*** SOFTWARE SECURITY***

***Professor: Corrado Aaron Visaggio***

***Assistant: Francesco Mercaldo***

***Malwares vs. Antimalwares****:*

*Evaluating malwares obfuscation techniques against antimalware detection algorithms*

***Authors****:*

*Meninno Michele: michele.meninno@gmail.com*

*Esposito Raffaele: e\_raffaele@alice.it*

*Battista Pasquale: pas.battista@gmail.com*

*De Lucia Agostino: agostino.delucia@hotmail.it*

**SUMMARY**

* **Purpose of the study**
* ***Let’s talk about android application and smali***
* ***Trivial transformations***
* ***Framework engine***
* ***Database***
* ***Results Overview***
* ***Results summary***
* ***Improvements proposals***
* ***References***

**PURPOSE OF THE STUDY**

Every day more than 1 million new Android devices are activated worldwide. This trend has made “android” the largest installed base of any mobile platform. Google Play is the premier marketplace for selling and distributing Android apps and it shows incredible numbers: 1.5 billion downloads a month! Everyone agrees that “android” is an incredible success, but are people sure to store their data on their android devices? This success has seen an always-growing android malware author interest.

Google Play is full of well-known free and paid anti-malware.

Are their detection algorithms effective when malware authors try to obfuscate malicious behaviors? To answer to this question we applied several common transformations to a data-set of android malwares. This data-set contains malwares of any android-malware family and is composed of 5560 malwares.

To apply these transformations we developed this framework *Alan,* it’s an open-source project, and its structure tries to help any possible future improvement.

The framework allows selecting transformations we want to apply.

**We applied our transformations to the entire original malware set and then submitted every single malware both in its original version and the transformed one on the website** [**www.virustotal.com**](http://www.virustotal.com) **(with his specific Java Api).** This website analyzes every malicious application with a set of widespread anti-malwares and for each of them they decide whether they are malicious or not.

We collected results applying transformations, all-combined together, in order to study how anti-malware detection algorithms perform.

**LET’S TALK ABOUT ANDROID APPLICATIONS AND SMALI**

Android applications are written in Java using Android software development kit. Instead of running Java bytecode Android runs *Dalvik* byte code stored in a .*dex* file.

Android is based on *linux kernel* and it would have been useless to apply transformations to the java code (which is often obfuscated). In order to work on a low-level code, it can be used a tool for reverse engineering called *apktool* whichallowsto decompile and recompile android applications.

*Apktool* can decode resources to nearly original form and rebuild them after making some modifications. The most important directory that we obtain using *apktool* is the *smali* directory. *Smali* is a human readable dalvik bytecode that has been our transformation target.

Using android sdk, *Apktool* is able to rebuild the applications after our modifications and helps to be sure about the correctness of the apk(several errors will be displayed if *smali* syntax is not correct.)

**TRIVIAL TRANSFORMATIONS**

First of all, two trivial transformations are applied simply using two tools *(Signapk*, *Apktool)*, they are called: *Disassembling & Reassembling* and *Repacking*. These two transformations are executed every time we use the framework, even without selecting any other transformation.

1. ***Disassembling & Reassembling:***

Mac:Users:Mik:Dropbox:PasqRaffMik:Alan2.png

This transformation is based on the *apktool* representation of the items contained in the *.dex* file. Disassembling an application, the command “*apktool d apkname”* creates several directories representing the original application resources: code, android manifest, etc. The command ”*apktool b apkDirectory”* creates an application based on the new apktool *dex* file representation.

**2) *Repacking***

Every android application has got a developer signature key that will be lost after disassembling the application and then reassembling it. To create a new key we used the tool *signapk* to avoid detection signatures that match the developer keys or a checksum of the entire application package.

***3) Changing package name***

This transformation set the application package name with a fixed-string, defined before transformation run.

**TRANSFORMATIONS DETECTABLE BY STATIC ANALYSIS**

In this work we used transformations that can be detected by static analysis.

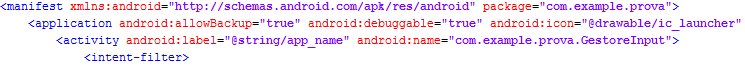
The majority of existent anti-malwares are based on these detection techniques that can be selected and combined on the framework with a check-box list.

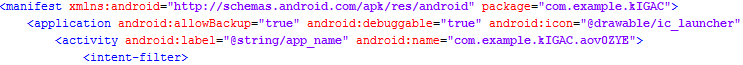
The execution order of the transformations is defined like a chain with removable elements.

***1) Identifier Renaming:***

The goal of this transformation is to rename every smali identifier (classes name, packages name, methods name, variables name etc…).

In this case the transformation can change package name and classes identifier, for each smali file, using a random string generator, handling calls in external classes to the modified classes.

Android manifest *Pre-transformation*

*Android manifest Post-transformation*

*Class name Pre-transformation*

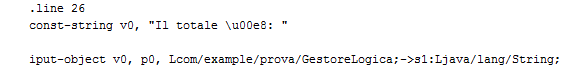


*Class name Post-trasnformation*

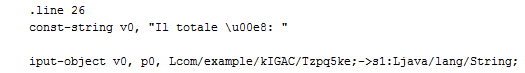




*Class call and package name Pre-Trasformation*



*Class call and package name Post-transformation*



***2) Data Encoding:***

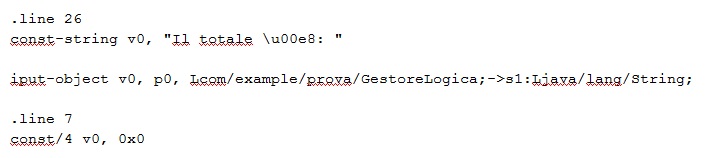
Strings can be used to create signatures that identify malwares.

This transformation encodes strings with a *Caesar cipher*.

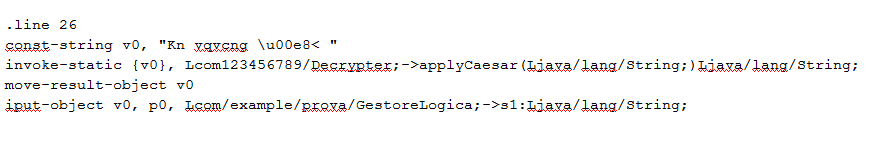
The original string will be restored, during application run-time, with a call to a *smali* function that knows the *Caeser key*.

This function has been created from a java class inserted into an *android project* and then the *smali* has been obtained thanks to *apktool* disassembling function.

*Pre-transformation string*

**

*Post-transformation string*

**

***N.B.***

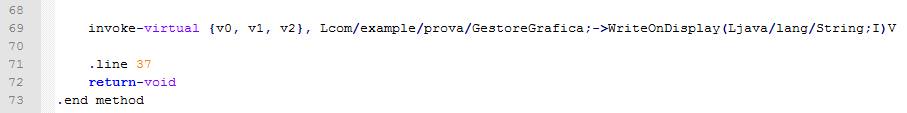
*Special characters won’t be changed since if they are obfuscated they will cause run-time problems.*

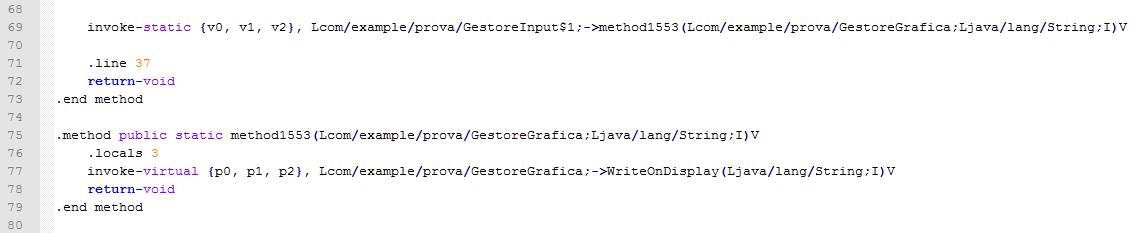
***3) Call indirections:***

This transformation modifies the application call graph.

Into the *smali* code every call is changed with a call to a new method inserted by the transformation. This new method calls the original method saving the right execution order.

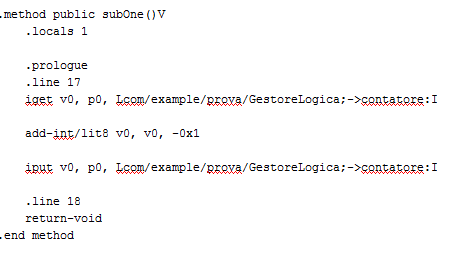
The transformation can be applied to every kind of call, in this case has been applied to every void *smali* method invoked with the “invoke-virtual” construct.

*Pre-transformation*

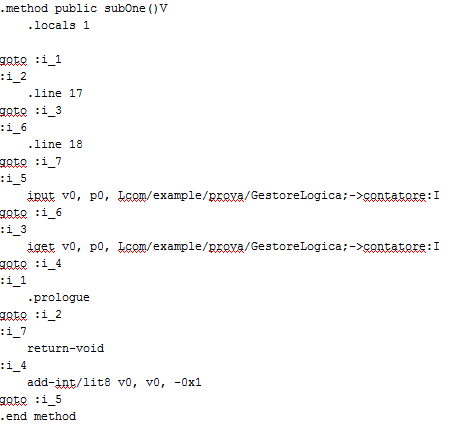
*Pre-transformation*

***4) Code Reordering:***

The aim of this transformation is to reorder *smali* methods inserting *goto* instructions in order to save the correct runtime execution. Every method has been changed with a new method where every instruction has been moved to a random index within the method body. The transformation has been applied only to methods that don’t contain any type of jump (if, switch, recursive calls).

*Pre-transformation*

*Post-transformation*



**N.B.**When in the *smali* code there is an invoke instruction it mustn’t be split from the following move instructions. Otherwise, the *smali* syntax will be compromised. They are always paired together in the *smali* code, even after the transformation.

***5) Junk Code Insertion:***

This transformation provides three different junk code insertions, which can be used stand-alone or combined:

* 1) Insertion of nop instructions at the beginning of each method.
* 2) Insertion of nop instructions and unconditional jumps at the beginning of each method.
* 3) Allocation of three additional registers on which performing garbage operations:

***e.g.***

*const/4 V,L*

*move V,V*

*if-eqz V,:T*

*:T*

*if-ltz V,:T*

*:T*

*if-eq V,V,:T*

*:T*

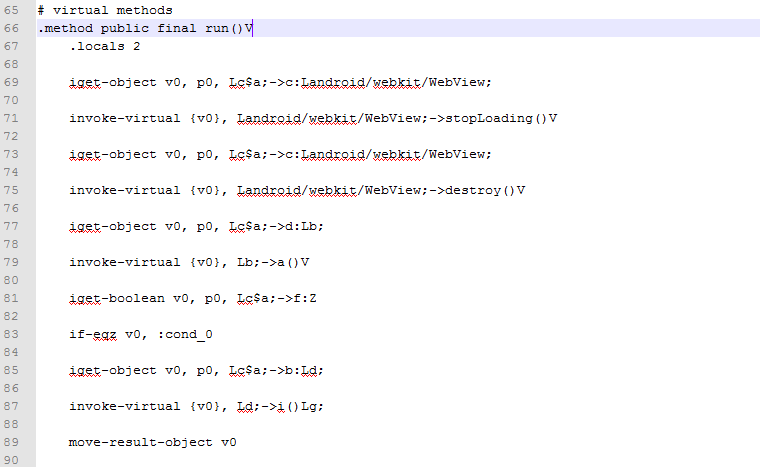
*if-le V,V,:T*

*:T*

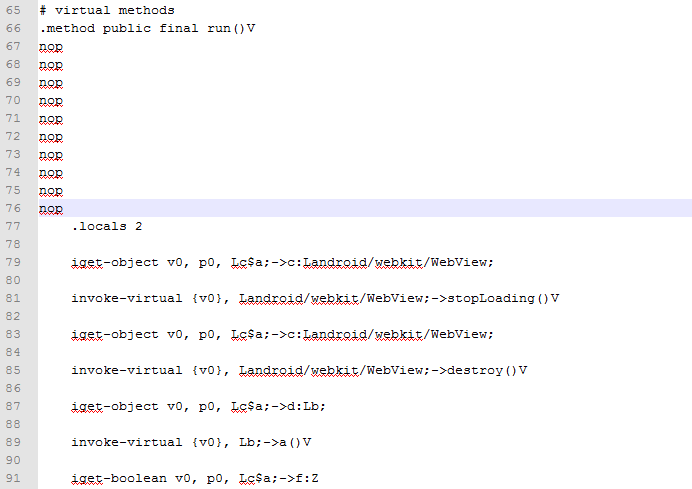
***N.B.***

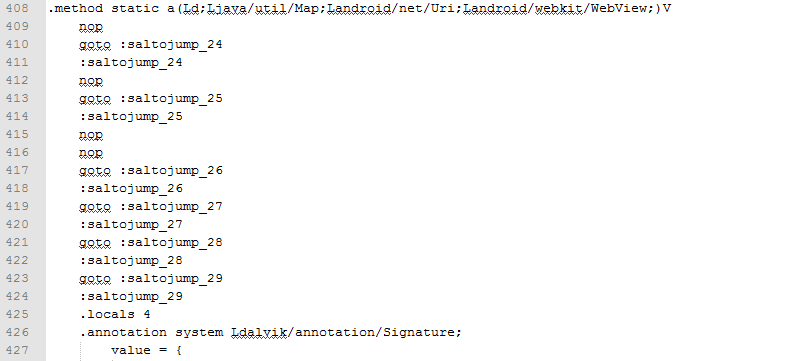
This type of junk code insertion is inserted at most once for each method divided in two blocks. The first block is placed soon after registers allocation and the second one is placed before the first method invocation. If the target method doesn’t contain any method invocation only the first block will be inserted.

*This is smali method before inserting any junk code.*

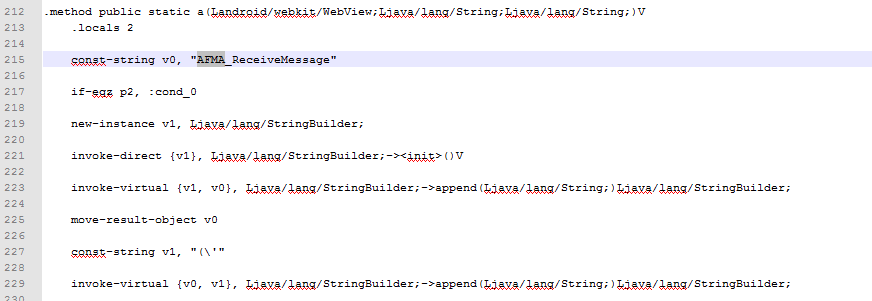


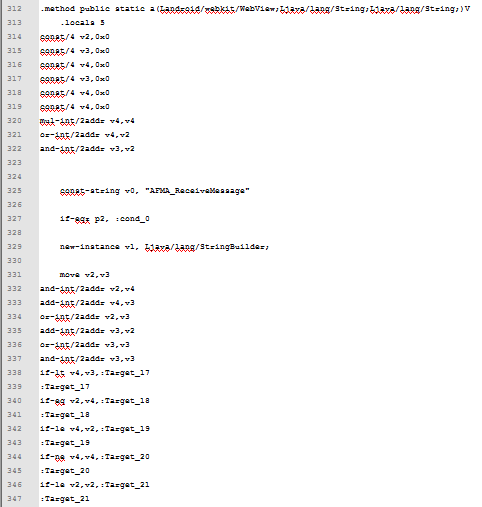
*This is the same method after nop insertion. (Type 1)*



*This is a method with nop and unconditional junk instructions (Type 2)*

*The two following screens show the same method before and after junk code insertion of Type* *3*





***6) Composite Transformations:***

All the transformations can be combined together without any problem.

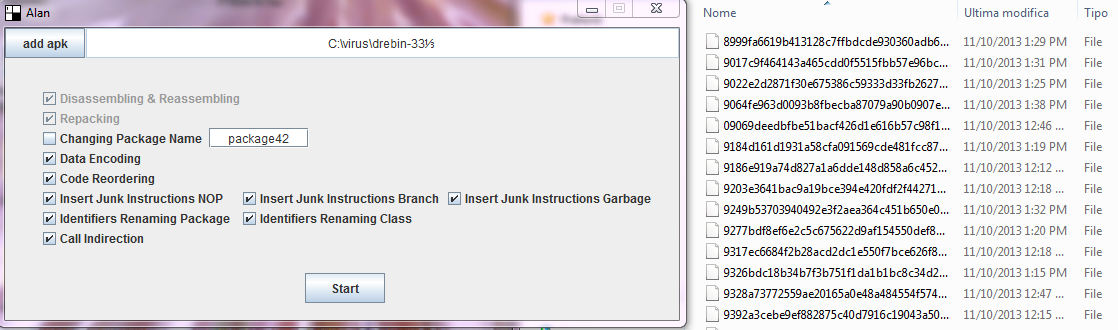
**FRAMEWORK ENGINE**

We developed this simple interface (coded in Java) that can be used to select one application or a directory containing several applications.

The engine starts when the “*Start*” button is pressed.

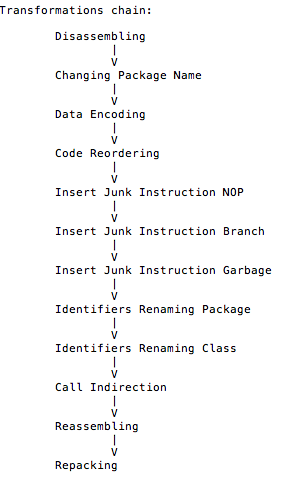
****

Framework UI 1: in this case a single apk has been selected

****

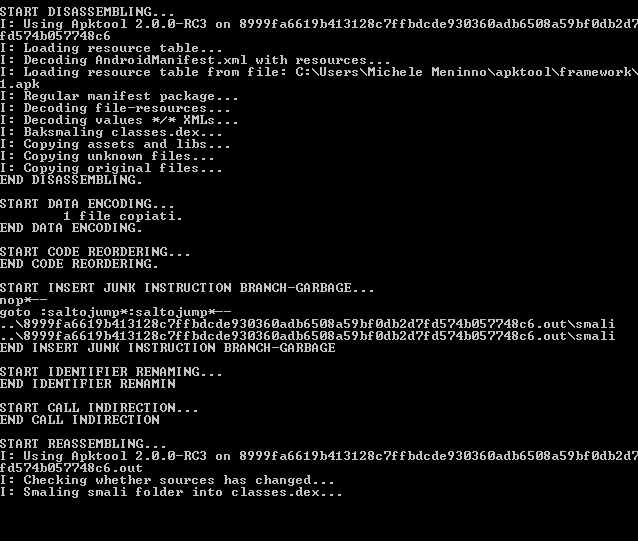
Framework UI 2: in this case a directory has been selected

We used a specific execution order to avoid conflicts when transformations are combined together. In this chain if a transformation it’s not selected it will be skipped.

****

Transformations chain

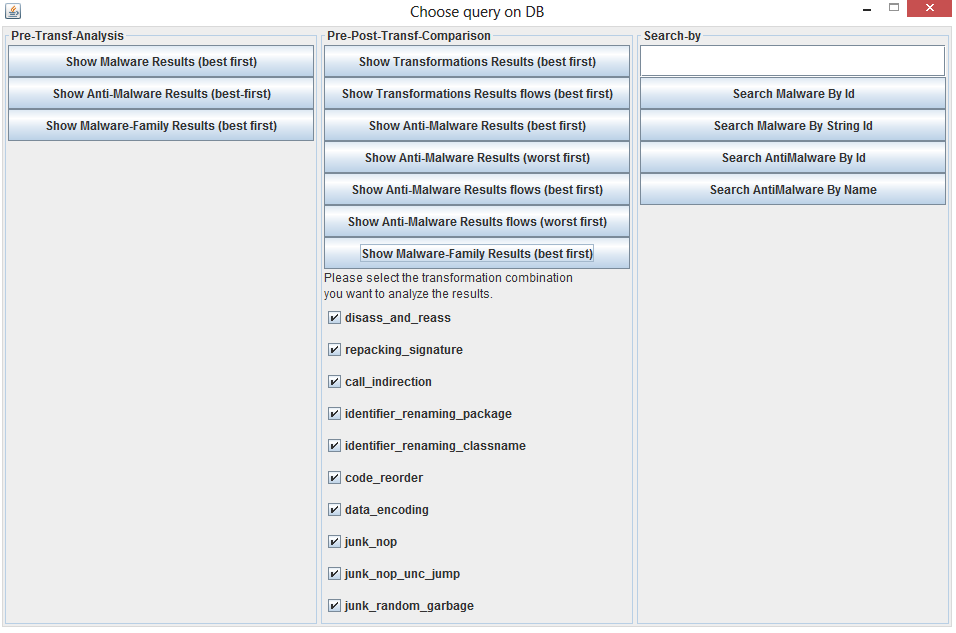
Once the start button is pressed a windows batch script will be created and executed, storing the selected applications into a specific directory.

****

**DATABASE**

In order to collect data coming from the analysis of thousands of malwares, we developed a simple SQL database (*MySql*).

We build up a UI based tool (coded in java) enabling to easily make interesting queries to the DB.



DataBase UI

Query results are presented in HTML files.

Our DB stores simple fine-grained data organized in 5 tables:

1. *malware*

This table stores malwares data, the core subject of our analysis.

1. *anti-malware*

This table stores anti-malwares data provided by *VirusTotal*.

1. *malware-family*

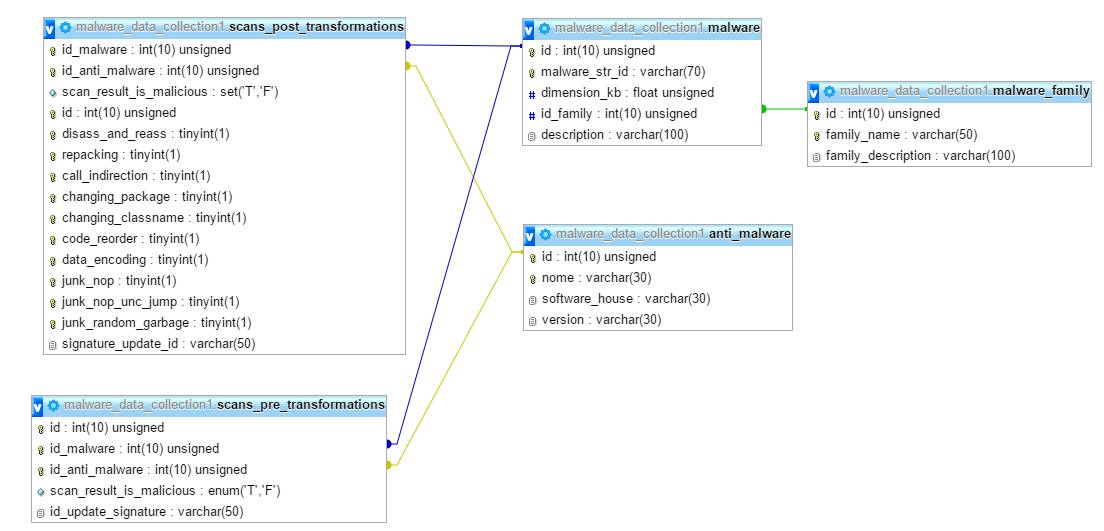
This table stores malware-family information.

1. *scans-pre-transformations*

This table stores *virusTotal* scan results of not-transformed malwares.

1. scans-post-transformations

This table stores *virusTotal* scan results of transformed malwares.

**RESULTS OVERVIEW**

dataBase 1: database schema

We worked on a *data-set*, composed of **5560** malwares belonging to **178** different malware families.

We applied all the transformations combined together on the entire *data-set.* During our transformation process we lost **794** malwares mainly due to *apktool* failing to decompile or recompile the application. A smaller part of the loss was caused by our transformations since on so large malware dataset sometimes *smali* is too complex or simply conflicting with transformation algorithms.

**Anti-malwares results (best first)**

As expected, the majority of anti-malwares find less malicious applications on the transformed *data-set* then on the original *data-set*. We collected results into an

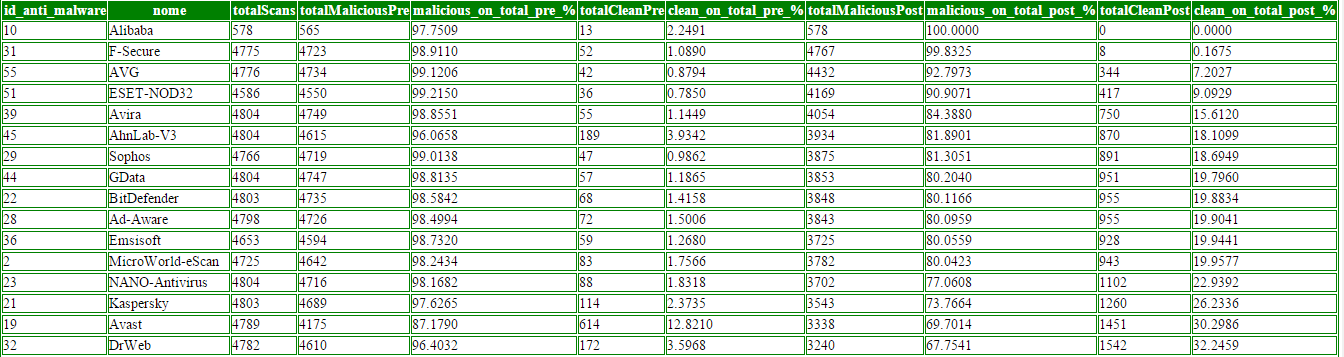
html file, link table: [Anti-malwares result (best first)](https://www.dropbox.com/s/54lqd43qayvn84r/ResultsComparisonAntimalware_bf.html?dl=0)

Table 1: anti-malware results previes (best first)

**totalScans**: the number of malwares analyzed by an antimalware both before and after transformations are applied.

**totalMaliciousPre**: the number of malwares that before being transformed are considerated malicious.

**totalMaliciousPost**: the number of malwares that after being transformed are considered malicious.

**totalCleanPre**: : the number of malwares that before being trasformed are considered clean.

**totalCleanPost**: the number of malwares that after being trasformed are considerated clean.

**The remaining fields** show the percentage ratio between the values of the aforementioned fields and totalScans.

**This table** is ordered by descending values of the field **malicious\_on\_total\_post**.

**Anti-malwares results flows (best first)**

Link table: [**Anti-malwares results flows (best first)**](https://www.dropbox.com/s/3to5z72v4wyp6j1/ResultsComparisonAntimalware_bf_flows.html?dl=0)

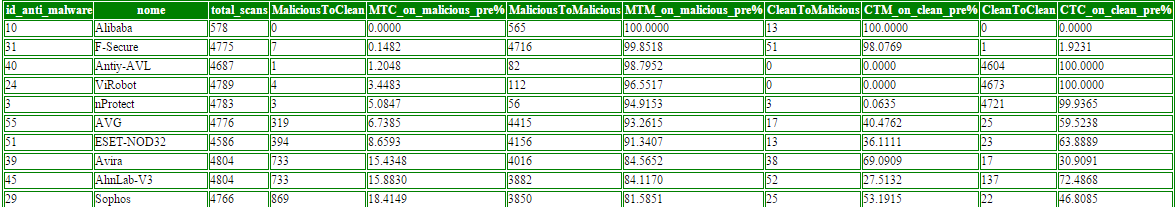


Table 2: anti-malware results flows preview(best first)

This table basically shows the same kind of information shown in **table1** using an higher level of detail, for each antimalware the following info are shown :

**totalScans**: The number of malwares analyzed by the antimalware both before and after transformations are applied.

**MaliciousToClean** : The number of malwares judged as malicious before transformations are applied and clean after their application.

**MaliciousToMalicious:** The number of malwares judged as malicious before transformations are applied as well as after their application.

**CleanToMalicious:** The number of malwares judged as clean before transformations are applied and malicious after their application.

**CleanToClean:** The number of malwares judged as clean before transformations are applied as well as after their application.

**This table** is ordered by descending values of the field **MTM\_on\_malicious\_pre**.

The two following tables show the same kind of information of the two tables presented, yet antimalwares with bad performance are shown first in the ordering used.

**Anti-malwares results (worst first)**

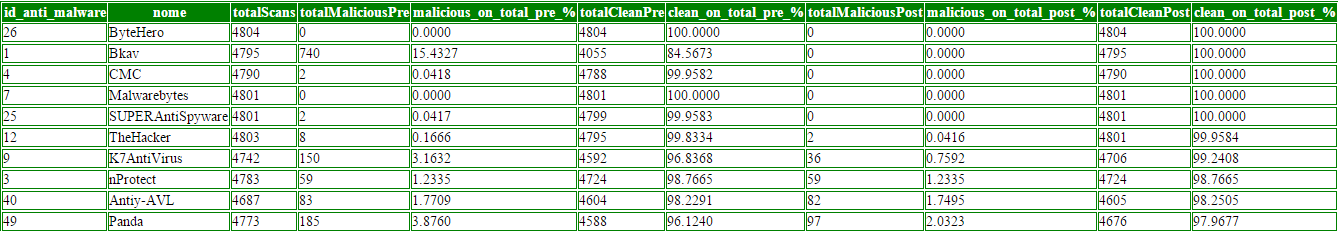
****Link table: [**Anti-Malwares result (worst first)**](https://www.dropbox.com/s/h76q4fduetjwky9/ResultsComparisonAntimalware_wf.html?dl=0)

Table 3: anti-malware results preview(worst first)

**Anti-malwares results flows (worst first)**

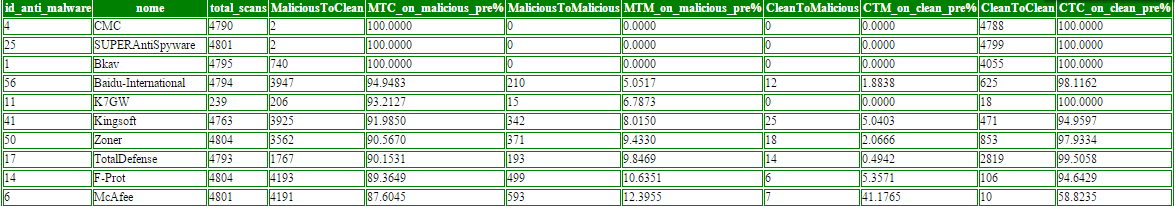
****Link table: [**Anti-malwares results flows (worst first)**](https://www.dropbox.com/s/vji87ace3hlm2yb/ResultsComparisonAntimalware_wf_flows.html?dl=0)

Table 4: anti-malware results flows preview (worst first)

**Malware results (best first)**

Link table: [**Malwares results (best first)**](https://www.dropbox.com/s/k1vctjifl2blkix/ResultsComparisonMalware_bf.html?dl=0)

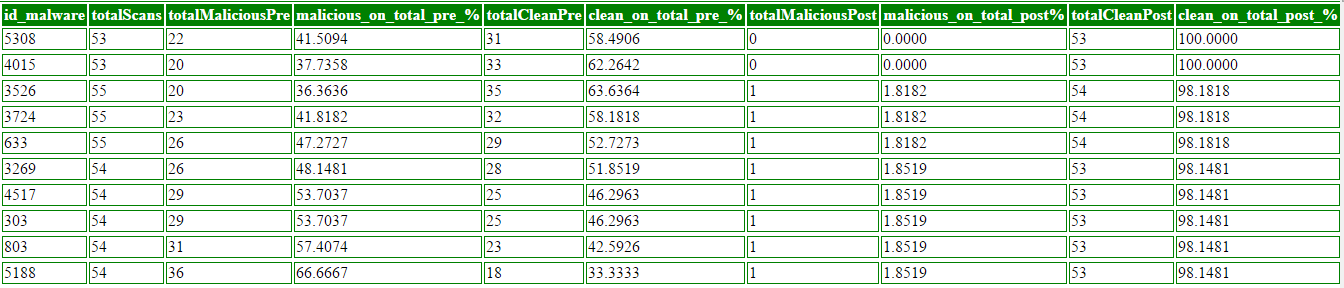
The following table shows malwares performance against antimalwares scans**.**

Table 5: malware results preview (best first)

**totalScans**: the number of antimalwares that have analyzed the malware both before transformations are applied and after (transformations applied).

**totalMaliciousPre**: the number of antimalwares that consider the malware malicious before being transformed.

**totalMaliciousPost**: the number of antimalwares that consider the malware malicious after being transformed.

**totalCleanPre**: the number of antimalwares that consider the malware clean before being transformed.

**totalCleanPost**: the number of antimalwares that consider the malware clean after being transformed.

**The remaining fields** show the percentage ratio between the values of the aforementioned fields and totalScans.

**This table** is ordered by descending values of the field **clean\_on\_total\_post**.

**Malware results flows (best first)**

Link table: [**Malware results flows (best first)**](https://www.dropbox.com/s/eeixgny1493wukk/ResultsComparisonMalware_flows_bf.html?dl=0)

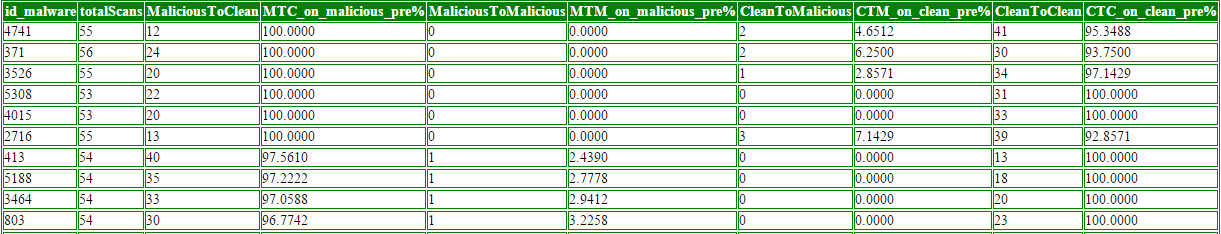
****

Table 6: malware results flows preview (best first)

This table basically shows the same kind of information shown in **table5** using an higher level of detail, for each malware the following info are shown:

**totalScans**: The number of antimalwares that analyze the malware.

**MaliciousToClean** :The number of antimalwares that judge the malware as malicious before transformations are applied and clean after their application.

**MaliciousToMalicious:** The number of antimalwares that judge the malware as malicious before transformations are applied as well as after their application.

**CleanToMalicious:** The number of antimalwaresthat judge the malware as clean before transformations are applied and malicious after their application.

**CleanToClean:** The number of antimalwares that judge the malware as clean before transformations are applied as well as after their application.

**This table** is ordered by descending values of the field **MTC\_on\_malicious\_pre**.

**Malware-family results**

In terms of malware families we developed this table that shows , for each family, how many malwares are able to “convince” the majority of the anti-malwares that scanned them, to be “clean”.

Only 21 families on 178 didn’t obtain the max of the score after the applications of the transformations.

Link table: [Malware family results (best first)](https://www.dropbox.com/s/ajwlyrgv8y4ldks/ResultsComparisonFamily_bf.html?dl=0)

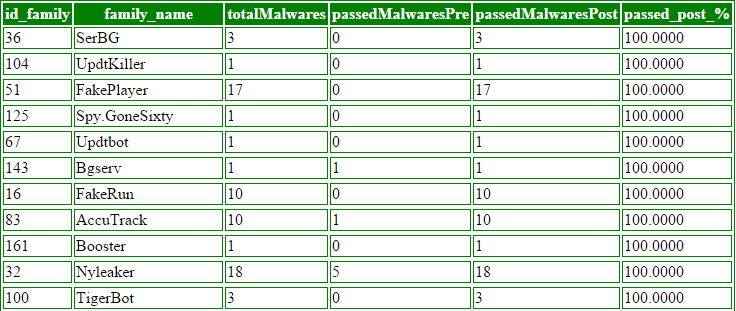


Table 7: malware family results preview (best first)

**totalMalwares**: the number of malwares belonging to a specific malware-family which have been analyzed both before and after being transformed at least by one antimalware.

**passedMalwaresPre**: the number of malwares belonging to a specific malware-family which are considered clean from the majority of antimalwares before being transformed.

**passedMalwaresPost**: the number of malwares belonging to a specific malware-family which are considered clean from the majority of antimalwares after being transformed.

**passed\_post\_%:**percentage of **passedMalwarePost** on **totalMalwares** .

**This table** is ordered by descending values of the field **passed\_post\_%**.

**RESULTS SUMMARY**

From the analysis of the aforementioned results we can make the following interesting statements :

* About **7%** of antimalwares under study detect as malicious more than **90%** of the transformed malwares that analyze.
  + On the original malware dataset this percentage is about **47%**.
* About **68%** of antimalwares under study detect as *malicious* less than **50%** of the transformed malwares that analyze.
  + On the original malware dataset this percentage is about **33%** .
* About **81%** of transformed malwares succeed in being cosidered clean/trusted by al least **50%** of the antimalwares that analyze them.
  + On the original malware dataset this percetage is about **5%**.
* About **77%** of malwares family have all their transformed malwares considered clean by the majority of antimalwares.
  + On the original malware dataset this percentage is about **6%.**

**IMPROVEMENTS PROPOSALS**

Developed transformations could be improved in several aspects trying to solve their described limitations.

Also transformations that can be detected from dynamic analysis can be added and have their impact studied.

The results we have gathered could be analyzed in more depth and provide

empirical support to many other useful considerations.

Furthermore every single transformation could be applied to the entire data-set analyzing every transformed data-set through virus-total.

The same could be done to study the impact of two transformations combined and so on for each possible combination length, improving the understanding of what modern antimalware detection algorithms lack the most.

**REFERENCES**

[1] *Vaibhav Rastogi, Yan Chen, and Xuxian Jiang, Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks, JANUARY 2014*

[2] [*http://developer.android.com/about/index.html*](http://developer.android.com/about/index.html)

[3] https://code.google.com/p/android-apktool/

[4] <https://www.virustotal.com>

[5] https://code.google.com/p/signapk/