MIME: A Formal Approach to (Android) Emulation Malware Analysis*

Fabio Bellini, Roberto Chiodi, and Isabella Mastroeni

Dipartimento di Informatica - Università di Verona - Italy
(fabio.bellini|roberto.chiodi)@studenti.univr.it
isabella.mastroeni@univr.it

Abstract. In this paper, we propose a new dynamic and configurable approach to anti-emulation malware analysis, aiming at improving transparency of existing analyses techniques. We test the effectiveness of existing widespread free analyzers and we observe that the main problem of these analyses is that they provide static and immutable values to the parameter used in anti-emulation tests. Our approach aims at overcoming these limitations by providing an abstract non-interference-based approach modeling the fact that parameters can be modified dynamically, and the corresponding executions compared.

Keywords: Anti-emulation malware, abstract non-interference, program analysis

1 Introduction

The recent technological escalation led to a massive diffusion of electronic devices with permanent Internet connection. One of the most widespread mobile OS is Android, that have reached more than 1 billion device activations in the last year, with an average of 1.5 million activations per day. By installing software coming from untrusted markets, a user may cause/introduce lots of vulnerabilities on his system, such as privilege escalation, remote control, financial charge and data leakage [18]. For instance, one in five Android users faces a mobile threat, and the half of them installs trojans designed to steal money.

The problem. In order to study malware payloads, it is necessary to analyze malicious software by using specific tools, based on emulation and virtualization, which statically and dynamically analyze the code. The problem is that some malware try to avoid these analyses by exploiting environment detection tricks allowing them to understand whether they are emulated or not. These techniques are called *anti-emulation checks* [6, 15]. If an anti-emulation check detects the presence of a virtual environment, the malware changes its behavior showing only harmless executions or simply aborting the computation. In [11] it has been proposed a method to automatize the creation of red pills, generating thousands

* This work is partly supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT.

of mnemonic opcodes that trigger different behaviors in real and emulated environments. Futhermore, lots of anti-emulation checks were find out for many different emulation environments like QEMU, Bochs and VMWare [4,8,13,14,16]. On the other hand, several tools were developed to reduce discrepancies between real and emulated environments, trying to obtain *perfect transparency* [3, 7, 17]. Recently, the security focus moved to the Android mobile environment, where virtualization on devices is inefficient and not widespread nowadays. There are currently some Android analyzers available that scan applications trying to extract their main features like permissions, services or used networks, for detecting malicious actions. The problem, is that also Android malware started to embed anti-emulation checks, making them resilient to analyses, and, in [12], it is shown how simple is to bypass analyzers by using trivial anti-emulation checks.

Our approach. We test 28 samples belonging to 15 different known malware families on 9 analysis frameworks available online, such as Andrubis and VirusTotal. We analyze the obtained results, providing a more specific perspective on the connection between the state-of-the-art of anti-emulation techniques and our samples sources. This work allows us to identify the limitations of the existing analyzers, such as the lack of versatility and customization, usually caused by the general trend to prefer better performance instead of stronger protection. Moreover, we observe that there are no formal frameworks allowing us to semantically understand the problem of anti-emulation. A semantic comprehension would allow us to compare different techniques, and to understand how we can tune our analysis in order to adapt it to different attacking scenarios. In the existing literature, the only attempt to formalize the notion of anti-emulation is given as an *interference* between the environment and the program execution [6]. The problem with this notion is that it is too strong, since benign applications may change behaviors depending on the environment. Hence, we propose a formal definition of anti-emulation, given in terms of abstract non-interference [5], a weakening of non-interference based on abstract interpretation [2], where both the property that can/cannot interfere, and the output observation are modeled as properties of the concrete behavior. This formal framework is used for better understanding how we can make *anti*-anti-emulation checks stronger depending on the platform we work on, allowing us to improve existing analysis tools and providing a first overview of an ideal analysis framework called *Multiple Investigation in Malware Emulation* (MIME).

2 Limitations of Existing Android Malware Analyses

We started the test phase analyzing the anti-emulation checks in Android well-known malware families. In our work, we consider: *BadNews*, *BaseBridge*, *BgServ*, *DroidDream Light*, *Droid KungFu* – 1, 2, 3, 4, Sapp, Update –, *FakeMart*, *Geinimi*, *Jifake*, *OBad* and *ZSone*. For each malware family, we chose two different variants to verify how frameworks react to small code differences that are not related to malware payload – only in Jifake and Droid KungFu Update this was not possible, because only one version was available. We submitted all these samples

to 9 different analyzers, free and available with Web interface: *AndroTotal Andrubis*, *APKScan*, *Dexter*, *ForeSafe*, *Mobile-SandBox*, *VirusImmune*, *VirusTotal* and *VisualThreat*. In our tests, we submitted samples which were statically and dynamically analyzed or scanned by a pool of anti-virus software: all the previous frameworks could cover one or more of these categories. By summarizing, we collected 252 different combinations malware-analyzer that are fully available in [1]. In order to avoid emulation, most of malware check several environment issues, such as constants in Android Build class and/or other information as IMEI[1], IMSI[2] and telephone sensors management. Thus, in order to verify the behavior and, consecutively, the presence of anti-emulation checks in those malware, we mainly need dynamic analysis: this means that the most complete results come from Andrubis, APKScan, ForeSafe and VirusTotal. Nevertheless, we observe that even these frameworks use trivial anti-emulation-related parameters, such as IMEI, IMSI, etc. Other kinds of malware anti-emulation checks are also in general used, as shown in the following example.

```
deviceId=android.provider.Settings.System.getString(context1.
    getContentResolver(),"android_id");
if (deviceId == null){
                deviceId = "Emulator";
}
```

Listing 1.1. `Geinimi` anti-emulation check inspecting the `android_id` value.

We can observe that, most of the actual frameworks do not provide the possibility to dynamically customize the configuration of the virtual machine (in the following denoted VM), making easy for a malware to detect the virtual environment. Finally, the analyzers we considered in the test phase, do not allow multiple execution in different virtual environments, but always provide the same configuration for the VM, hence, if different executions in different environments are required, it is necessary to manually upload the samples in several frameworks. All these limitations make the analysis of anti-emulation malware often imprecise (being detected) and/or expensive.

3 Formal definition of Anti-emulation

We show here that the existing notion of anti-emulation [6] is too strong for really capturing the problem, and we propose a model based on *abstract* non-interference. In non-interference we have some information to protect that has not to interfere with the observable information. In the anti-emulation field, the information to protect is the "kind" (virtual or not) of execution environment: when a malware uses anti-emulation techniques there is a flow of information from the "kind" of execution environment to the malware.

Abstract Non-interference Suppose the variables of program split in private (`H`) and public (`L`). Let ρ a property characterizing the *attacker observation*, while

[1] International Mobile Equipment Identity.
[2] International Mobile Subscriber Identity.

ϕ is the property stating what, of the private data, can flow to the output observation, also called *declassification*. A program P satisfies ANI if $\forall h_1, h_2 \in \mathtt{H}, \forall l \in \mathtt{L}$: $\phi(h_1) = \phi(h_2) \Rightarrow \rho([\![P]\!](h_1, l)) = \rho([\![P]\!](h_2, l))$. Whenever the attacker is able to observe the output property ρ, then it can observe nothing more than the property ϕ of the input [5, 9, 10].

Observational Semantics and formal definition of anti-emulation. We focus on Android applications which are written in Java and compiled to *Dalvik* bytecode, with the possibility to use a large part of the standard Java library. Let $\mathtt{App}$ be the set of Android applications. According to the definition in [6], we model the behavior of a program $\mathtt{A}$ as a function depending on the input memory $\sigma \in \mathtt{Mem}$ and on the environment $\mathtt{E} \in \mathtt{Env}$. An environment provides "all the aspects of the system the program might query or discover, including the other software installed, the characteristics of hardware, or the time of day" [6]. In order to describe the actions performed by a program we consider a set of *Events* $\mathscr{E}$ describing the relevant actions performed by the application during its execution. Let $\Sigma = \mathscr{E} \times \mathtt{Mem}$ be the set of program states then, the observational semantics is: $[\![\cdot]\!] : \mathtt{App} \longrightarrow (\mathtt{Env} \times \mathtt{Mem} \to \wp(\Sigma^*))$. Given $\mathtt{A} \in \mathtt{App}$, $[\![\mathtt{A}]\!]$ is a function providing the trace of events executed by the program $\mathtt{A}$ depending on an hosting environment $\mathtt{E}$ and on an initial input σ. The only formal characterization of anti-emulation [6] says that $\mathtt{P}$ uses *anti-emulation techniques* if its execution under a real environment $\mathtt{E}_r$ changes its behavior under an emulated environment $\mathtt{E}_e$, although input σ is the same and environments are very similar: $[\![\mathtt{P}]\!](\mathtt{E}_e, \sigma) \neq [\![\mathtt{P}]\!](\mathtt{E}_r, \sigma)$. In this case, $\neq$ denotes the fact that the two executions are "observationally" different.

The ANI-based approach. Our approach, is based on what we observe of the program executions, i.e., on the granularity of the set $\mathscr{E}$ modeling the events, namely the actions, considered suspicious. In particular, we have to identify the set $\mathscr{M} \subseteq \mathscr{E}$ of the suspicious/malicious events. We denote $\mathscr{B} \stackrel{\text{def}}{=} \mathscr{E} \setminus \mathscr{M}$ the set of all the supposed benign events. The set $\mathscr{M}$ can be extracted depending on known information regarding the sample that we want to analyze. For instance, when considering applications handling images could be *malicious* the action of sending an SMS, while this action is perfectly acceptable for instant messaging applications. Alternatively, we can use the common payloads investigated in [18].

Let $\mathbb{E} \subseteq \mathtt{Env}$ the admitted range of variation for (virtual and real) environments, $\sigma \in \mathtt{Mem}$ an input for the application $\mathtt{A}$, let $X \subseteq \Sigma^*$ and $X_{|\mathbb{E}} \subseteq \mathscr{E}$ the set of events in the traces in X

$$Sel_{\mathbb{E}}(\langle \mathtt{E}, \sigma \rangle) = \begin{cases} \langle \mathbb{E}, \sigma \rangle \text{ if } \mathtt{E} \in \mathbb{E} \\ \langle \mathtt{E}, \sigma \rangle \text{ otherwise} \end{cases} \qquad Obs_{\mathscr{M}}(X) = \begin{cases} \mathtt{True} \text{ if } X \cap \mathscr{M} \neq \emptyset \\ \mathtt{False} \text{ otherwise} \end{cases}$$

$Sel_{\mathbb{E}}$ decides whether an environment is in the considered range $\mathbb{E}$ or not, while $Obs_{\mathscr{M}}$ checks whether the application shows only benign behaviors or even malicious ones.

The core of the abstract non-interference-based model of anti-emulation is the flow of information between the execution environment and the application behavior. Our definition strongly depends on the set $\mathscr{M}$. Thus, fixed the set $\mathscr{M}$

and given the abstractions $Sel_{\mathbb{E}}$ and $Obs_{\mathscr{M}}$ defined above, an application $\mathtt{A}$ does not use *anti-emulation* with respect to $\mathbb{E}$ and $\mathscr{M}$ if:

$$\begin{aligned}\forall\langle \mathtt{E}_1, \sigma_1\rangle, \langle \mathtt{E}_2, \sigma_2\rangle \in \mathtt{Env} \times \mathtt{Mem}.\ Sel_{\mathbb{E}}(\langle \mathtt{E}_1, \sigma_1\rangle) = Sel_{\mathbb{E}}(\langle \mathtt{E}_2, \sigma_2\rangle) \Rightarrow \\ Obs_{\mathscr{M}}([\![\mathtt{A}]\!](\langle \mathtt{E}_1, \sigma_1\rangle)) = Obs_{\mathscr{M}}([\![\mathtt{A}]\!](\langle \mathtt{E}_2, \sigma_2\rangle))\end{aligned} \tag{1}$$

The malware does not contain any anti-emulation check, with respect to the environment range $\mathbb{E}$ and the malicious actions set $\mathscr{M}$, if by varying the environment inside $\mathbb{E}$ we observe only malicious actions (the malware does not contain anti-emulation checks) or only benign actions (the model does not capture the correct anti-emulation checks of the malware analyzed).

4 Multiple Investigation in Malware Emulation (MIME)

We propose an analyzer architecture based in a configurable VM called MIME – Multiple Investigation in Malware Emulation. Existing analyses use an environment setting which is static and immutable, in the sense that they cannot be configured depending on the application contexts. Moreover, from our empirical studies, we observed that the problem of existing analyses is that parameters are set to fixed, and often trivial, values. This means that a truly transparent analyzer should be able to provide values that the malware expects from a real environment. Unfortunately, it is quite unrealistic to find a value robust to different malware. The idea we propose is to consider the ANI definition of anti-emulation, where we look for anti-emulation checks by analyzing malware several times, and each time under different environment settings. In other words, we let the input environment to vary, by setting a list of anti-emulation parameters (such as IMEI, IMSI, ...), and we observe how the corresponding execution is affected. Our goal is the automation of this formal model by making systematic the variation of the environment setting and automatic the corresponding executions and comparisons. The main idea of MIME is to perform several executions depending a configurable environment, which is systematically modified until we find an anti-emulation check (detected by an evaluation function) or until we exhaustively explore the space of environments we consider. In Figure 1, we show the proposed analysis architecture.

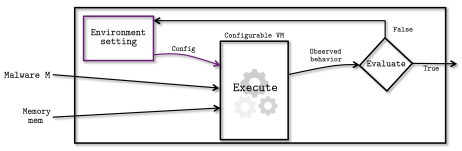

Fig. 1. Simple architecture of a MIME customizable analyzer.

Configuring MIME architectures. We choose a pool of n anti-emulation-connected input parameters. We associate with each parameter a list of prearranged values, in order to automatize the environment setting. It is worth noting, that both the set of parameters and the corresponding values can be easily customized. In

the MIME strategy, we represent these parameters like *rotors*, and each value corresponds to a position: by changing only one position at a time, we can detect which are the malware reactions. In general, let $\mathtt{p}_i$ the i-th parameter, let $\mathtt{R}_i$ be the rotor for $\mathtt{p}_i$, and let $\mathtt{V}_i$ be the set of values for $\mathtt{p}_i$. This idea, in our formal model corresponds to consider a set $\mathbb{E}_i$ for each parameter $\mathtt{p}_i$, which simply corresponds to the execution environment where the parameter p_i is set to one of the values in $\mathtt{V}_i$, while all the other parameters are set to a default/trivial initial value (for IMEI it may be all 0s). This choice is necessary since we aim at understanding precisely whether the parameter p_i, and not others, may affect execution. In this case, we consider Eq. 1, with input abstraction $Sel_{\mathbb{E}_i}$. Namely, the MIME strategy proposes to verify n times the Eq. 1, each time with respect one parameter $\mathtt{p}_i$, i.e., with respect to $\mathbb{E}_i$. Now, the evaluation function returns a boolean value identifying the presence of at least one malicious action in $\mathscr{M}$.

Using MIME for analyzing anti-emulation malware. Let us explain how the MIME strategy aims at improving the transparency of existing analyses. It is clear that the price to pay is in the performance of the analysis. Let `Rotors` be the matrix of all rotors, each rotor i has a number of possible different values $\mathtt{max}_i$. Suppose to have $n+1$ rotors, then we define `max` as maximal value in $\left\{ \mathtt{max}_i \,\middle|\, i \in [0,n] \right\}$. Let `M` be the analyzed malware, and `mem` the machine state memory. `Config` is the array containing the $n+1$ positions of the rotors used for setting the environment. `Execute` corresponds to the observation of the malware execution, formally it corresponds to the observed semantics $[\![\cdot]\!]$ where `Config` corresponds precisely to its environment input. `Evaluate` corresponds to the evaluation function that, as we noticed before, is modeled by the output abstraction $Obs_{\mathscr{M}}$. Finally, `position_anti_emu` is the array of the rotor positions where an anti-emulation check is detected by the evaluation. Listing 1.2

```
Function: Mime
Input: M, Rotors[n+1][max],mem
result = null;
for each k in [0,n] Config[k]=0; //Environment reset (rotors initial pos)
trace = Execute(M,Config,mem);
val_0=Evaluate(trace);
for each k in [0,n] Config[k]=max_k; //Environment setting to max rotors pos
trace = Execute(M,Config,mem);
val_m = Evaluate(trace);
if (val_0 != val_m) result = to_string(Catching_anti_emu(M, Rotors,mem));
if (val_0 == val_m)
        if (val_0) result = malware; // M is a malware without anti-emulation
        else result = pot_not_malware; //M is not recognized as malware
Output: result
```

Listing 1.2. MIME approach in anti-emulation malware recognition.

implements the idea that we first compute the default configuration of rotor parameters, whose evaluation `val_0` is `true` if a malicious action in $\mathscr{M}$ is observed. Then we execute the malware in the smarter configuration possible in our model, where the rotors are set to the maximal values, and we evaluate in `val_m` whether the corresponding execution exhibits malicious events. If the two values are different it means that somewhere in the code, there is an anti-emulation check, and therefore we launch the function `Catching_anti_emu`. If

the two values are both equal to `true` then it means that the malware does not use anti-emulation checks. Finally, if the values are both equal to `false`, and we knew that it was a malware, it means that it uses an anti-emulation check based on different parameters or it uses a conjunctive combination of tests on the known parameters. We lose this situation since we check parameters one at time. However, to the best of our knowledge, based on the experimental results, this last situation is not used by anti-emulation checks. Listing 1.3 shows

```
Function: Catching_anti_emu
Input: M, Rotors[n+1][max],mem
position_anti_emu[n] = null;
for each i in [0,n] {
        for each k in [0,n] Config[k]=0; //Environment reset
        for each j in [0, max_i-1]{
                Config[j]=Rotors[i,j];
                trace = Execute(M,Config,mem);
                val = Evaluate (trace);
                if (val) position_anti_emu[i] = j; break;
                else     j = j+1;
        }
        i = i+1; //change rotor
}
Output: position_anti_emu
```

Listing 1.3. Catching anti-emulation checks in MIME.

`Catching_anti_emu`. In this case, we have to vary the environment setting looking for the parameters responsible of the anti-emulation check. Since we aim at observing the interference of each single parameter on the malware behavior, we let only one rotor at time to change value, while all the others are set to default initial values. For this reason, each time we finish to analyze the interference of one rotor, we reset it before changing rotor. We always check all the rotors since there may be more than one anti-emulation check. `val` becomes `true` if some malicious action is detected, at this point since we know that there was also an harmless execution (the routine is called only in this case), it means that the current rotor contains an anti-emulation check in the current position, that we store in `position_anti_emu`. At the end of the execution, this vector contains all the rotor positions, namely the parameters values, used in anti-emulation checks. If at the end this vector is all `null`, then it means that the anti-emulation checks involve unobserved parameters or are based on different techniques.

5 Future Works

We develop our approach in the Android word however, our approach to anti-emulation can be easly generalized to any platform. In this case, it is necessary to change the `Evironment setting` and the VM, in order to let it analyze desktop malware. Until now, no implementation of our approach has been made, so a possible future implementation of MIME will be useful to successfully analyze anti-emulation malware. This would help also in understanding the scalability problem. Finally, our ANI model of anti-emulation is strongly related to the definition of the $\mathscr{M}$ set of the malicious events. We would like to improve this

model and study in depth the relation between the $\mathscr{M}$ set and the anti-emulation checks detection in malware, characterizing the amount of both false positives and negatives.

References

1. F. Bellini, R. Chiodi, and I. Mastroeni. Mime: A formal approach for multiple investigation in (android) malware emulation analysis. Technical Report RR 97/2015, 2015. `http://hdl.handle.net/11562/926789`.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of POPL '77*, pages 238–252. ACM, 1977.
3. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc. of CCS '08*, pages 51–62. ACM, 2008.
4. P. Ferrie. Attacks on virtual machine emulators. Symantec Corporation, 2007.
5. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of POPL '04*, pages 186–197. ACM, 2004.
6. M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proc. of VMSec '09*, pages 11–22. ACM, 2009.
7. M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting environment-sensitive malware. In *Proc. of RAID'11*, pages 338–357. Springer-Verlag, 2011.
8. T. Liston and E. Skoudis. On the cutting edge: Thwarting virtual machine detection, 2006.
9. I. Mastroeni. On the rôle of abstract non-interference in language-based security. In *Proc. APLAS '05*, LNCS 3780, pages 418–433. Springer-Verlag, 2005.
10. I. Mastroeni. Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his 60th Birthday.*, pages 41–65, 2013.
11. R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proc. of WOOT '09*, page 2. USENIX Association, 2009.
12. T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proc. of EuroSec '14*, pages 5:1–5:6. ACM, 2014.
13. D. Quist, V. Smith. Detecting the presence of virtual machines using the local data table. *Offensive Computing*, 2006.
14. T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Proc. of ISC '07*, pages 1–18. Springer-Verlag, 2007.
15. Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction, 2004.
16. K. Vishnani, A. R. Pais, and R. Mohandas. Detecting & defeating split personality malware. In *Proc. of SECURWARE 2011*, pages 7–13, 2011.
17. L. K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2e: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. *SIGPLAN Not.*, 47(7):227–238, March 2012.
18. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of SP '12*, pages 95–109. IEEE Computer Society, 2012.