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

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,all combined together, 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 (with a specific Java Api).
This website analyzes files (apk files in our case) with a set of widespread anti-malwares, each one of these gives the response on the malicious nature of the files analyzed.
We collected the results 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* which allows to 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:*

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*

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.example.prova">
    <application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ic_launcher"
        <activity android:label="@string/app_name" android:name="com.example.prova.GestoreInput">
            <intent-filter>
```

Android manifest Post-transformation

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.example.kIGAC">
    <application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ic_launcher"
        <activity android:label="@string/app_name" android:name="com.example.kIGAC.aov0ZYE">
            <intent-filter>
```

Class name Pre-transformation

```
.class public Lcom/example/prova/GestoreLogica;
.super Ljava/lang/Object;
.source "GestoreLogica.java"
```

Class name Post-trasnformation

```
.class public Lcom/example/kIGAC/Tzpq5ke;
.super Ljava/lang/Object;
.source "Tzpq5ke.java"
```

❖❖❖

Class call and package name Pre-Trasformation

```
    .line 26
    const-string v0, "Il totale \u00e8: "

    iput-object v0, p0, Lcom/example/prova/GestoreLogica;->s1:Ljava/lang/String;
```

Class call and package name Post-transformation

```
    .line 26
    const-string v0, "Il totale \u00e8: "

    iput-object v0, p0, Lcom/example/kIGAC/Tzpq5ke;->s1:Ljava/lang/String;
```

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

```
.line 26
const-string v0, "Il totale \u00e8: "

iput-object v0, p0, Lcom/example/prova/GestoreLogica;->s1:Ljava/lang/String;

.line 7
const/4 v0, 0x0
```

Post-transformation string

```
.line 26
const-string v0, "Kn vqvcng \u00e8< "
invoke-static {v0}, Lcom123456789/Decrypter;->applyCaesar(Ljava/lang/String;)Ljava/lang/String;
move-result-object v0
iput-object v0, p0, Lcom/example/prova/GestoreLogica;->s1:Ljava/lang/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

```
68
69      invoke-virtual {v0, v1, v2}, Lcom/example/prova/GestoreGrafica;->WriteOnDisplay(Ljava/lang/String;I)V
70
71      .line 37
72      return-void
73  .end method
```

Pre-transformation

```
68
69      invoke-static {v0, v1, v2}, Lcom/example/prova/GestoreInput$1;->method1553(Lcom/example/prova/GestoreGrafica;Ljava/lang/String;I)V
70
71      .line 37
72      return-void
73  .end method
74
75  .method public static method1553(Lcom/example/prova/GestoreGrafica;Ljava/lang/String;I)V
76      .locals 3
77      invoke-virtual {p0, p1, p2}, Lcom/example/prova/GestoreGrafica;->WriteOnDisplay(Ljava/lang/String;I)V
78      return-void
79  .end method
80
```

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

```
.method public subOne()V
    .locals 1

    .prologue
    .line 17
    iget v0, p0, Lcom/example/prova/GestoreLogica;->contatore:I

    add-int/lit8 v0, v0, -0x1

    iput v0, p0, Lcom/example/prova/GestoreLogica;->contatore:I

    .line 18
    return-void
.end method
```

Post-transformation

```
.method public subOne()V
    .locals 1

goto :i_1
:i_2
    .line 17
goto :i_3
:i_6
    .line 18
goto :i_7
:i_5
    iput v0, p0, Lcom/example/prova/GestoreLogica;->contatore:I
goto :i_6
:i_3
    iget v0, p0, Lcom/example/prova/GestoreLogica;->contatore:I
goto :i_4
:i_1
    .prologue
goto :i_2
:i_7
    return-void
:i_4
    add-int/lit8 v0, v0, -0x1
goto :i_5
.end method
```

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.

```
# virtual methods
.method public final run()V
    .locals 2

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->stopLoading()V

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->destroy()V

    iget-object v0, p0, Lc$a;->d:Lb;

    invoke-virtual {v0}, Lb;->a()V

    iget-boolean v0, p0, Lc$a;->f:Z

    if-eqz v0, :cond_0

    iget-object v0, p0, Lc$a;->b:Ld;

    invoke-virtual {v0}, Ld;->i()Lg;

    move-result-object v0

```

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

```
# virtual methods
.method public final run()V
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
    .locals 2

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->stopLoading()V

    iget-object v0, p0, Lc$a;->c:Landroid/webkit/WebView;

    invoke-virtual {v0}, Landroid/webkit/WebView;->destroy()V

    iget-object v0, p0, Lc$a;->d:Lb;

    invoke-virtual {v0}, Lb;->a()V

    iget-boolean v0, p0, Lc$a;->f:Z
```

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

```
.method static a(Ld;Ljava/util/Map;Landroid/net/Uri;Landroid/webkit/WebView;)V
    nop
    goto :saltojump_24
    :saltojump_24
    nop
    goto :saltojump_25
    :saltojump_25
    nop
    nop
    goto :saltojump_26
    :saltojump_26
    goto :saltojump_27
    :saltojump_27
    goto :saltojump_28
    :saltojump_28
    goto :saltojump_29
    :saltojump_29
    .locals 4
    .annotation system Ldalvik/annotation/Signature;
        value = {
```

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

```
.method public static a(Landroid/webkit/WebView;Ljava/lang/String;Ljava/lang/String;)V
    .locals 2

    const-string v0, "AFMA_ReceiveMessage"

    if-eqz p2, :cond_0

    new-instance v1, Ljava/lang/StringBuilder;

    invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

    invoke-virtual {v1, v0}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    move-result-object v0

    const-string v1, "(\'"

    invoke-virtual {v0, v1}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
.method public static a(Landroid/webkit/WebView;Ljava/lang/String;Ljava/lang/String;)V
    .locals 5
const/4 v2,0x0
const/4 v3,0x0
const/4 v4,0x0
const/4 v3,0x0
const/4 v4,0x0
const/4 v4,0x0
mul-int/2addr v4,v4
or-int/2addr v4,v2
and-int/2addr v3,v2


    const-string v0, "AFMA_ReceiveMessage"

    if-eqz p2, :cond_0

    new-instance v1, Ljava/lang/StringBuilder;

    move v2,v3
and-int/2addr v2,v4
add-int/2addr v4,v3
or-int/2addr v2,v3
add-int/2addr v3,v2
or-int/2addr v3,v3
and-int/2addr v3,v3
if-lt v4,v3,:Target_17
:Target_17
if-eq v2,v4,:Target_18
:Target_18
if-le v4,v2,:Target_19
:Target_19
if-ne v4,v4,:Target_20
:Target_20
if-le v2,v2,:Target_21
:Target_21
```

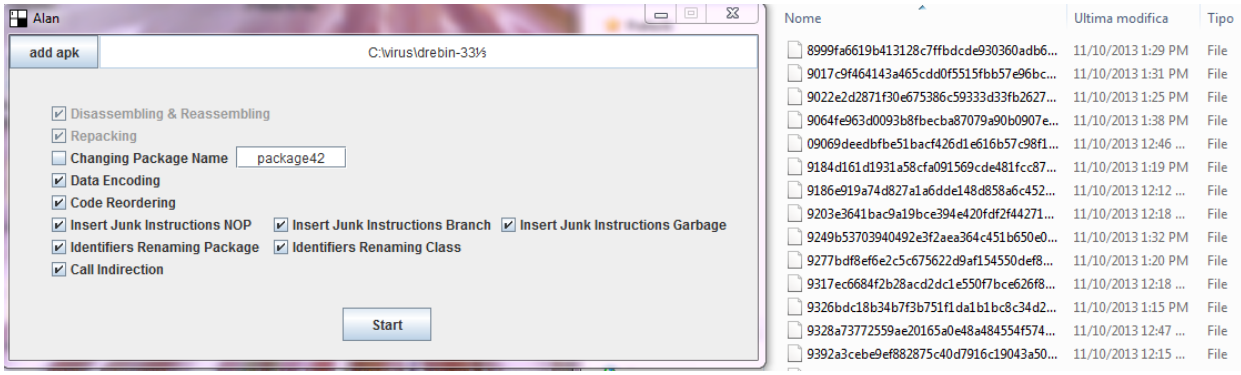
6) Composite Transformations:

All the transformations can be combined together without any problem.

FRAMEWORK ENGINE

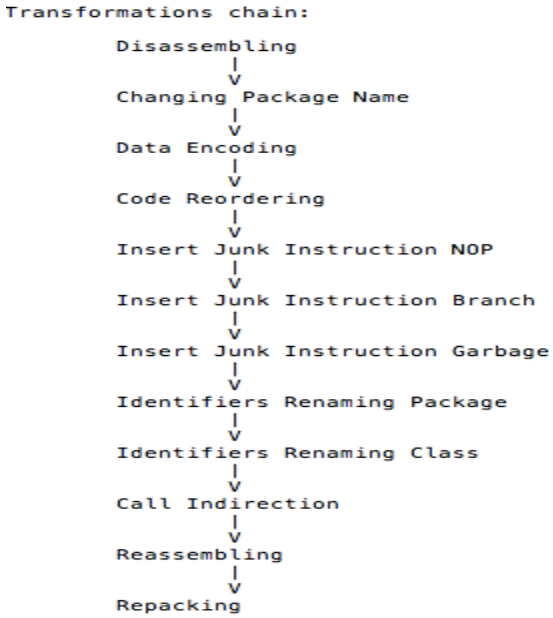
We developed this simple interface (coded in Java) that can be used to select one application or a directory containing several applications.

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 transformed applications into a specific directory.

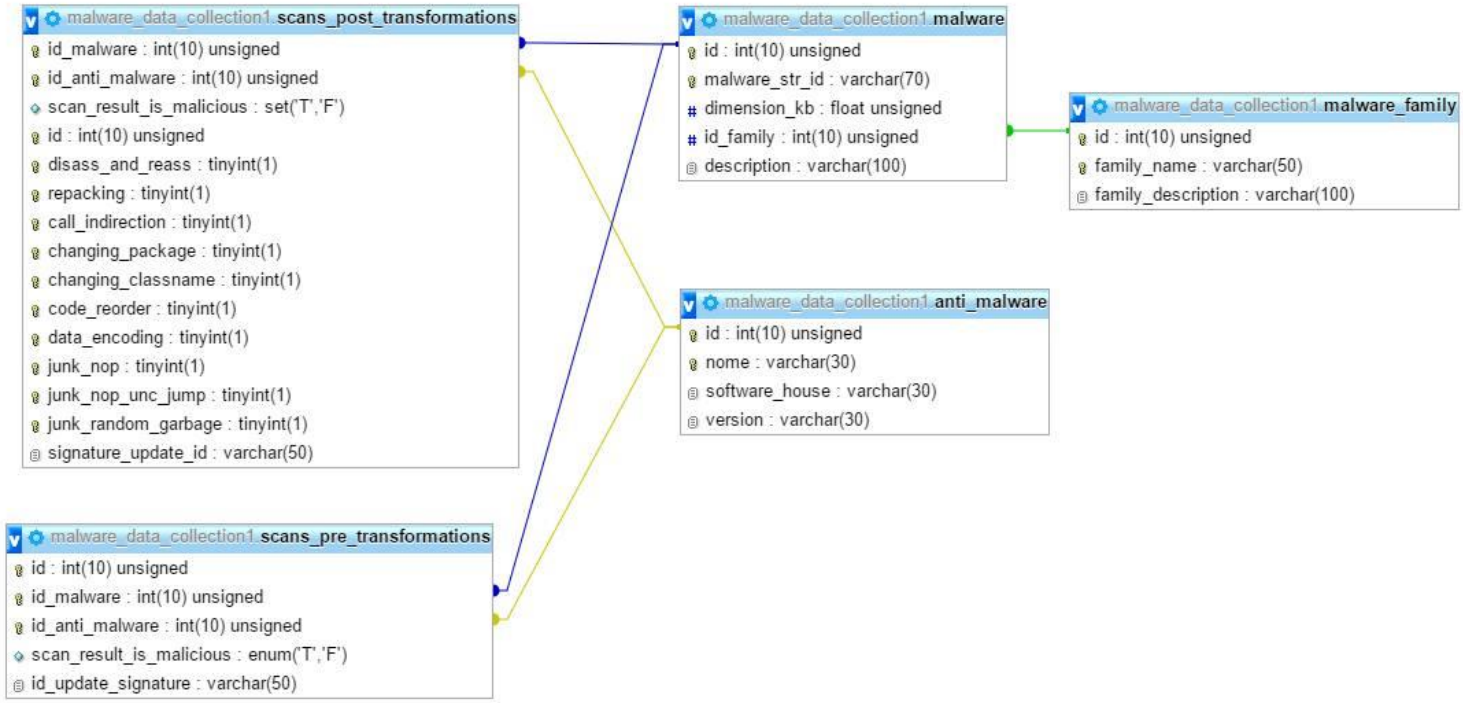
```
START DISASSEMBLING...
I: Using Apktool 2.0.0-RC3 on 8999fa6619b413128c7ffbdcde930360adb6508a59bf0db2d7
fd574b057748c6
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\Michele Meninno\apktool\framework\
1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
END DISASSEMBLING.

START DATA ENCODING...
        1 file copiati.
END DATA ENCODING.

START CODE REORDERING...
END CODE REORDERING.

START INSERT JUNK INSTRUCTION BRANCH-GARBAGE...
nop*--
goto :saltojump*:saltojump*--
..\8999fa6619b413128c7ffbdcde930360adb6508a59bf0db2d7fd574b057748c6.out\smali
..\8999fa6619b413128c7ffbdcde930360adb6508a59bf0db2d7fd574b057748c6.out\smali
END INSERT JUNK INSTRUCTION BRANCH-GARBAGE

START IDENTIFIER RENAMING...
END IDENTIFIER RENAMIN

START CALL INDIRECTION...
END CALL INDIRECTION

START REASSEMBLING...
I: Using Apktool 2.0.0-RC3 on 8999fa6619b413128c7ffbdcde930360adb6508a59bf0db2d7
fd574b057748c6.out
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
```

DATABASE

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

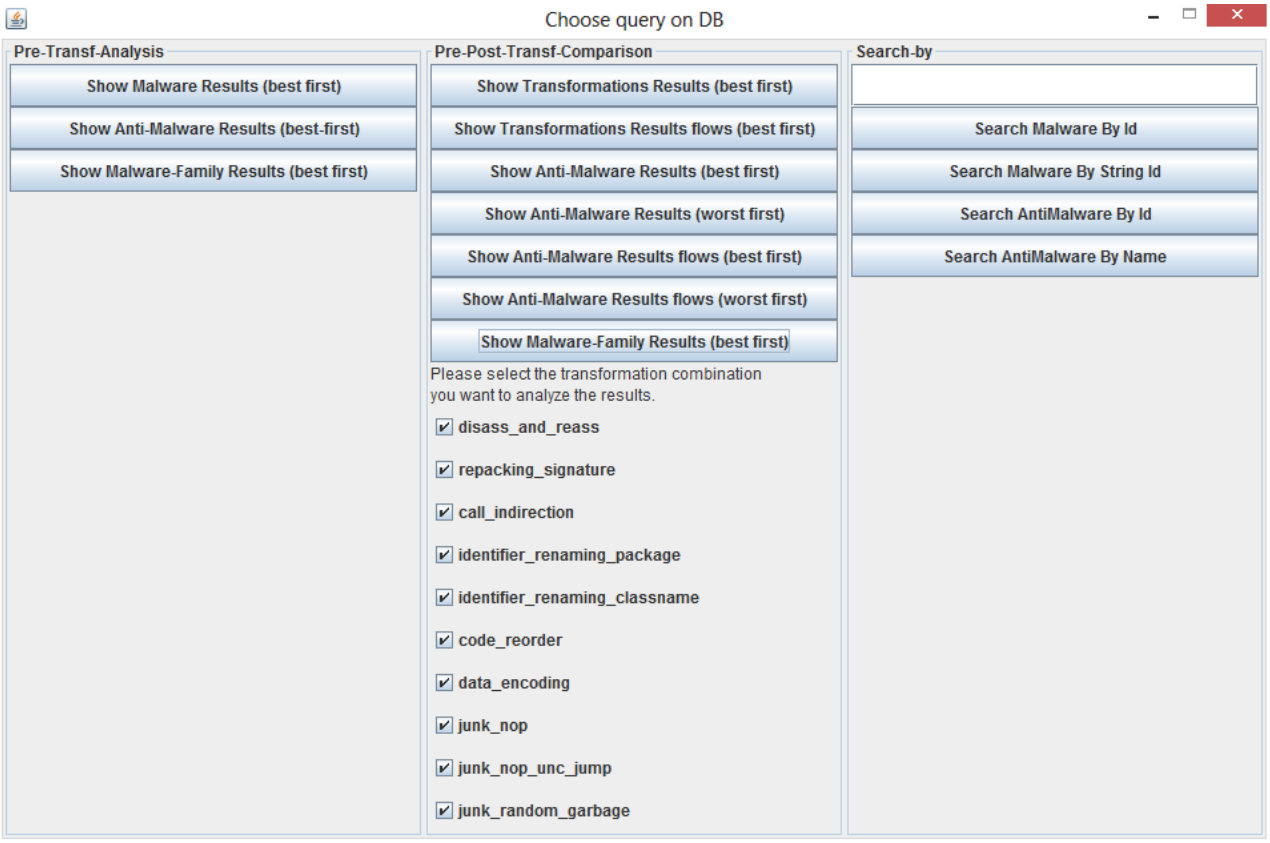
Our DB stores simple fine-grained data organized in 5 tables:

1) *malware*
   This table stores malwares data, the core subject of our analysis.
2) *anti-malware*
   This table stores anti-malwares data provided by *VirusTotal*.
3) *malware-family*
   This table stores malware-family information.
4) *scans-pre-transformations*
   This table stores *virusTotal* scan results of not-transformed malwares.
5) scans-post-transformations
   This table stores *virusTotal* scan results of transformed malwares.



Database schema

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



Database UI

Query results are presented in HTML files.

RESULTS OVERVIEW

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, table link: Anti-malwares result (best first)

id_anti_malware	nome	totalScans	totalMaliciousPre	malicious_on_total_pre_%	totalCleanPre	clean_on_total_pre_%	totalMaliciousPost	malicious_on_total_post_%	totalCleanPost	clean_on_total_post_%
10	Alibaba	578	565	97.7509	13	2.2491	578	100.0000	0	0.0000
31	F-Secure	4775	4723	98.9110	52	1.0890	4767	99.8325	8	0.1675
55	AVG	4776	4734	99.1206	42	0.8794	4432	92.7973	344	7.2027
51	ESET-NOD32	4586	4550	99.2150	36	0.7850	4169	90.9071	417	9.0929
39	Avira	4804	4749	98.8551	55	1.1449	4054	84.3880	750	15.6120
45	AhnLab-V3	4804	4615	96.0658	189	3.9342	3934	81.8901	870	18.1099
29	Sophos	4766	4719	99.0138	47	0.9862	3875	81.3051	891	18.6949
44	GData	4804	4747	98.8135	57	1.1865	3853	80.2040	951	19.7960
22	BitDefender	4803	4735	98.5842	68	1.4158	3848	80.1166	955	19.8834
28	Ad-Aware	4798	4726	98.4994	72	1.5006	3843	80.0959	955	19.9041
36	Emsisoft	4653	4594	98.7320	59	1.2680	3725	80.0559	928	19.9441
2	MicroWorld-eScan	4725	4642	98.2434	83	1.7566	3782	80.0423	943	19.9577
23	NANO-Antivirus	4804	4716	98.1682	88	1.8318	3702	77.0608	1102	22.9392
21	Kaspersky	4803	4689	97.6265	114	2.3735	3543	73.7664	1260	26.2336
19	Avast	4789	4175	87.1790	614	12.8210	3338	69.7014	1451	30.2986
32	DrWeb	4782	4610	96.4032	172	3.5968	3240	67.7541	1542	32.2459

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)

Table link: **Anti-malwares results flows (best first)**

id_anti_malware	nome	total_scans	MaliciousToClean	MTC_on_malicious_pre%	MaliciousToMalicious	MTM_on_malicious_pre%	CleanToMalicious	CTM_on_clean_pre%	CleanToClean	CTC_on_clean_pre%
10	Alibaba	578	0	0.0000	565	100.0000	13	100.0000	0	0.0000
31	F-Secure	4775	7	0.1482	4716	99.8518	51	98.0769	1	1.9231
40	Antiy-AVL	4687	1	1.2048	82	98.7952	0	0.0000	4604	100.0000
24	ViRobot	4789	4	3.4483	112	96.5517	0	0.0000	4673	100.0000
3	nProtect	4783	3	5.0847	56	94.9153	3	0.0635	4721	99.9365
55	AVG	4776	319	6.7385	4415	93.2615	17	40.4762	25	59.5238
51	ESET-NOD32	4586	394	8.6593	4156	91.3407	13	36.1111	23	63.8889
39	Avira	4804	733	15.4348	4016	84.5652	38	69.0909	17	30.9091
45	AhnLab-V3	4804	733	15.8830	3882	84.1170	52	27.5132	137	72.4868
29	Sophos	4766	869	18.4149	3850	81.5851	25	53.1915	22	46.8085

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.
The remaining fields show the percentage ratio between the values of the aforementioned fields and **(MaliciousToClean + MaliciousToMalicious)**.
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)

Table link: **Anti-Malwares result (worst first)**

id_anti_malware	nome	totalScans	totalMaliciousPre	malicious_on_total_pre_%	totalCleanPre	clean_on_total_pre_%	totalMaliciousPost	malicious_on_total_post_%	totalCleanPost	clean_on_total_post_%
26	ByteHero	4804	0	0.0000	4804	100.0000	0	0.0000	4804	100.0000
1	Bkav	4795	740	15.4327	4055	84.5673	0	0.0000	4795	100.0000
4	CMC	4790	2	0.0418	4788	99.9582	0	0.0000	4790	100.0000
7	Malwarebytes	4801	0	0.0000	4801	100.0000	0	0.0000	4801	100.0000
25	SUPERAntiSpyware	4801	2	0.0417	4799	99.9583	0	0.0000	4801	100.0000
12	TheHacker	4803	8	0.1666	4795	99.8334	2	0.0416	4801	99.9584
9	K7AntiVirus	4742	150	3.1632	4592	96.8368	36	0.7592	4706	99.2408
3	nProtect	4783	59	1.2335	4724	98.7665	59	1.2335	4724	98.7665
40	Antiy-AVL	4687	83	1.7709	4604	98.2291	82	1.7495	4605	98.2505
49	Panda	4773	185	3.8760	4588	96.1240	97	2.0323	4676	97.9677

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

This table is ordered by descending values of the field **clean_on_total_post.**

Anti-malwares results flows (worst first)

Table link: **Anti-malwares results flows (worst first)**

id_anti_malware	nome	total_scans	MaliciousToClean	MTC_on_malicious_pre%	MaliciousToMalicious	MTM_on_malicious_pre%	CleanToMalicious	CTM_on_clean_pre%	CleanToClean	CTC_on_clean_pre%
4	CMC	4790	2	100.0000	0	0.0000	0	0.0000	4788	100.0000
25	SUPERAntiSpyware	4801	2	100.0000	0	0.0000	0	0.0000	4799	100.0000
1	Bkav	4795	740	100.0000	0	0.0000	0	0.0000	4055	100.0000
56	Baidu-International	4794	3947	94.9483	210	5.0517	12	1.8838	625	98.1162
11	K7GW	239	206	93.2127	15	6.7873	0	0.0000	18	100.0000
41	Kingsoft	4763	3925	91.9850	342	8.0150	25	5.0403	471	94.9597
50	Zoner	4804	3562	90.5670	371	9.4330	18	2.0666	853	97.9334
17	TotalDefense	4793	1767	90.1531	193	9.8469	14	0.4942	2819	99.5058
14	F-Prot	4804	4193	89.3649	499	10.6351	6	5.3571	106	94.6429
6	McAfee	4801	4191	87.6045	593	12.3955	7	41.1765	10	58.8235

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

This table is ordered by descending values of the field **MTC_on_malicious_pre**.

Malware results (best first)

Table link: **Malwares results (best first)**

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

id_malware	totalScans	totalMaliciousPre	malicious_on_total_pre_%	totalCleanPre	clean_on_total_pre_%	totalMaliciousPost	malicious_on_total_post%	totalCleanPost	clean_on_total_post_%
5308	53	22	41.5094	31	58.4906	0	0.0000	53	100.0000
4015	53	20	37.7358	33	62.2642	0	0.0000	53	100.0000
3526	55	20	36.3636	35	63.6364	1	1.8182	54	98.1818
3724	55	23	41.8182	32	58.1818	1	1.8182	54	98.1818
633	55	26	47.2727	29	52.7273	1	1.8182	54	98.1818
3269	54	26	48.1481	28	51.8519	1	1.8519	53	98.1481
4517	54	29	53.7037	25	46.2963	1	1.8519	53	98.1481
303	54	29	53.7037	25	46.2963	1	1.8519	53	98.1481
803	54	31	57.4074	23	42.5926	1	1.8519	53	98.1481
5188	54	36	66.6667	18	33.3333	1	1.8519	53	98.1481

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)

Table link: **Malware results flows (best first)**

id_malware	totalScans	MaliciousToClean	MTC_on_malicious_pre%	MaliciousToMalicious	MTM_on_malicious_pre%	CleanToMalicious	CTM_on_clean_pre%	CleanToClean	CTC_on_clean_pre%
4741	55	12	100.0000	0	0.0000	2	4.6512	41	95.3488
371	56	24	100.0000	0	0.0000	2	6.2500	30	93.7500
3526	55	20	100.0000	0	0.0000	1	2.8571	34	97.1429
5308	53	22	100.0000	0	0.0000	0	0.0000	31	100.0000
4015	53	20	100.0000	0	0.0000	0	0.0000	33	100.0000
2716	55	13	100.0000	0	0.0000	3	7.1429	39	92.8571
413	54	40	97.5610	1	2.4390	0	0.0000	13	100.0000
5188	54	35	97.2222	1	2.7778	0	0.0000	18	100.0000
3464	54	33	97.0588	1	2.9412	0	0.0000	20	100.0000
803	54	30	96.7742	1	3.2258	0	0.0000	23	100.0000

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 antimalwares that 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.
The remaining fields show the percentage ratio between the values of the aforementioned fields and **(MaliciousToClean + MaliciousToMalicious)**.
This table is ordered by descending values of the field **MTC_on_malicious_pre**.

Malware-family results

In order to take into cosideration 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 40 families on 178 didn't obtain the max of the score (100%) after the applications of the transformations.

Table link: Malware family results (best first)

id_family	family_name	totalMalwares	passedMalwaresPre	passedMalwaresPost	passed_post_%
36	SerBG	3	0	3	100.0000
104	UpdtKiller	1	0	1	100.0000
51	FakePlayer	17	0	17	100.0000
125	Spy.GoneSixty	1	0	1	100.0000
67	Updtbot	1	0	1	100.0000
143	Bgserv	1	1	1	100.0000
16	FakeRun	10	0	10	100.0000
83	AccuTrack	10	1	10	100.0000
161	Booster	1	0	1	100.0000
32	Nyleaker	18	5	18	100.0000
100	TigerBot	3	0	3	100.0000

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/trusted 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*

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

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

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