Orchestrating Android Malware Experiments

Jean-François Lalande, Pierre Graux, Tomás Concepción Miranda

Orchestrating Android Malware Experiments

Jean-François Lalande, Pierre Graux and Tomás Concepción Miranda
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France

***Abstract*—Experimenting with Android malware requires to manipulate a large amount of samples and to chain multiple analyses. Scripting such a sequence of analyses on a large malware dataset becomes a challenge: the analysis has to handle fails on the computer and crashes on the used smartphone, in case of dynamic analyses. We present a new tool, PyMaO, for handling such experiments on a regular desktop PC with the highest performance throughput. PyMaO helps to write sequences of analyses and handle partial experiments that should be restarted after a crash or continued with new unknown analyses. The tool also offers a post processing capability for generating number tables or bar graphs from the analyzed datasets.**

I. Introduction

Researchers working on malware analysis usually performs benchmarks on large datasets of malware samples [1], [2]. These datasets can contain malicious applications, benign applications or both, and eventually some metadata that help to investigate the samples. When experimenting with these datasets, the samples are consumed by the software that perform a bunch of analysis. Analysis can be either static or dynamic. For example, a static analysis can open the APK file of the sample and perform an analysis of the resources and of the application bytecode. On the other hand, a dynamic analysis may execute the application in an emulator or real smartphone for extracting dynamic events. The obtained artifacts are then collected and can feed another step of the experiment, for example a machine learning tool.

The code for handling these analyses and collecting the results are much more complex than a loop. In the literature, Andrubis [1] was one of the most complex developed framework that handles multiple analyses on large volume of applications. They were able to achieve 3,500 analyses per day. Andlantis [2] is another solution using 200 computing nodes, with very high throughput and scalability. Most of the time, researchers that have developed software for coordinating multiple analyses, do not share the code [2], [1], except few tools, like the IRMA framework[1]. Additionally, this code have precise hardware requirements i.e. multiple physical servers that enables to have high performance throughput. These solutions cannot be used when prototyping experiments and are only useful for production time.

In this paper we present the PyMaO[2] (Python Malware Orchestrator), a tool for orchestrating an experiment on a regular PC desktop. An experiment is a sequence of analyses. Each analysis can depend on other ones and have pre-conditions authorizing its run. An experiment is robust to crashes of the tool (results are not lost) or to crashes of the smartphones that are used by dynamic analyses.

II. Solved challenges

Prototyping software for malware analysis requires to test methodologies with a dataset containing enough material for triggering special cases that can reveal some bugs. When manipulating a smartphone or an emulator, the execution of malicious software may damage the operating system, requiring to restore the used device. As a consequence, any error should be clearly reported to the user for further investigation. If our tool crashes, it should be able restart itself and to take into account that some results have been previously computed and should not be done again. In our tool, this is achieved using a JSON representation of the results of all analyses that is reloaded if the tool have to evaluate again an APK file.

Moreover, the tool brings simplicity when crafting new experiments and performances at execution time. This is achieved by coding each analysis as an independent class and by combining these analyses with pre-conditions authorizing the execution of each of them. At runtime, multiple workers use all the available CPU threads and manipulate the uncompressed data directly into the RAM memory.

III. Orchestrator architecture and usage

A. Overview

Figure 1 depicts an overview of the architecture of PyMaO. This tool receives a set of APKs and an experiment to conduct on each APK file. The experiment is composed by a sequence of multiple analyses. An analysis is launched, for a given APK, if all its dependencies have already been launched and if its associated conditions are met. An example of an experiment is given in Section III-B. All these requirements are handled by the producer which is in charge of dispatching the analyses' run to the workers. Each worker is executed on a separate thread. When a worker finishes an analysis, it updates the JSON file corresponding to the analyzed APK. Finally, all the JSONs are processed by the reporter. This part is described in Section III-C.

When performing a dynamic analysis, a worker can uses a real smartphone. To avoid concurrency issues, each worker has a dedicated device. Because running malware on a smartphone may crash the smartphone, a watchdog thread has been implemented, which continuously receives heartbeats sent by a homemade software installed on every phone. If the watchdog does not receive a heartbeat, it reboots and restores the crashed phone to an initial state.

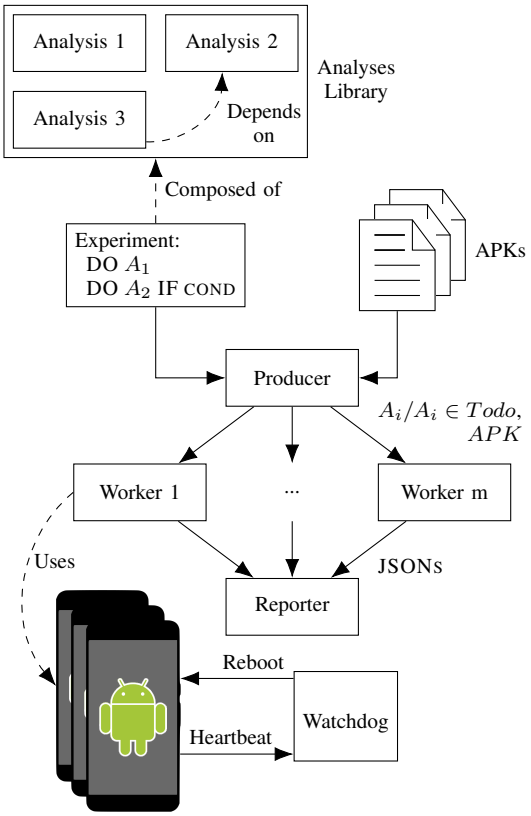

Fig. 1. Architecture overview

```
1 class XpExample(Experiment):
2   def appendAnalysis(self):
3     self.analyses.append((Apkid(self), None))
4     self.analyses.append((Packer(self),
5             [{"Apkid": {"status": "done"}}]))
6     self.analyses.append((AdbInstall(self),
7             [{"Packer" : {"packer": True}}]))
8     self.analyses.append((LaunchAndSurvive(self),
9             [{"AdbInstall" : {"install": True}}]))
```

Listing 1. Experiment example

B. Declaring chains of analysis

PyMaO has been developed with usability in mind. Thus, developing new experiments that use external tools is simple. Listing 1 contains all the required code, except few `import` statements, to create a fully working experiment. All the remaining information is contained in the `super` class `Experiment` (line 1). The created experiment first runs APKiD[3], a tool that detects packers artifacts (line 3). Then, if APKiD has correctly run (condition line 5), it parses its output using the `Packer` analysis (line 4). Finally, if the usage of a packer has been detected (line 7), the experiment installs (line 6). If this install succeeded (line 9), the tool launches the APK and checks its liveness (line 8). By dividing analyses in unitary operations, PyMaO avoids to copy past these functionalities for new designed experiments.

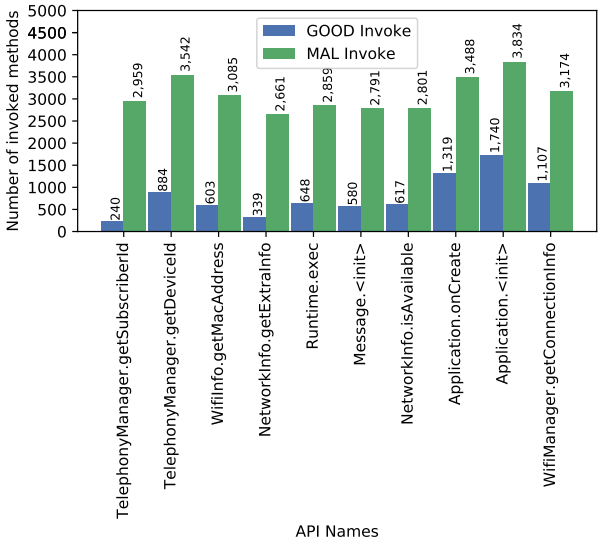

Fig. 2. Top 10 most used API methods by malware in relation to goodware [3]

C. Reporting results

After executing an experiment, the tool can extract an aggregated information from the JSON files. We developed two post processing scripts generating two types of results. First, the tool can generate tables for counting features in the JSON files and generate ratios. For example, the table can report the number of detected malware if the experiment is a detection algorithm. Second, the tool can draw charts for reporting the usage of a feature in the dataset. For example, Figure 2 is an extract of an experiment conducted previously on the GM19 dataset [3], that shows the top most used methods on 5000 goodware and 5000 malware.

IV. Future work

We continue to enhance the tool with new functionalities. For example, the resource usage (CPU, devices, memory) can be handled more accurately. By evaluating the profile of each analysis, the workers can choose the best task to achieve with the current remaining amount of resources available. Another future work concerns the generation of subsets of datasets with a descriptive language requesting the wanted features.

References

[1] M. Lindorfer and M. Neugschwandtner, "ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. San Jose, CA, USA: IEEE Computer Society, 2014.

[2] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe, "Andlantis: Large-scale android dynamic analysis," *CoRR*, vol. abs/1410.7751, 2014. [Online]. Available: http://arxiv.org/abs/1410.7751

[3] V. Viet Triem Tong, C. Herzog, T. Concepción Miranda, P. Graux, J.-F. Lalande, and P. Wilke, "Isolating malicious code in android malware in the wild," in *14th International Conference on Malicious and Unwanted Software*. Nantucket, MA, USA: IEEE Computer Society, 2019.

Notes

[1]https://github.com/quarkslab/irma
[2]https://gitlab.inria.fr/cidre-public/pymao
[3]https://github.com/rednaga/APKiD