clearly indicate their functionality, which is enough at the moment. These actions are created according to the ExplorationStrategy as described before and serve the

[10] *<ROOT_ DIR>/droidmate/projects/command/src/main/groovy/org/droidmate/command*
[11] *<ROOT_ DIR>/droidmate/projects/core/src/main/groovy/org/droidmate/tools*
[12] 11
[13] *<ROOT_ DIR>/droidmate/projects/core/src/main/groovy/org/droidmate/device*
[14] 11
[15] *<ROOT_ DIR>droidmate/projects/command/src/main/groovy/org/droidmate/command/exploration*
[16] *<ROOT_ DIR>/droidmate/projects/core/src/main/groovy/org/droidmate/exploration/actions*

purpose to execute UI-interactions on the screen of the device. But the fact that they are wrapped into the explorationLoop made this point perfect for our adaptions. We wrapped the explorationLoop into another loop that is controlled by a socket that receives "go"-signals by the ComponentExploration and then answers with a signal for succesful exploration. So instead of executing several Droidmate runs we simply execute one single Droidmate run that executes the explorationLoop for each component started by us. As simple as this approach seems, it is highly effective as it solves all setup problems coming along with the initialization process. The only obstacle left was that at the begin of the explorationLoop Droidmate executes a *ResetAppExplorationAction* from which we had to take the whole functionality as a result of this problem, which removes a part of Droidmates robustness. But our tests have shown that the advantage of our solution is bigger than the the disadvantages since we now effectively overcame the Python-Java-border and are able to control Droidmate from within our scripts. For completeness reasons we want to add here, that we also had to remove the output generation of Droidmate, since there seems to be a bug when starting the graphical visualization program gnuplot which is used for generating the outputs, but since we are not interested in those outputs this should be no problem.

F. Event Stimulation

Android Malware often registers for system-wide Android events to get notified of the chosen events and then trigger malicious actions. [ZJ12] To trigger this behaviour we developed the Event Stimulation module which implements this ablity in a quite simple manner. The Android SDK contains a text file in which all actions sent as system events are listed. Our first approach was that, while activated, the Event Stimulation module only waits for a random timespan between 0 and 30 seconds, then picks randomly one of those actions and sends the corresponding intent via adb shell with the command

```
adb shell 'am broadcast
        -a <action>'
```

Then the action is logged with a timestamp to identify triggered behaviour of the app.
Problematic was the fact, that some of the actions are more important for malware then others, so we switched from the random approach to ordering the actions by priority. Additionally we added the possibility to adapt the timespan how long the stimulation should run by passing an argument
Furthermore we faced the fact that some events seem to crash the system process which results in a reboot of the emulator. Of course this is completely inacceptable for us, so we removed those actions by testing. While this is completely fine for some unimportant events, we had to invent an alternative solution for others, like the *BOOT_ COMPLETED* action, since malware often registers for those actions. What turned out to be a working solution was extending the adb shell command such that the intent is only sent to the app under analysis. Eventually the other actions only result in a crash because the intents do not contain valid data. Future efforts will be made by us to evaluate valid data for all the system events since also for events that do not crash the system it would be quite helpful if they delivered vaild data which may be intercepted by malware. The list of raw actions that we used is listed in Appendix A: *List of Actions used in Android System Events* for the convenience of over-enthusiastic readers.

III. Disguise Techniques

One of the most important capabilities a dynamic analysis platform needs to deliver is to hide the fact that the target app is analyzed. In our setting we integrated some little features to implement this capability but we want to invest a lot more time in it in future efforts. At the moment we simply provide a big fake contact database which we tried to make look like realistic contacts, we installed the most usual apps of a standard enduser.
Unfortunately we do not yet have features to hide indicators for the VM that we use or any other fake sensor data, but a lot of ideas what can be done there.

IV. Building

1) Install python3

```
sudo apt-get install python3
```

2) Install the python libraries needed by Androguard.

```
sudo apt install
python3-pyqt5 python3-pyperclip
python3-networkx ipython3
python3-future python3-pyasn1
python3-cryptography python3-magic
python3-pydot
```

3) Install openjdk8 and make sure that the JAVA_ HOME environment variable points to its directory.

```
sudo apt-get install openjdk-8-jre
export JAVA_HOME=/path/to/openjdk8
```

4) Install Android SDK API19 and API23 and make sure that the folder structure completely equals *<SDK_ location>/platforms/android-19* and *<SDK_ location>/platforms/android-23*, if not rename the folders. Take care that a folder *licenses* with a valid Android license is contained in the folder (happens automatically when installing with AndroidStudio).
5) In the folder that contains the SDK replace the folders *build-tools*, *platform-tools* and *tools* with those provided by us.
6) Set the ANDROID_ HOME to the SDK directory.

```
export ANDROID_HOME=/path/to/sdk
```

7) Install VirtualBox.

```
sudo apt-get install virtualbox
```

8) Install gradle.[17]
9) Install Apache ant and add its *bin* directory to the PATH environment variable.[18]
10) Clone our git-repository.
11) Switch into the repository folder and execute *build.sh*
12) Execute *hostonlysetup.sh* with root privileges.

[17]Tested with version 4.0, for installation instructions consider https://gradle.org/

[18]Tested with version 1.10.1, for installation intructions consider http://ant.apache.org/

13) Import our virtual machine (*LInux.ova*) in VirtualBox.
14) Start the VM and wait for Android to boot up.
15) Switch into Terminal-Mode (*ALT + F1*).
16) Execute *./data/local/fredhunter.sh &*
17) Switch back to Window-Mode (*ALT + F7*).
18) In a terminal on the host execute *vboxmanage snapshot LInux take Droidsand*. It is important that the snapshot exactly is named "Droidsand".
19) Make sure that your user is granted to use *tcpdump* without needing root privileges.

V. Usage

Starting Droidsand is quite easy. Simply navigate to the root project folder and execute

```
python3 run.py <path-to-apk>
        <path-to-output-directory>
        [-h|-u|-f|(-s)?(-n)?(-c)?(-a)?(-e(-t)?)?)
                (--snapshot)?]
```

where <path-to-ap>is the path to the apk file which shall be analyzed and <path-to-output-directory>is the path where all the obtained data will be saved. We will now give a short overview on the optional arguments.

- -h (–help)
  Shows the help message and exits.
- -u (–user)
  Activates the user mode where all modules can be dynamically activated and deactivated at runtime via command prompt.
- -f (–full)
  Activates the full automatical mode, which means all modules are activated. In full analysis, ComponentExploration and Event Stimulation are sequentially executed.
- -s (–strace)
  Activates syscall tracing with strace.
- -n (–network)
  Activates network traffic monitoring and logging.
- -c (–exploration)
  Activates the Component Exploration **and** Droidmate.
- -a (–artist)
  Activates Internal Method Tracing with ARTist.
- -e (–events)
  Activates the Event Stimulation.
- -t (–time)
  Specifies how long the Event Stimulation should run (Default: 30s).
- –snapshot
  Enables creating a snapshot of the emulator state after the analysis run.

The default behaviour of Droidsand is full analysis mode. Note that activation of the ComponentExploration excludes parallel activation of the EventStimulation and vice-versa, since the emulator tends to crash if both are activated. If both flags are set, they are executed sequentially.

VI. Error Handling

During the complete analysis process there may occur several errors for which we implemented handling mechanisms to increase the robustness of the whole platform. All modules permanently monitor the execution of the started subprocess for error indicators like a thrown *CalledProcessError*, as well as they scan the output of the subprocess for failures. As an example, the gradle build daemon of Droidmate may encounter a problem and fail. In this case our Droidmate control script will notice this failure and continue exploration without Droidmate. For some known issues we have integrated backup mechanisms, for example when ARTist does not merge apps correctly due to compression as already explained, we merge them manually as described in Internal Method Tracing. If the passive modules (strace, network monitoring, internal method tracing) face an unknown error, they stop the respective subprocesses and the analysis continues without the module.
Our heavy use of adb to control the emulator obviously makes us vulnerable when encountering a problem with the tool. To harden our platform against issues with adb we also integrated a backup mechanism for those problems. All adb methods[19] catch a thrown CalledProcessError and handle it by restarting adb. After trying this procedure twice without success, we classify the error as critical and abort the analysis run. Adb commands that fail due to the command itself will also lead to aborting but the error will be printed out such that the user can investigate the problem further.
The most critical part of the analysis run are the active modules (Component Exploration, Droidmate, Event Stimulation). Although Droidmate has built-in capabilities to reset apps and is even able to confirm the error message "Unfortunately app x has stopped.", which together with resetting the app in the Component Exploration is a really robust handling of the case that an app crashes, there exists the possibility that triggering a specific component together with Droidmate leads to a whole system crash. Even more severe is the fact that if the emulator crashes, the ComponentExploration will send a "go" signal to Droidmate, but Droidmate does not acknowledge the successful exploration, but waits for a "stop" signal such that both tools are blocked. In this case we have no other possibility than raising a *TimeOutError* from the socket connection. Another indicator for a crashed emulator is a TimeOutError thrown by adb. We noticed this event in some of our tests and invested a lot of time to handle it, but the main problem is that rebooting the emulator is no solution, because then we would have to somehow interrupt the analysis, collect the results so far and then exactly restore the state in which we were before the error occured and continue analysis except triggering the one component that led to the crash. This approach would have raised the need for extensive code adaptions and it does not completely ensure that the error is handled safely. Preventing the error also is no option, since we cannot predict which components will lead to the crash. We finally decided to accept the possibility and to limit the damage in the best way as possible by handling it as we handle all other unknown errors. Concretely, we catch TimeOutErrors and RuntimeErrors by collecting and ordering all analysis information obtained so far in the usual manner and then terminating the execution.

[19] *adbutils.py*

VII. Evaluation

A. Performance evaluation

To test the performance of our tool we executed a complete analysis run with 32 apps, which took seven hours. The complete list of app names can be found in Appendix C: Apks used in performance tests.

B. Evaluation with Malware Samples

We picked four examples from the Kharon Malware database [KHA] and inspected the results to identify strengths and weaknesses of our tool. In the following we will present the results for the samples.

a) WipeLocker: [20] The big winner of the WipeLocker run is clearly the internal method tracing module. We were able to see all steps of the malware execution as described in the detailed description on the database. On startup, ARTist logs the creation of the *IntentServiceClass*, as well as we can see the calls of *getTopActivity* and *Async_ sendSMS* after regular timespans. We also were able to identify the method that demands device administrator privileges from the user:

```
09-26 12:41:41.469  2969  2969
D ArtistCodeLib: Caller ->
com.elite.IntentServiceClass.
access$0(IntentServiceClass.java:19)
09-26 12:41:41.469  2969  2969
D ArtistCodeLib: Caller ->
com.elite.DeviceManager.
<init>(DeviceManager.java:14)
09-26 12:41:41.470  2969  2969
D ArtistCodeLib: Caller ->
com.elite.MyServices$Async_sendSMS.
<init>(MyServices.java:175)
09-26 12:41:41.470  2969  2969 D
ArtistCodeLib: Caller ->
com.elite.MyServices.
getTopActivity(MyServices.java:75)
```

On the other hand we noticed a crucial problem of using Droidmate. The tool is not able to accept or decline permission requests since they do not belong to the app itself but to the system. But the app even triggers its malicious behaviour when the request is ignored. In our strace logs we could identify the traversal of the app to delete the files on the SD card.

```
[pid  2969] newfstatat(AT_FDCWD,
"/storage/emulated/0/Pictures/1.jpg",
{st_mode=S_IFREG|0660, st_size=4265, ...},
0) = 0
[pid  2969] newfstatat(AT_FDCWD,
"/storage/emulated/0/Pictures/1.jpg",
{st_mode=S_IFREG|0660, st_size=4265, ...},
AT_SYMLINK_NOFOLLOW) = 0
[pid  2969] unlinkat(AT_FDCWD,
"/storage/emulated/0/Pictures/1.jpg",
0) = 0
```

We could confirm the strace results by inspecting the snapshot made after execution and checking the respective directiories. Additionally ARTist later showed us a constantly called receiver which we assume to be the *SMSReceiver* described in the malware report.

```
09-26 12:42:57.712  2969  2969
D ArtistCodeLib: Caller ->
com.elite.MyServices$1.
onReceive(MyServices.java:119)
```

b) Videoplayer: [21] The Video Player ransomware may not be the best example for malware analysis since its' controlling server has been taken down. Nevertheless, we decided to run our platform with it as the target and indeed our ARTist logs show that the Android *Volley* network package is heavily used and that the app tries to establish a connection.

```
09-30 14:35:52.404  2982  3022 D
ArtistCodeLib: Caller ->
com.android.volley.Request.getUrl(Request.java:25
09-30 14:35:52.405  2982  3022 D
ArtistCodeLib: Caller ->
com.android.volley.Request.
getCacheKey(Request.java:258)
09-30 14:35:52.405  2982  3022 D
ArtistCodeLib: Caller ->
com.android.volley.Request.
addMarker(Request.java:180)
09-30 14:35:52.407  2982  3026 D
ArtistCodeLib: Caller ->
com.android.volley.Request.
addMarker(Request.java:180)
09-30 14:35:52.408  2982  3026 D
ArtistCodeLib: Caller ->
com.android.volley.Request.
isCanceled(Request.java:287)
```

The main reason why we decided to analyze this specific malware is because the ip address of the control server is known such that we could easily check the results of our network monitoring tool and we were not disappointed. The network logs clearly show that the malware tries to establish a connection to the ip 148.251.154.104, which is the known ip of the control server.

c) SaveMe: [22] By inspecting the SaveMe malware we recognized another weakness of our tool. An error that leads to an emulator crash seems to influence the network monitoring results. Although we were able to see all the steps the malware does in the ARTist log, namely the AsyncTasks *sendmyinfos()*, *sendstatus()* and *senddata()* together with the services *CHECKUPD* and *GTSTSR* we were not able to log the explicit communication with the control server with our network monitoring module, due to the fact that the emulator hung up after a Google Service crashed.

```
09-30 14:36:25.643  2957  2957 D
ArtistCodeLib: Caller ->
```

[20] http://kharon.gforge.inria.fr/dataset/malware_WipeLocker.html

[21] http://kharon.gforge.inria.fr/dataset/malware_Videoplayer.html

[22] http://kharon.gforge.inria.fr/dataset/malware_SaveMe.html

```
com.savemebeta.GTSTSR.<init>(GTSTSR.java:81)
09-30 14:36:25.659  2957  2957 D
ArtistCodeLib: Caller ->
com.savemebeta.
GTSTSR$StatusTask.<init>(GTSTSR.java:144)
09-30 14:36:25.659  2957  2957 D
ArtistCodeLib: Caller ->
com.savemebeta.GTSTSR.
onCreate(GTSTSR.java)
...
09-30 14:36:30.231  2957  2994 D
ArtistCodeLib: Caller ->
com.savemebeta.
CHECKUPD$sendmyinfos$1.<init>(CHECKUPD.java:1)
09-30 14:36:30.236  2957  2994 D
ArtistCodeLib: Caller ->
com.savemebeta.CHECKUPD$sendmyinfos.
doInBackground(CHECKUPD.java:1)
09-30 14:36:31.237  2957  2996 D
ArtistCodeLib: Caller ->
com.savemebeta.CHECKUPD$sendmyinfos.
access$1(CHECKUPD.java:135)
09-30 14:36:31.237  2957  2996 D
ArtistCodeLib: Caller ->
com.savemebeta.CHECKUPD$sendmystatus.
<init>(CHECKUPD.java:181)
09-30 14:36:31.238  2957  2996 D
ArtistCodeLib: Caller ->
com.savemebeta.CHECKUPD$sendmyinfos$1.
run(CHECKUPD.java:171)
```

d) BadNews: This time we found the first hint of malicious behaviour with the network monitoring module. Despite communicating with the admob servers, the app established a connection to *http://www.mobidisplay.net/api/adv.php* which is quite similar to the "intentionally anonymized" URL that is given in the database description. Even more suspicous was the fact that the app sent some phone data to this server, although not all since for example our emulator has no phone number. We believe that due to exactly this fact, concretely that we provided no real phone data, the last step of the malware run (installing additional apps) was not executed. But we could find more evidence that the malware indeed triggered its' behaviour by searching in the ARTist logs where we finally found exactly the methods named in the database.

```
09-30 14:33:43.603  2529  2529 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
BootReceiver.<init>(BootReceiver.java)
09-30 14:33:43.604  2529  2529 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
BootReceiver.isMyServiceRunning
(BootReceiver.java:32)
09-30 14:33:43.605  2622  2622 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
AdvService.<init>(AdvService.java)
09-30 14:33:43.606  2529  2529 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
BootReceiver.onReceive
(BootReceiver.java:16)
09-30 14:33:43.607  2622  2622 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
AdvService.startUpdater
(AdvService.java:179)
09-30 14:33:43.607  2622  2622 D
ArtistCodeLib: Caller ->
com.mobidisplay.advertsv1.
AdvService.onStartCommand
(AdvService.java)
```

VIII. Future Work

Although we already realized many core components for our platform, we'd still like to extend it in some ways. The most important thing we want to realize in the future is the inspection of *Inter-Process-Comunication (IPC)* which plays a very important role in such an interactive system as Android, where a lot communication between the processes is handled via *Intents*. What is relatively easy doable is hooking into the *Binder* class and intercept all *Binder transactions*, which are the most important part of the IPC. [AR14] The more difficult point is to translate those transactions into a human readable format, such that it is possible to understand for example which functions of which service are called. We already tested a tool called *jtrace* [Lev] which is advertised to be an "augmented, Android aware strace" and to provide the capability of "Binder message parsing". Unfortunately, the developer exaggerates the capabilities of his tool, as the most of the output are only the transaction codes in hex-encoding. Another tool that we found but which we not tested yet since it requires kernel modifications which in our current setting could lead to deadly cross modification caused bugs since we already use the modified kernel of the x86 project, is a tool the developer calls *BinderFilter* [WB16]. Its code is publicly accessible and if the results are decoded in the same manner as the samples contained in the repository it could be the ideal addition to our platform. Alternatively we could consider an approach like used by the *Copperdroid* [TKFC15] developers, who implemented a technique they call *unmarshalling oracle* which is basically an unmodified emulator that is used for decoding the parcel data during the binder transactions and that additionally uses an AIDL database to map known services and their methods to the corresponding transaction codes. Either way, the inter process communication is a lot of analysis data that we want to capture such that a module for analyzing it is indispensable.

During our tests we also noticed a need for refining Droidmates' exploration strategy as well as improving its' behaviour in special situations like permission requests. Thos steps would not require a lot of effort but extremely improve the efficiency of the GUI exploration.

Another part of the project that leaves room for improvements are the Disguise Techniques. What needs to be done here in every case is hiding indicators for the execution in a VM like the virtual device hardware entries. But there exists a lot of more ways to provide realistic device behaviour like fake data from sensors e.g. the gyroscope. Together with this comes the need for actual data in the Intents that we use when executing the EventStimulation.

An additional nice-to have would be a possibility to taint-track variables as proposed in TaintDroid. [ECJea10] What we could imagine too would be a monitoring of some sensitive API's (like camera, contacts,...) such that direct access to those API's would be logged separately which would make evaluating the analysis results more easy. What comes also along with evaluating the analysis results would be something like a "Dashboard-Generator" that collects all results and converts them to a user-friendly html overview.
Finally we plan to port the project to Android Nougat (7.1.1) which should not be difficult as ARTist also suports Nougat and Droidmate is currently ported to the newer version.

REFERENCES

[AND] Android souce - downloading and building. https://source.android.com/source/requirements. Accessed: 2017-09-16.

[ANG] https://github.com/androguard/androguard. Accessed: 2017-09-05.

[AR14] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls ipc, controls the droid. 2014. Accessed: 2017-09-12.

[ART] Homepage of the artist project. https://artist.cispa.saarland/. Accessed: 2017-09-28.

[BBS+17] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017.

[BP] Build patch - unsupported reloc 43. https://android-review.googlesource.com/#/c/223100/1/core/clang/HOST_x86_common.mk. Accessed: 2017-09-16.

[BUGa] wrap.<packagename>functionality broken. https://issuetracker.google.com/issues/62179655. Accessed: 2017-09-07.

[BUGb] emulator-x86 + kvm triggers an endless loop in qemu-setup.c. https://issuetracker.google.com/issues/36949985. Accessed: 2017-09-19.

[DEV] Android developers guide - strace. https://source.android.com/devices/tech/debug/strace. Accessed: 2017-09-07.

[ECJea10] William Enck, Landon P. Cox, Jaeyeon Jung, and et al. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. http://www.appanalysis.org/tdroid10.pdf, 2010. Accessed: 2017-09-12.

[JZ16] Konrad Jamrozik and Andreas Zeller. Droidmate: A robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 293–294, New York, NY, USA, 2016. ACM.

[KHA] Kharon malware dataset. http://kharon.gforge.inria.fr/dataset/index.html. Accessed: 2017-09-30.

[Lev] Jonathan Levin. jtrace - augmented, android aware strace. http://newandroidbook.com/tools/jtrace.html. Accessed: 2017-09-12.

[MON] Android developers guide - monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html. Accessed: 2017-09-28.

[Q] Homepage of the qemu emulator software. https://www.qemu.org/. Accessed: 2017-09-16.

[RAHB15] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. How current android malware seeks to evade automated code analysis. In *9th International Conference on Information Security Theory and Practice (WISTP'2015)*, August 2015.

[RDB+11] Christian Rossow, Christian J. Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten van Steen, Felix C. Freiling, and Norbert Pohlmann. Sandnet: Network Traffic Analysis of Malicious Software. In *Proceedings of Building Analysis Datasets and Gathering Experience Returns for Security - ACM BADGERS Workshop*, 2011.

[REP] Droidmate - github repository. https://github.com/konrad-jamrozik/droidmate. Accessed: 2017-09-10.

[TKFC15] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *22nd Annual Network and Distributed System Security Symposium, San Diego, California, USA*, 2015. Accessed: 2017-09-12.

[UI] Android developers guide - ui automator. https://developer.android.com/training/testing/ui-automator.html. Accessed: 2017-09-10.

[WB16] David Wu and Sergex Bratus. A context-aware kernel ipc firewall for android. http://www.cs.dartmouth.edu/~sergey/binderfilter/binderfilter_tr.pdf, 2016. Accessed: 2017-09-12, Github repository on https://github.com/dxwu/BinderFilter.

[x86] Android-x86 - porting android to x86. http://www.android-x86.org/. Accessed: 2017-09-23.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

APPENDIX

A. List of Actions used in Android System Events

All actions that result in a crash are removed.

android.intent.action.BOOT_COMPLETED
android.intent.action.PHONE_STATE
android.app.action.ACTION_PASSWORD_CHANGED
android.app.action.ACTION_PASSWORD_EXPIRING
android.app.action.ACTION_PASSWORD_FAILED
android.app.action.ACTION_PASSWORD_SUCCEEDED
android.app.action.DEVICE_ADMIN_DISABLED
android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
android.app.action.DEVICE_ADMIN_ENABLED
android.app.action.DEVICE_OWNER_CHANGED
android.app.action.INTERRUPTION_FILTER_CHANGED
android.app.action.LOCK_TASK_ENTERING
android.app.action.LOCK_TASK_EXITING
android.app.action.NEXT_ALARM_CLOCK_CHANGED
android.app.action.NOTIFICATION_POLICY_ACCESS_GRANTED_CHANGED
android.app.action.NOTIFICATION_POLICY_CHANGED
android.app.action.PROFILE_PROVISIONING_COMPLETE
android.app.action.SYSTEM_UPDATE_POLICY_CHANGED
android.bluetooth.a2dp.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.a2dp.profile.action.PLAYING_STATE_CHANGED
android.bluetooth.adapter.action.CONNECTION_STATE_CHANGED
android.bluetooth.adapter.action.DISCOVERY_FINISHED
android.bluetooth.adapter.action.DISCOVERY_STARTED
android.bluetooth.adapter.action.LOCAL_NAME_CHANGED
android.bluetooth.adapter.action.SCAN_MODE_CHANGED
android.bluetooth.adapter.action.STATE_CHANGED
android.bluetooth.device.action.ACL_CONNECTED
android.bluetooth.device.action.ACL_DISCONNECTED
android.bluetooth.device.action.ACL_DISCONNECT_REQUESTED
android.bluetooth.device.action.BOND_STATE_CHANGED
android.bluetooth.device.action.CLASS_CHANGED
android.bluetooth.device.action.FOUND
android.bluetooth.device.action.NAME_CHANGED
android.bluetooth.device.action.PAIRING_REQUEST
android.bluetooth.device.action.UUID
android.bluetooth.devicepicker.action.DEVICE_SELECTED
android.bluetooth.devicepicker.action.LAUNCH
android.bluetooth.headset.action.VENDOR_SPECIFIC_HEADSET_EVENT
android.bluetooth.headset.profile.action.AUDIO_STATE_CHANGED
android.bluetooth.headset.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.input.profile.action.CONNECTION_STATE_CHANGED
android.bluetooth.pan.profile.action.CONNECTION_STATE_CHANGED
android.hardware.action.NEW_PICTURE
android.hardware.action.NEW_VIDEO
android.hardware.hdmi.action.OSD_MESSAGE
android.hardware.input.action.QUERY_KEYBOARD_LAYOUTS
android.intent.action.ACTION_POWER_CONNECTED
android.intent.action.ACTION_POWER_DISCONNECTED
android.intent.action.ACTION_SHUTDOWN
android.intent.action.AIRPLANE_MODE
android.intent.action.APPLICATION_RESTRICTIONS_CHANGED
android.intent.action.BATTERY_CHANGED
android.intent.action.BATTERY_LOW
android.intent.action.BATTERY_OKAY
android.intent.action.CAMERA_BUTTON
android.intent.action.CONFIGURATION_CHANGED
android.intent.action.CONTENT_CHANGED

android.intent.action.DATA_SMS_RECEIVED
android.intent.action.DATE_CHANGED
android.intent.action.DEVICE_STORAGE_LOW
android.intent.action.DEVICE_STORAGE_OK
android.intent.action.DOCK_EVENT
android.intent.action.DOWNLOAD_COMPLETE
android.intent.action.DOWNLOAD_NOTIFICATION_CLICKED
android.intent.action.DREAMING_STARTED
android.intent.action.DREAMING_STOPPED
android.intent.action.EXTERNAL_APPLICATIONS_AVAILABLE
android.intent.action.EXTERNAL_APPLICATIONS_UNAVAILABLE
android.intent.action.FETCH_VOICEMAIL
android.intent.action.GTALK_CONNECTED
android.intent.action.GTALK_DISCONNECTED
android.intent.action.HEADSET_PLUG
android.intent.action.HEADSET_PLUG
android.intent.action.INPUT_METHOD_CHANGED
android.intent.action.LOCALE_CHANGED
android.intent.action.MANAGE_PACKAGE_STORAGE
android.intent.action.MEDIA_BAD_REMOVAL
android.intent.action.MEDIA_BUTTON
android.intent.action.MEDIA_CHECKING
android.intent.action.MEDIA_EJECT
android.intent.action.MEDIA_MOUNTED
android.intent.action.MEDIA_NOFS
android.intent.action.MEDIA_REMOVED
android.intent.action.MEDIA_SCANNER_FINISHED
android.intent.action.MEDIA_SCANNER_SCAN_FILE
android.intent.action.MEDIA_SCANNER_STARTED
android.intent.action.MEDIA_SHARED
android.intent.action.MEDIA_UNMOUNTABLE
android.intent.action.MEDIA_UNMOUNTED
android.intent.action.MY_PACKAGE_REPLACED
android.intent.action.NEW_OUTGOING_CALL
android.intent.action.NEW_VOICEMAIL
android.intent.action.PACKAGE_ADDED
android.intent.action.PACKAGE_CHANGED
android.intent.action.PACKAGE_DATA_CLEARED
android.intent.action.PACKAGE_FIRST_LAUNCH
android.intent.action.PACKAGE_FULLY_REMOVED
android.intent.action.PACKAGE_INSTALL
android.intent.action.PACKAGE_NEEDS_VERIFICATION
android.intent.action.PACKAGE_REMOVED
android.intent.action.PACKAGE_REPLACED
android.intent.action.PACKAGE_RESTARTED
android.intent.action.PACKAGE_VERIFIED
android.intent.action.PROVIDER_CHANGED
android.intent.action.PROXY_CHANGE
android.intent.action.REBOOT
android.intent.action.SCREEN_OFF
android.intent.action.SCREEN_ON
android.intent.action.TIMEZONE_CHANGED
android.intent.action.TIME_SET
android.intent.action.TIME_TICK
android.intent.action.UID_REMOVED
android.intent.action.USER_PRESENT
android.intent.action.WALLPAPER_CHANGED
android.media.ACTION_SCO_AUDIO_STATE_UPDATED
android.media.AUDIO_BECOMING_NOISY
android.media.RINGER_MODE_CHANGED
android.media.SCO_AUDIO_STATE_CHANGED
android.media.VIBRATE_SETTING_CHANGED

```
android.media.action.CLOSE_AUDIO_EFFECT_CONTROL_SESSION
android.media.action.HDMI_AUDIO_PLUG
android.media.action.OPEN_AUDIO_EFFECT_CONTROL_SESSION
android.net.conn.BACKGROUND_DATA_SETTING_CHANGED
android.net.conn.CONNECTIVITY_CHANGE
android.net.nsd.STATE_CHANGED
android.net.scoring.SCORER_CHANGED
android.net.scoring.SCORE_NETWORKS
android.net.wifi.NETWORK_IDS_CHANGED
android.net.wifi.RSSI_CHANGED
android.net.wifi.SCAN_RESULTS
android.net.wifi.STATE_CHANGE
android.net.wifi.WIFI_STATE_CHANGED
android.net.wifi.p2p.CONNECTION_STATE_CHANGE
android.net.wifi.p2p.DISCOVERY_STATE_CHANGE
android.net.wifi.p2p.PEERS_CHANGED
android.net.wifi.p2p.STATE_CHANGED
android.net.wifi.p2p.THIS_DEVICE_CHANGED
android.net.wifi.supplicant.CONNECTION_CHANGE
android.net.wifi.supplicant.STATE_CHANGE
android.nfc.action.ADAPTER_STATE_CHANGED
android.os.action.DEVICE_IDLE_MODE_CHANGED
android.os.action.POWER_SAVE_MODE_CHANGED
android.provider.Telephony.SIM_FULL
android.provider.Telephony.SMS_CB_RECEIVED
android.provider.Telephony.SMS_DELIVER
android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED
android.provider.Telephony.SMS_RECEIVED
android.provider.Telephony.SMS_REJECTED
android.provider.Telephony.SMS_SERVICE_CATEGORY_PROGRAM_DATA_RECEIVED
android.provider.Telephony.WAP_PUSH_DELIVER
android.provider.Telephony.WAP_PUSH_RECEIVED
android.speech.tts.TTS_QUEUE_PROCESSING_COMPLETED
android.speech.tts.engine.TTS_DATA_INSTALLED
```

B. Kernel log when loading snapshots

```
[    52.994027] BUG: unable to handle kernel NULL pointer dereference at
(null)
[    52.994027] IP: [<ffffffff8104afee>] __wake_up_common+0x20/0x79
[    52.994027] PGD 15135067 PUD 150f3067 PMD 0
[    52.994027] Oops: 0000 [#1] PREEMPT
[    52.994029] CPU: 0 PID: 0 Comm: swapper Not tainted 3.10.0+ #69
[    52.994046] Hardware name:  , BIOS QEMU 01/01/2007
[    52.994061] task: ffffffff81a16500 ti: ffffffff81a00000 task.ti: ffffffff81a00000
[    52.994077] RIP: 0010:[<ffffffff8104afee>]  [<ffffffff8104afee>]
                __wake_up_common+0x20/0x79
[    52.994105] RSP: 0018:ffffffff81a21df8  EFLAGS: 00010092
[    52.994120] RAX: 0000000000000003 RBX: 0000000000000086 RCX: 0000000000000000
[    52.994136] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff88001d724b28
[    52.994151] RBP: ffffffff81a21e38 R08: 0000000000000000 R09: 0000000000000008
[    52.994167] R10: 00000000ffffffff R11: 0000000000000246 R12: ffff88001d724b28
[    52.994287] R13: 0000000000000000 R14: ffff88001ebc81e0 R15: 0000000000000001
[    52.994306] FS:  0000000000000000(0000) GS:ffffffff81a1e000(0000)
                knlGS:0000000000000000
[    52.994322] CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
[    52.994338] CR2: 0000000000000000 CR3: 0000000015116000 CR4: 00000000000006b0
[    52.994355] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[    52.994372] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
[    52.994387] Stack:
[    52.994402]  0000000000000000 00000001810a32ea ffffffff81a39de0 0000000000000086
[    52.994462]  ffff88001d724b28 ffffffff81c3a370 ffff88001ebc81e0 ffffffff81a01fd8
```

```
[    52.994522]  ffffffff81a21e68 ffffffff8104df6b 0000000100000001 0000000000000000
[    52.994583] Call Trace:
[    52.994597]  <IRQ>
[    52.994610]  [<ffffffff8104df6b>] __wake_up+0x39/0x64
[    52.994653]  [<ffffffff81422142>] goldfish_pipe_interrupt+0x73/0xa9
[    52.994671]  [<ffffffff81075adf>] handle_irq_event_percpu+0x85/0x213
[    52.994688]  [<ffffffff810073e7>] ? native_sched_clock+0x3b/0x70
[    52.994705]  [<ffffffff81075cba>] handle_irq_event+0x4d/0x6b
[    52.994721]  [<ffffffff81077ce6>] handle_edge_irq+0xbf/0xf1
[    52.994745]  [<ffffffff81002df5>] handle_irq+0x1f/0x25
[    52.994761]  [<ffffffff81002ccc>] do_IRQ+0x42/0xa6
[    52.994778]  [<ffffffff815b1468>] common_interrupt+0x68/0x68
[    52.994808]  <EOI>
[    52.994821]  [<ffffffff81007f2e>] ? default_idle+0x48/0x100
[    52.994863]  [<ffffffff81007ef2>] ? default_idle+0xc/0x100
[    52.994879]  [<ffffffff81008625>] arch_cpu_idle+0x13/0x18
[    52.994896]  [<ffffffff810568d8>] cpu_startup_entry+0x14b/0x210
[    52.994913]  [<ffffffff815a85b6>] rest_init+0x7a/0x7e
[    52.994929]  [<ffffffff81ac7c61>] start_kernel+0x337/0x344
[    52.994945]  [<ffffffff81ac747d>] x86_64_start_reservations+0x2a/0x2c
[    52.994961]  [<ffffffff81ac7547>] x86_64_start_kernel+0xc8/0xcc
[    52.994976] Code: 48 89 47 10 eb db 5b 41 5c 5d c3 55 48 89 e5 41 57 41 89 d7
                    41 56 41 55 41 89 cd 41 54 49 89 fc 53 48 83 ec 18 48 8b 17
                    89 75 cc <48> 8b 1a 48 8d 42 e8 48 83 eb 18 48 8d 50 18 49
                    39 d4 74 36 4c
[    52.995739] RIP  [<ffffffff8104afee>] __wake_up_common+0x20/0x79
[    52.995766]  RSP <ffffffff81a21df8>
[    52.995780] CR2: 0000000000000000
[    52.995808] ---[ end trace 10161f3ce1f579a5 ]---
[    52.995823] Kernel panic - not syncing: Fatal exception in interrupt
```

C. Apks used in performance tests

```
codematics.universal.tv.remote.control_34.apk
com.aguirre.android.mycar.activity_246.apk
com.ancestry.android.apps.ancestry_103.apk
com.androidsx.heliumvideochanger.warping_42.apk
com.apalon.myclockfree_127.apk
com.apalon.weatherlive.free_129.apk
com.avast.android.batterysaver_2951.apk
com.bestringtonesapps.freeringtonesforandroid_47.apk
com.btows.photo_79.apk
com.crema.instant_10.apk
com.disneydigitalbooks.DisneyMagicBrushTimer_goo_25.apk
com.disney.frozensaga_goo_192.apk
com.dobsoftstudios.gunfustickman2_43.apk
com.duolingo_439.apk
com.ea.game.tetris2011_row_2022.apk
com.ebay.kleinanzeigen_5538.apk
com.endomondo.android_230.apk
com.fingersoft.hillclimb_129.apk
com.flightradar24free_66110.apk
com.flipkart.android_760800.apk
com.google.android.apps.youtube.gaming_19130023.apk
com.google.android.gm_58706527.apk
com.jb.gokeyboard.theme.goldgokeyboardtheme.getjar_100.apk
com.jb.gokeyboard.theme.keyboardthemetattoo.getjar_25.apk
com.kauf.talking.baum.TalkingMikeMouse_39.apk
com.kodakalaris.kodakmomentsapp_1701120146.apk
com.korrisoft.draw.your.game_455.apk
com.kpmoney.android_166.apk
com.linkedin.android_90300.apk
```

com.lt.latte.brick_29.apk
com.makeshop.podbbang_196.apk
com.melodis.midomiMusicIdentifier.freemium_20007.apk