Android: From Reversing to Decompilation

Anthony Desnos, Geoffroy Gueguen
ESIEA: Operational Cryptology and Virology Laboratory
desnos@esiea.fr, gueguen@esiea.fr

Abstract

This talk deals with Android's bytecode analysis. The Android system is now widespread, and lots of applications are developed each days. These applications are mostly written in Java, though it is possible to do calls to binaries or shared libraries. To be executed on the DVM the Java source code is translated into Java bytecode (.class files) and then a tool named 'dx' is used to convert it into the Dalvik format (.dex files). Such a conversion is needed as the DVM is a register-based machine whereas the JVM is a stack-based one, and as such they have different opcodes.

Due to the nature of the bytecode, its reversing is somewhat easier than machine code. Indeed, unlike machine code, (Dalvik) bytecode contains semantic information that allows us to do a better analysis. We can get useful details on variables, fields, methods... We can create signatures for a method, or we can use the android permissions to see where a specific one is used in an application. The analysis part allows us to extract the control flow graph (which is composed of basic blocks, and which cannot be modified dynamically due to the virtual machine) which is used to represent the different possibles paths of an application.

Furthermore, we have implemented new algorithms to calculate the similarity distance between two applications, a useful information to know if your application has been stolen from the android market. It's also possible to use similarity to do 'diffing' of Android applications is useful to see patches of bugs or insertion of evil code, this is why we have developed a combination of techniques to quickly see the differences between two applications. Moreover it's interesting to have the ability to manipulate in a simple way all these new formats (APK, DEX, Dalvik bytecode, Android's binary xml) to automate testing directly in a program or in a specific interpreter.

There are some ways to retrieve the Java source code of an application from the bytecode, for instance some people use a software which transforms Dex bytecode into Java bytecode and then combine it with a regular Java decompiler. But the resulting code often looks like an obfuscated version which does not compile than real source code. That's why we developed a new decompiler which uses only Dalvik bytecode to create an original Java source code. We present a new open-source tool (Androguard) written in Python (and some parts of C language) which help the reversing of Android applications, as well as our decompiler.

1 Introduction

According to [22], Android consumers in the U.S spend an average of 56 minutes per day on their phones, with two-thirds of this time spent with Android applications. Moreover, Andy Rubin, Senior Vice President of Mobile at Google [31] stated that they have over 500.000 Android devices activated every day. So it is easy to understand [26] why this platform is an increasingly interesting target for misuse.

Many tools have been released to interact with Android applications. Assemblers and dissasemblers, like smali/bakmali [17] are useful to reverse engineer Android applications. Advanced tools such as apktool [7] are very useful to modify an application and repackage it. Decompilers [30] [28] transform Android bytecode to Java bytecode to enable use of classical Java decompilers [23] [14] [27], although they have some issues [4] during the decompilation.

In this paper, we introduce new algorithms for static analysis of Android applications, in order to address some problems that users and developers are encountering. All of the algorithms are based on the similarity distance using real world compressors [11]. The first problem that we discuss is how it is possible to create a rip-off indicator to identify whether an application is similar to another one or not, i.e., to determine if someone pirated an application or parts of an application. We can extend this problem to extract automatically malware that has been injected into an application. In addition, if we can find similarities between two applications, the algorithm can be applied to evaluate the efficiency of an obfuscator on the application. A secondary issue that we address, is how it is possible to obtain and visualize the differences between two valid versions of an application. We have used our algorithms to identify small dissimilarities in methods and in basic blocks (without using graphs) in order to use the longest common subsequence algorithm [1] to extract exact differences. We then discuss how it is possible to recover the source code of an application thanks to the information the dalvik bytecode carries. We also show an amusing bug we found in disassemblers regarding the recovery of numbers.

In the first section, we define the similarity distance, and describe how we have choose our compressors in order to have the maximum performance with the best distance similarity. Next, we describe our main algorithm to compare applications in order to find similarities. We then extend this algorithm to identify dissimilarities and perform difference analysis between two applications. In the last section we describe the different phases commonly implemented in a decompiler and how we adapted them to our own. In the conclusion, we discuss future work and some open problems to encourage further research in this area.

2 Similarity

The similarity distance based on real world compressors is called the Normalized Compression Distance (NCD) [11]. The NCD of two elements A and B is defined as $d_{NCD}(A, B)$. We can compute

- $C(A)$ and $L_A = L(C(A))$;
- $C(B)$ and $L_B = L(C(B))$;
- $C(A|B)$ and $L_{A|B} = L(C(A|B))$;

where $A|B$ is the concatenation of A and B, C is the compressor, and L is the length of a string. Then $d_{NCD}(A, B)$ is defined by

$$d_{NCD}(A, B) = \frac{L_{A|B} - \min(L_A, L_B)}{\max(L_A, L_B)}. \tag{1}$$

The NCD is based on the similarity [11] of elements. A compressor C is normal if the following four axioms are satisfied up to an additive O(log n), where n is the maximal binary length of the elements involved in the inequalities:

1. Idempotency: C(xx) = C(x), and C(ε) = 0, where ε is the empty string.
2. Monotonicity: C(xy) $\geqslant$ C(x).
3. Symmetry: C(xy) = C(yx).
4. Distributivity: C(xy) + C(z) $\leqslant$ C(xz) + C(yz).

The idea demonstrated in the following simple example. If you take three elements

- X ("HELLO WORLD") and the length of the compression Y = C(X) = 6,
- X' ("HELLO WOORLD") and the length of the compression of Y' = C(X') = 7,
- X" ("HI !!!") and the length of the compression of Y" = C(X") = 3.

the compression of C(XX’) will be similar to C(X) whereas the compression of C(XX”) will not be similar to C(X).

In addition to these four inequalities being respected by a given compressor, in a real context we must have timing constraints for execution of the algorithm. The compression rate is not a determining factor for the choice of the compressor if it complies with the following rules:

1. C respects the four inequalities,
2. C(x) is calculated within an acceptable amount of time.

We studied 5 implementations of different algorithms to choose the one that respects the most inequalities:

- zlib [25], bz2 [3], LZMA [2], XZ [12], Snappy [18].

We chose to test all compressors with random text data sets [13] (the test sets are text data, because we will not submit binary data to our compressor) on the four inequalities. The following tables show the resulting number of matches, the total length of compression, and the execution time of each compressor:

	Idempotency
LZMA	0/9, 900, 1.45565796
XZ	0/9, 1824, 0.72005010
zlib	0/9, 894, 0.00037599
bz2	0/9, 1294, 0.00088286
Snappy	1/9, 1208, 0.00010705

	Monotonicity
LZMA	72/72, 9645, 11.65594506
XZ	72/72, 17108, 5.70679307
zlib	72/72, 9177, 0.00256300
bz2	72/72, 11334, 0.00596595
Snappy	72/72, 13354, 0.00065804

	Symmetry
LZMA	26/72, 12266, 11.68412399
XZ	54/72, 20008, 5.68789411
zlib	26/72, 11474, 0.00305796
bz2	72/72, 12972, 0.00746584
Snappy	14/72, 17476, 0.00071502

	Distributivity
LZMA	504/504, 153377, 163.45748401
XZ	504/504, 259812, 79.99475789
zlib	504/504, 144557, 0.03746986
bz2	504/504, 170142, 0.09060693
Snappy	504/504, 215810, 0.00880289

After these tests (more in [13]) Snappy was selected because speed of compression is far superior and this compressor respects the inequalities.

2.1 Similarity Algorithm

We propose an algorithm to find efficiently and quickly the similarities (and differences) of methods between two applications (without debugging information). In our case, we have tested and implemented it with bytecode (Java [29] or Dalvik [20]) files but it is possible to extend this algorithm to classical binaries. This algorithm is an improvement in the field of executable comparison as it applies new techniques with very efficient, precise, and timely results in many domains.

This algorithm is composed of the following steps:

- Generate signatures for each method.
- Identify all methods which are identical.
- Identify all methods which are partially identical by using NCD (with Snappy compressor).

So the global idea is to associate each method of the first application with others of the second application (unless the method match directly) by using NCD with an appropriate compressor.
The algorithm produces and detects the following elements:

- identical methods,
- similar methods,
- new methods,
- deleted methods.

2.1.1 Initialization of methods

The algorithm (1) must initialize each method, so we can associate several attributes to a method as shown in figure 1:

- the entropy, based on the raw binary data,
- a buffer which represents the sequence of instructions, with useless information removed from it,
- a unique checksum (or hash) based on the previous buffer,
- a signature.

In order to create the checksum of a method, we must remove and retain some information associated with an instruction. It is a very important to remove false positives easily. Thus we need to remove information which is a direct result of the compilation and information which can be changed by a simple modification:

- registers, offset of a jump instruction.

And we need to retain:

- the original name of the instruction (group in a single name the jump instructions), strings, integers and floats, constants, etc.

Algorithm 1 Algorithm for filtering names and operands of an Android instruction

```
Require: Instruction
  buffer ← ””
  // goto instructions
  if Instruction.opvalue ⩾ 0x28 and Instruction.opvalue ⩽ 0x2a then
    buffer ← ”goto”
  else
    buffer ← Instruction.opname
  end if
  // an integer is present in the instruction
  if integer in Instruction.operands then
    buffer ← buffer + getIntegers( Instruction )
  end if
  // a string is present in the instruction
  if Instruction.opvalue == 0x1a then
    buffer ← buffer + getString( Instruction )
  end if
  return buffer
```

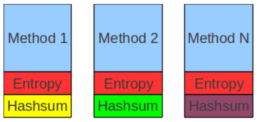


Figure 1: Attributes associated with a method

So if we use a unique checksum for each method, it is possible to quickly remove from the comparison, methods which are exactly identical as show in figure 2.

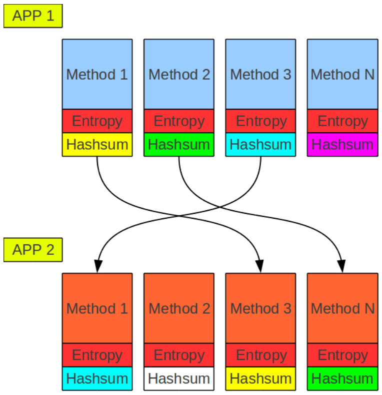


Figure 2: Remove identical methods by using hash

2.1.2 Generating signatures of methods

We used a normal grammar presented by Silvio Cesare [9] to generate different types of signatures as needed including the following:

- Control Flow Graph,
- API (current application, Android, Java),
- Strings,
- Exceptions.

We considered this signature as an interesting input for the distance similarity because this signature does not include information about instructions. The information in the signature is more general, (e.g., the presence of a basic block, if a specific package is called, etc). Of course it is possible to modify the grammar in order to have a controlled output.

The following is an example of the grammar signature for a method:

```
Procedure ::= StatementList
StatementList ::= Statement | Statement StatementList
Statement ::= BasicBlock | Return | Goto | If | Field | Package | String | Exception
Return ::= 'R'
Goto ::= 'G'
If ::= 'I'
BasicBlock ::= 'B'
Field ::= 'F'0 | 'F'1
Package ::= 'P' PackageNew | 'P' PackageCall
PackageNew ::= 'C'
PackageCall ::= 'M'
PackageName ::= ε | Id
String ::= 'S' Number | 'S' Id
Exception ::= Id
Number ::= \d+
Id ::= [a–zA–Z]\w+
```

Thus we can generate a particular signature for a method based on selected features. We must create the best signature to identify methods, in order to identify highly identical methods. Moreover we have not used particular dependencies like strings, names of classes/methods/fields, etc. We have used only the following elements:

- Control Flow Graph,
- External API used (Android + Java),
- Exceptions.

For example, we can generate different signatures for one method, with more or less detail:

```
Lorg/t0t0/androguard/TC/TCD; equal (I Ljava/lang/String;)Ljava/lang/String;
-> :     B[P1F0P0P1P1SP1P1SP1P1P1P1IS]B[S]B[RS]B[SG]
-> :     B[P1F0P0P1P1S1P1P1S6P1P1P1P1IS4]B[S4]B[RS4]B[S4G]
-> :     B[F0SSIS]B[S]B[RS]B[SG]
-> :     B[P1{Ljava/lang/Integer;toString(I)Ljava/lang/String;}F0P0{Ljava/lang/StringBuilder
   ;}[...]P1{Ljava/lang/String;equals(Ljava/lang/Object;)Z}IS]B[S]B[RS]B[SG]
```

2.1.3 Identification of methods

If we need to ensure a better match between methods, it is possible to add the following elements to calculate the similarity distance by using a filtering algorithm [5]:

- the entropy,
- the buffer which represents the sequence of instructions.

With the NCD (Algorithm 2, the signature function returns the previous described signature, and the clean function removes useless information of each instruction of the method), we use the Snappy compressor to accelerate the comparison between all methods, to detect exactly the same methods (algorithm 3, figure 3), partially the same methods (algorithm 3), new methods (algorithm 4, figure 4) or deleted methods (algorithm 5).

Algorithm 2 Algorithm of similarity to calculate the distance between two methods

```
Require: a1, a2
  n1 ← NCD( signature( a1 ), signature( a2 ) )
  n2 ← NCD( clean( a1 ), clean( a2 ) )
  return (n1 + n2) / 2
```

Algorithm 3 Algorithm for finding identical or similar methods between two applications

```
Require: apps1, apps2
  exact ← {}
  diff ← {}

  for a1 ← apps1.methods() do
    if a1.hash not in apps2.hash then
      for a2 ← apps2.methods() do
        if a2.hash not in apps1.hash then
          Append( diff[ a1 ], similarity( a1, a2 ) )
        end if
      end for
    else
      Append( exact, a1 )
    end if
  end for
  for a1 ← diff.methods() do
    a1.sort()
  end for
```

Algorithm 4 Algorithm for identifying new methods

```
Require: apps1, apps2, diff
  new ← []
  for a2 ← apps2.methods() do
    if a2.hash not in apps1.hash then
      state ← true
      for a1 ← diff.methods() do
        if a1.checksort( a2 ) then
          state ← false
          break
        end if
      end for
      if state == true then
        Append(new, a2)
      end if
    end if
  end for
```

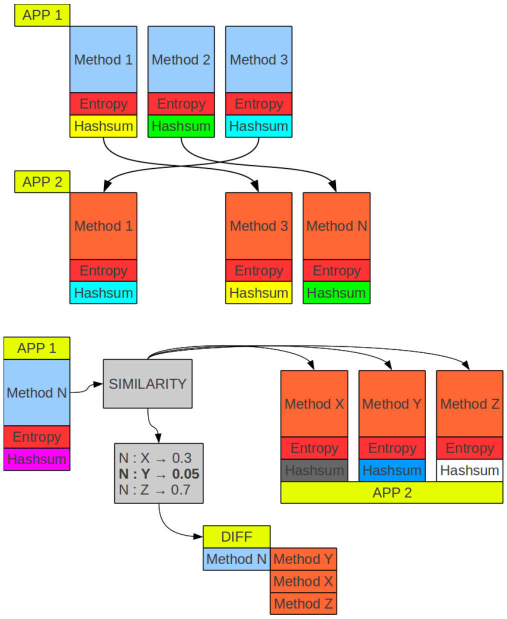


Figure 3: Find exact/similar methods between two applications

Algorithm 5 Algorithm for identifying deleted methods

```
Require: apps1, apps2, diff
  delete ← []
  for a1 ← diff.methods() do
    if a1.getsort() == 0 then
      Append( delete, a1 )
    end if
  end for
```

The signature of each method helps to compare all methods more quickly, but an algorithm based on clustering can filter identical methods before our algorithm in order to accelerate the comparison, and can be very interesting in the case of comparing one element against N elements.

It is possible to detect all different or identical methods, but we do not care about the entire application. This is a problem as we need to take into account other information associated with an Android application in order to extend the ratio of similarity to a entire application.

We can add the following information from an application:

- strings, constants (integers, floats), various internal data like ”fill array data”.

2.2 Detecting pirated applications aka rip-off indicator

A major problem in the Android market is the theft of applications, because it is very easy to download an application and to crack/re-package it with smali/basksmali/apk-tool to push it in different markets. While it is possible to compare an application with reverse engineering tool manually using a tool such as baksmali [17], IDA [16] or Androguard [13], it is very time consuming and inefficient. Automated approaches are better to compare other applications with the original when piracy is suspected.

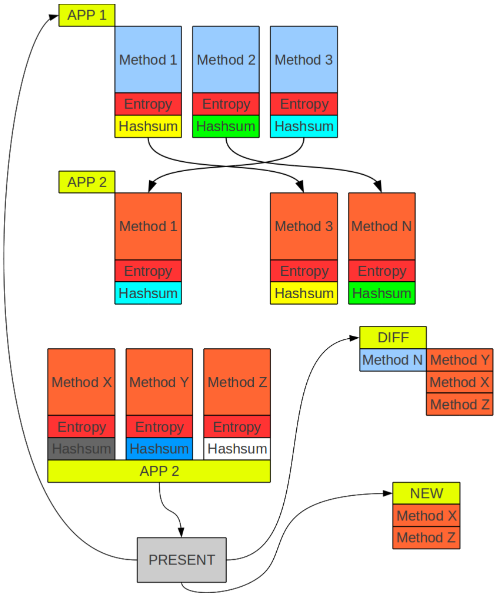


Figure 4: Identify new methods between two applications

With the previous algorithms, we have all the required information to calculate an indicator (between 0.0 to 100.0) to indicate whether the application has been stolen.

To do that (Algorithm 6), we apply a designation (between 0.0 (identical) to 1.0 (different)) to each attribute:

- 0.0 to a perfect identical method,
- value of the NCD for a partial identical method,
- value of the NCD for the general information of the application (strings, constants, etc.).

Moreover we exchange the compressor (Snappy) at the end of the algorithm with the XZ compressor in order to have a more comprehensible value for the final user.

Algorithm 6 Algorithm to calculate the similarity between two applications

```
Require: diff, exact
  marks ← []
  for a1 ← diff.methods() do
    Append( marks, GetFirstSortValue( a1 ) )
  end for
  for a1 ← exact.methods() do
    Append( marks, 0.0 )
  end for
  finalmark ← 0.0
  for i ← marks.get() do
    finalmark ← (finalmark + (1.0 - i))
  end for
  finalmark ← (finalmark / Len( marks )) * 100
```

We have tested the algorithm 6 in multiple cases. The cases include two different applications (Listing 1), two identical applications (Listing 2), two "quite" identical applications (Listing 3), and with a known stolen and repackaged application (Listing 4).

Listing 1: Similarity of two different applications

```
desnos@destiny:~/androguard$ ./androsim.py -i examples/obfu/classes_tc.dex apks/classes.dex
DIFF METHODS : 3
NEW METHODS : 199
MATCH METHODS : 0
DELETE METHODS : 4
[0.99816107749938965, 1.0, 1.0, 1.0]
0.0459730625153
```

Listing 2: Similarity of identical applications

```
DIFF METHODS : 0
NEW METHODS : 0
MATCH METHODS : 14
DELETE METHODS : 0
[0.08235294371843338, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
99.4509803752
```

Listing 3: Similarity of quite identical applications

```
DIFF METHODS : 1
NEW METHODS : 0
MATCH METHODS : 12
DELETE METHODS : 0
[0.14427860081195831, 0.095238097012042999, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0]
98.2891664441
```

Listing 4: Similarity of a stolen application

```
desnos@destiny:~/androguard$ ./androsim.py -i apks/HolyFuckingBiblev11-market-militia-.apk
    apks/holyfuckingbible.apk
DIFF METHODS : 1
NEW METHODS : 81
MATCH METHODS : 72
DELETE METHODS : 0
[0.8460613489151001, 0.091269843280315399, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
98.7333362268
```

We can see that the final mark is very useful for a developer to determine whether his application has been stolen or not, partially or totally, because he can inspect each method which matches with another.

2.3 Evaluation of Android obfuscators

Another problem with Java or Android applications is the transformation of the source code in bytecode. This transformation can be easily reversed by using a classical decompiler like jad, jd-gui or dava, with varying degrees of reliability. Moreover virtual machines do not allow code modification on the fly (polymorphic) and it is a real problem for classical packers. Nevertheless, Android developers use obfuscators frequently such as proguard [24] or dasho [33] to prevent the reverse engineering of their software.

This leads to an interesting problem. "How is it possible to evaluate an obfuscator?". We can answer this question by using similarity distance as described previously. If two applications have a high number of equal functions (or almost equal functions), we can say that there is a problem with the obfuscation process.

The obfuscator can use several techniques to protect a Java/Android application:

1. change names of classes, methods, fields,
2. modify the control flow,
3. code optimization,
4. dynamic code loading,
5. change instructions with metamorphic technique.

The first option is not efficient with our similarity distance because we do not use debugging information. The second option can be a problem with a classical distance, but the similarity distance is not sensible to this point due to the real compressor. The third can be detected as removal instructions. The two latest option is an issue and it is not covered by our algorithm. (We have not yet seen any tool which uses this technique.) Even if this technique will be used we will have to do a normalisation of each method before applying our algorithm.

We can protect an application with an obfuscator and we can calculate the distance by using a blackbox technique (our similarity algorithm). If this distance is close to 100 then the obfuscator did a poor job by using the first three techniques (or other equivalent techniques) as shown in listing 5.

Listing 5: Similarity of application protected by proguard and dasho

```
desnos@destiny:~/androguard$ ./androsim.py -i examples/obfu/classes_tc.dex examples/obfu/
    classes_tc_proguard.dex
DIFF METHODS : 7
NEW METHODS : 4
MATCH METHODS : 0
DELETE METHODS : 0
[0.47394958138465881, 0.040816325694322586, 0.059999998658895493, 0.040816325694322586,
    0.059999998658895493, 0.13333334028720856, 0.040816325694322586, 0.095238097012042999]
88.1878750864
desnos@destiny:~/androguard$ ./androsim.py -i examples/obfu/classes_tc.dex examples/obfu/
    classes_tc_dasho.dex
DIFF METHODS : 2
NEW METHODS : 0
MATCH METHODS : 10
DELETE METHODS : 0
[0.50084036588668823, 0.13114753365516663, 0.1428571492433548, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0]
94.0396534709
```

2.4 Malware

In our algorithm, we can extract automatically new methods (methods that are not present in the first application but in the second one). Typically, the case of an injected malware (Listing 6) in the Android official or unofficial markets, a classical application is taken from a market, and the malware writer injects his "evil" code in the application and propagates the new application in different markets.

Listing 6: Similarity of application with injected malware

```
desnos@destiny:~/androguard$ ./androsim.py -i apks/com.swampy.sexpos_162.apk apks/com.swampy
    .sexpos.apk-GEINIMI-INFECTED.apk
DIFF METHODS : 0
NEW METHODS : 51
```

```
MATCH METHODS : 218
DELETE METHODS : 0
[1.0, 0.0, [...]]
99.5433789954
desnos@destiny:~/androguard$ ./androsim.py -i apks/TAT-LWP-Mod-Dandelion-orig.apk apks/TAT-
    LWP-Mod-Dandelion.apk
DIFF METHODS : 0
NEW METHODS : 31
MATCH METHODS : 18
DELETE METHODS : 0
[0.68480598926544189, 0.0, [...]]
96.3957579512
```

It is possible to isolate the malware quickly if we know the original application, which is an easy task because the malware writer does not generally modify it, (i.e., all original code is intact). Further, it is also possible to use this technique with multiple (but different) samples, in order to extract identical methods, because these methods will be the methods of the malware.

Moreover we can isolate new injected methods, but it is possible to extend our algorithm by using diffing techniques to determine whether someone has modified an original method to inject a hook or not.

3 Differences between applications

It is interesting to calculate the differences between two versions of an application (for reverse engineering) to identify modifications in order to find a security bugfix, or after the injection of malware. The idea is to detect classical modifications in a method including:

- modification of codes in a basic block,
- addition of new basic blocks.

We can use the algorithm of similarity in the previous section to find identical/similar methods in order to extract modifications of instructions from basic blocks. We can add the following steps:

- Identification of identical basic blocks by using NCD,
- Extraction of added/removed instructions by using the longest common subsequence algorithm (LCS)[1].

Futhermore it is possible to change the signature of a method to have a small one to accelerate operations by removing exceptions and API because it is not an important point in this algorithm.

3.1 Identification of basic blocks

We have used the same algorithm 2 to compare basic blocks (algorithms 7, 8, 9, figures 5, 6), because it is just a different level of granularity. So we use the NCD (also with Snappy compressor) to compare basic blocks and checkums/hashes to quickly eliminate identical basic blocks.

3.1.1 Identification of added or removed instructions

After the identification of basic blocks we can use a classical algorithm like the longest common subsequence problem (LCS) [1] to extract new and old instructions (algorithm 10).

So we must modify our sequence of instructions into a string to use the LCS algorithm. We can also used a filtered algorithm on each single instruction to convert it to a value between 0 to 255.

For instance, the following basic blocks:

```
ADD 3
ADD 1
SUB 2
```

Algorithm 7 Algorithm of matching exactly or partially basic blocks between two methods

```
Require: bb1, bb2
  diffbb ← {}
  for b1 ← bb1.basicblocks() do
    if b1.hash not in bb2.hash then
      for b2 ← bb2.basicblocks() do
        if b2.hash not in bb1.hash then
          Append( diffbb[ b1 ], similarity(b1, b2) )
        end if
      end for
    end if
  end for
  for b1 ← diffbb.basicblocks() do
    b1.sort()
  end for
```

Algorithm 8 Algorithm of similarity between two basic blocks

```
Require: a1, a2
  n1 ← NCD( checksum( a1 ), checksum( a2 ) )
  return n1
```

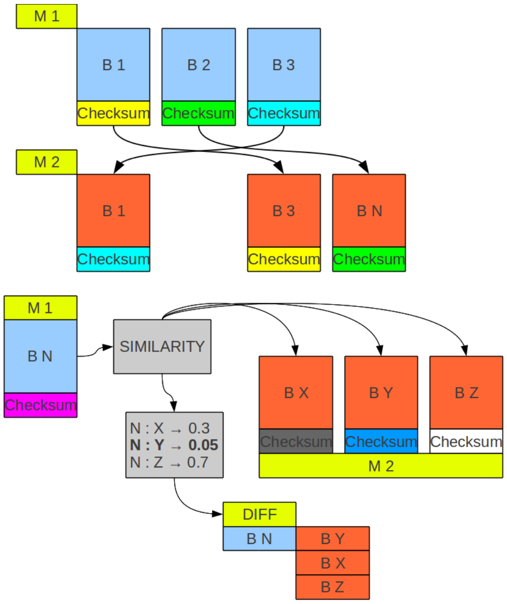


Figure 5: Find exactly/partially the same basic blocks between two methods

```
IGET
ADD 3
GOTO

ADD 3
ADD 3
SUB 2
IGET
```

Algorithm 9 Algorithm of matching exactly or partially basic blocks between two methods

```
Require: bb1, bb2
  newbb ← []
  for b2 ← bb2.basicblocks() do
    if b2.hash not in bb2.hash then
      state ← true
      for b1 ← diffbb.basicblocks() do
        if b1.checksort( b2 ) then
          state ← false
          break
        end if
      end for
      if state == true then
        Append( newbb, b2 )
      end if
    end if
  end for
```

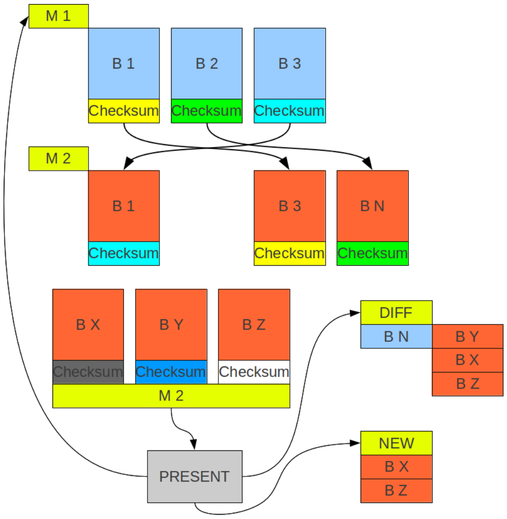


Figure 6: Find new basic blocks between two methods

Algorithm 10 Algorithm to extract added/removed instructions from a basic block

```
Require: b1, b2, toString
  addins ← []
  delins ← []
  X ← toString( b1 )
  Y ← toString( b2 )
  addins, delins ← LCS( X, Y )
```

```
MUL  4
GOTO
```

are transformed in the following strings :

```
”\x00\x01\x02\x03\x00\x04”
”\x00\x00\x02\x03\x05\x04”
```

By using the LCS algorithm, we can get the instructions that we must add or remove, as shown in figure 7, in the first string (our first basic block) in order to find the differences between two basic blocks. If we get the previous strings, it is possible to apply the LCS algorithm:

```
In  :  a = ”\x00\x01\x02\x03\x00\x04”
In  :  b = ”\x00\x00\x02\x03\x05\x04”
In  :  z = LCS( a, b )

In  :  z
[[0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 1, 1, 1],
 [0, 1, 1, 1, 1, 1, 1],
 [0, 1, 1, 2, 2, 2, 2],
 [0, 1, 1, 2, 3, 3, 3],
 [0, 1, 2, 2, 3, 3, 3],
 [0, 1, 2, 2, 3, 3, 4]]
```

We have a matrix with the different LCS, where the longest LCS has a length of four:

```
”\x00\x02\x03\x04”
```

This matrix is applied to the elements to add or to remove:

```
In  :  l_a = []  l_r = []
In  :  getDiff( z, a, b, len(a), len(b), l_a, l_r )

  00
– 01
+ 00
  02
  03
– 00
+ 05
  04

In [11]:  l_a
Out[11]:  [(1, '\x00'), (4, '\x05')]

In [12]:  l_r
Out[12]:  [(1, '\x01'), (4, '\x00')]
```

In this example, we must do the following actions in the first basic block to get the second one :

- add : ADD 3 (position 1), MUL 4 (position 4)
- remove : ADD 1 (position 1), ADD 3 (position 4)

3.2 Visualization

The visualization is an important element during the analysis of differences between two applications, because it is with this tool that the information is recovered. Current tools display differences with two CFGs (see figure 8), but it is possible to merge [8] two CFGs to have only one as shown in figure 9.

We can improve the visualization by using a simple graphical chart:

- green for added instructions,
- red for deleted instructions.

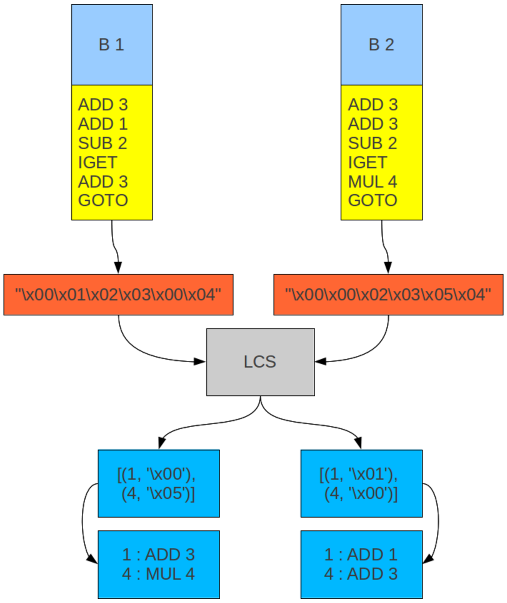

B 1
ADD 3
ADD 1
SUB 2
IGET
ADD 3
GOTO
B 2
ADD 3
ADD 3
SUB 2
IGET
MUL 4
GOTO
"\x00\x01\x02\x03\x00\x04"
"\x00\x00\x02\x03\x05\x04"
LCS
[(1, '\x00'), (4, '\x05')]
1 : ADD 3
4 : MUL 4
[(1, '\x01'), (4, '\x00')]
1 : ADD 1
4 : ADD 3

Figure 7: Find added/removed instructions from a basic block

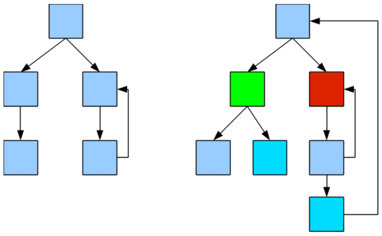


Figure 8: Visualization of differences by using two CFGs

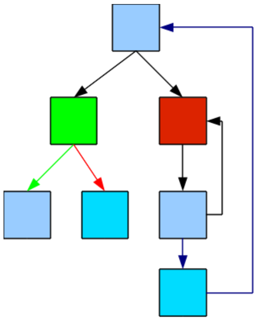


Figure 9: Visualization of differences by using one CFG

Jump instructions are a major problem when we use only one CFG to display differences, for example : ”IF X, Y [B-0x90, B-0x60]” which is a jump to 0x90 or 0x60 offsets (after the evaluation of the instruction) can be changed to : ”IF X, Z [B-0x80, B-0x50]” with a jump to old or new basic blocks. If the block 0x50 is the same as 0x60 but the block 0x80 is new, we need to change the display with correct differences :

- IF X, Y [B-0x90, B-0x60]
- IF X, Z [B-0x80, B-0x60]

We need to resolve identical or new basic blocks at different offsets (our algorithm is not dependent of the position of the basic block). We obtain the algorithm 11 which is the new CFG (with original, new and different basic blocks) as shown in figure 10.

Algorithm 11 Algorithm of matching exactly or partially basic blocks between two methods

```
Require: origbb, diffbb, newbb
  finalcfg ← []
  for b1 ← origbb.basicblocks() do
    if b1.name not in diffbb then
      b1.tag ← ORIG
      Append( finalcfg, b1 )
    else
      b1.tag ← DIFF
      Append( finalcfg, diffbb[ b1.name ] )
    end if
  end for
  for b1 ← newbb.basicblocks() do
    b1.tag ← NEW
    Append( finalcfg, b1 )
  end for
  finalcfg.sort()
```

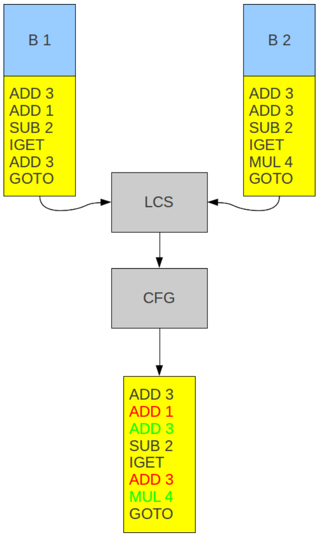


Figure 10: Differences and visualization between 2 basic blocks

3.3 Practical tests

Skype The 15th April 2011, AndroidPolice [6] released a new security vulnerability in Skype (version 1.0.0.831) for Android. This vulnerability exposes the users' name, phone number, and chat logs to all installed applications. The security bug is very simple, it is an incorrect usage of permissions to open files [6].

So, it is possible for another application to access to all information of your skype account, like account balance, full name, date of birth, city/state/country, home phone, office phone, cell phone, email addresses, your webpage, your bio, instant messages. A few days after this vulnerability, Skype [32] release a new version (1.0.0.983) which fixed this security bug. It is a good example to test our algorithm in a real case like Skype because this application is not small and it is an interesting case of reverse engineering. We can identify how many functions are :

- exactly identical: 8038,
- partialy identical: 165,
- new: 14,
- delete: 7.

We analyzed the 165 methods, by searching methods related to file permissions, by using the Java API or directly with chmod program. Inside the 165 methods, most of them are related to simple constant modification but we can identify a method really close to another one (with the same name) which manipulate files:

```
Lcom/skype/ipc/SkypeKitRunner; run ()V with Lcom/skype/ipc/SkypeKitRunner; run ()V
    0.269383959472
```

This method has four modified basic blocks, but only three basic blocks merit further investigation.

An integer value (it is the operating mode) of the method *openFileOutput* has been changed from 3 to 0 (listing 7) :

```
public abstract FileOutputStream openFileOutput (String name, int mode)
```

where 3 and 0 are respectively :

- MODE_WORLD_READABLE (allow all other applications to have read access to the created file) and MODE_WORLD_WRITEABLE (allow all other applications to have write access to the created file),
- MODE_PRIVATE (the default mode, where the created file can only be accessed by the calling application (or all applications sharing the same user ID)).

Listing 7: SkyperKitRunner class, run method: change mode of openFileOutput

```
DIFF run-BB@0x316 :
[...]
220(324) const-string v7 , [string@ 2998 'csf']
221(328)  + const/4 v8 , [#+ 0] , {0}
222(328)  - const/4 v8 , [#+ 3] , {3}
223(328) invoke-virtual v5 , v7 , v8 , [meth@ 120 Landroid/content/Context; (Ljava/lang/
    String; I) Ljava/io/FileOutputStream; openFileOutput]
[...]
```

In another basic block, the first argument of chmod has been changed (listing 8) from 777 to 750 :

- RWX, RWX, RWX
- RWX, R-X, —

Listing 8: SkyperKitRunner class, run method: change argument of chmod

```
DIFF run-BB@0x348 :
229(346) invoke-static [meth@ 5805 Ljava/lang/Runtime; () Ljava/lang/Runtime; getRuntime]
230(34c) move-result-object v2
231(34e) new-instance v4 , [type@ 899 Ljava/lang/StringBuilder;]
232(352) invoke-direct v4 , [meth@ 5848 Ljava/lang/StringBuilder; () V <init>]
233(358) + const-string v5 , [string@ 2921 'chmod 750 ']
234(358) - const-string v5 , [string@ 2904 'chmod 777 ']
235(358) invoke-virtual v4 , v5 , [meth@ 5855 Ljava/lang/StringBuilder; (Ljava/lang/String;)
     Ljava/lang/StringBuilder; append]
236(35e) move-result-object v4
237(360) invoke-virtual v3 , [meth@ 5719 Ljava/io/File; () Ljava/lang/String;
    getCanonicalPath]
```

And in the last modified basic block, there is a new call (listing 9) to a new method which fixes all files in the context directory of the application :

```
Lcom/skype/ipc/SkypeKitRunner; ([Ljava/io/File;) V fixPermissions]
```

which fixes all permissions (patch permissions from the previous version) to:

- RWX — — for a directory,
- RW- — — for a file.

Listing 9: SkyperKitRunner class, run method : call new method on files

```
417(5c8)  + move-object/from16 v0 , v19
418(5c8) invoke-virtual v4 , v3 , v2 , v5 , [meth@ 5804 Ljava/lang/Runtime; (Ljava/lang/
    String; [Ljava/lang/String; Ljava/io/File;) Ljava/lang/Process; exec]
419(5ce)  + move-object v1 , v4
420(5ce) move-result-object v2
421(5d0)  + invoke-direct v0 , v1 , [meth@ 1923 Lcom/skype/ipc/SkypeKitRunner; ([Ljava/io/
    File;) V fixPermissions]
```

4 Decompilation

The Dalvik virtual machine is a register-based machine which executes Dalvik bytecode instructions. Its working is different from a JVM, so its bytecode is different from Java bytecode. Hence, Java decompilers cannot decompile it.

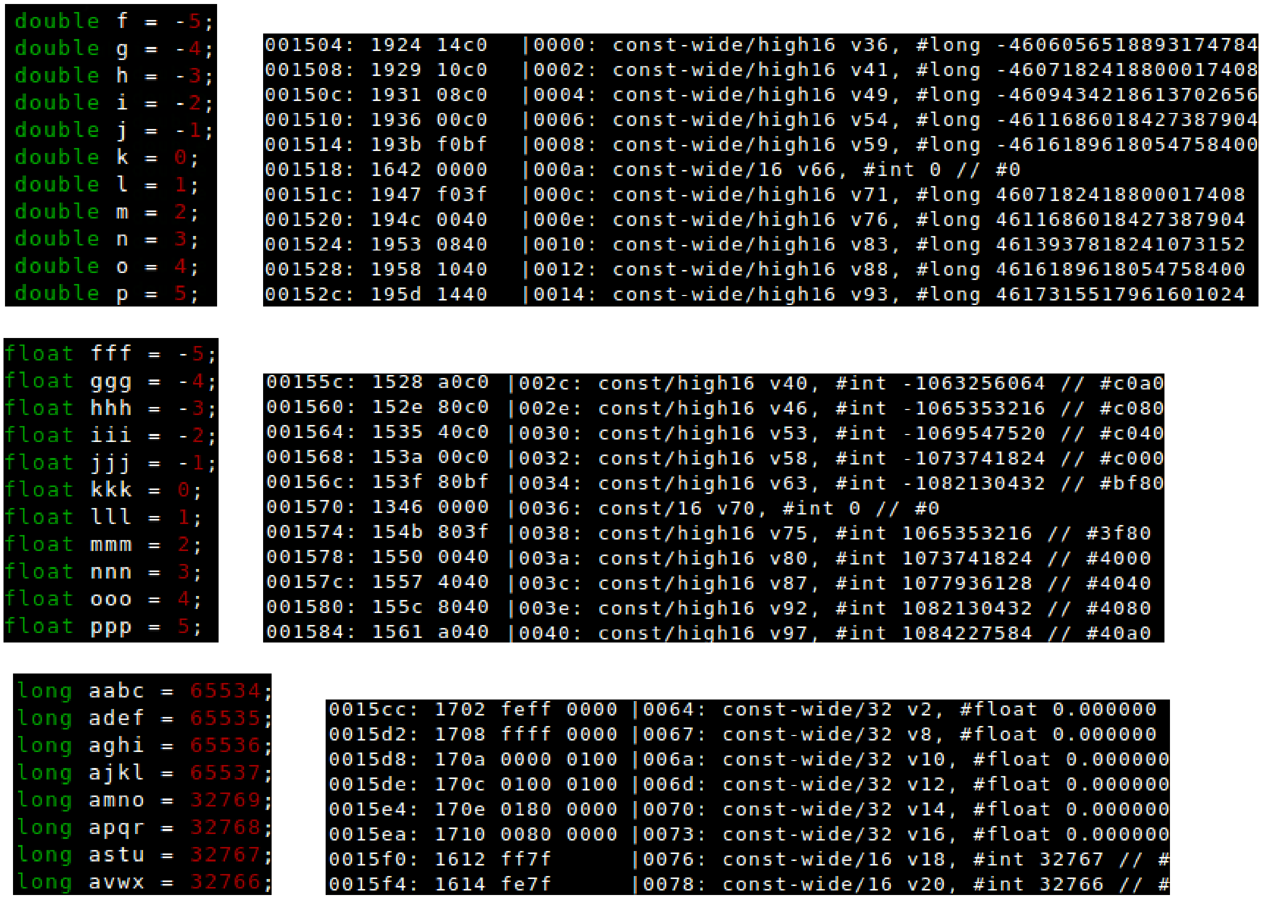
To overcome this issue, the common technique used by researchers who wants to take a look at the source of an Android application is to use bytecode-to-bytecode compilers like ded[28] or dex2jar[30] to transform the Dalvik bytecode into Java bytecode and then to use a regular Java decompiler (jd-gui[15], dava[27]. . .) on top of it. Such a method is prone to errors that may exist in the conversion of the bytecode, resulting in an erroneous source code.

Other analysts just use a disassembler [19, 21, 17]. We found that some of these disassemblers make mistakes which could be used to hide information to an analyst who would use the result of the disassembly. For example, dexdump does not correctly recover numbers, while IDA and baksmali doesn't even try although they give different outputs.

To be able to decompile an application, we first need to disassemble its bytecode. This first step is performed by androguard. Due to the nature of the bytecode (which contains some high-level information) we have access to classes, fields and methods of the application along with their type and name.

Once the disassembly process is done, we can start the decompilation. There is two main challenges that we have to face :

- the first one is data-flow analysis,



```
double f = -5;
double g = -4;
double h = -3;
double i = -2;
double j = -1;
double k = 0;
double l = 1;
double m = 2;
double n = 3;
double o = 4;
double p = 5;
```

```
001504: 1924 14c0  |0000: const-wide/high16 v36, #long -4606056518893174784
001508: 1929 10c0  |0002: const-wide/high16 v41, #long -4607182418800017408
00150c: 1931 08c0  |0004: const-wide/high16 v49, #long -4609434218613702656
001510: 1936 00c0  |0006: const-wide/high16 v54, #long -4611686018427387904
001514: 193b f0bf  |0008: const-wide/high16 v59, #long -4616189618054758400
001518: 1642 0000  |000a: const-wide/16 v66, #int 0 // #0
00151c: 1947 f03f  |000c: const-wide/high16 v71, #long 4607182418800017408
001520: 194c 0040  |000e: const-wide/high16 v76, #long 4611686018427387904
001524: 1953 0840  |0010: const-wide/high16 v83, #long 4613937818241073152
001528: 1958 1040  |0012: const-wide/high16 v88, #long 4616189618054758400
00152c: 195d 1440  |0014: const-wide/high16 v93, #long 4617315517961601024
```

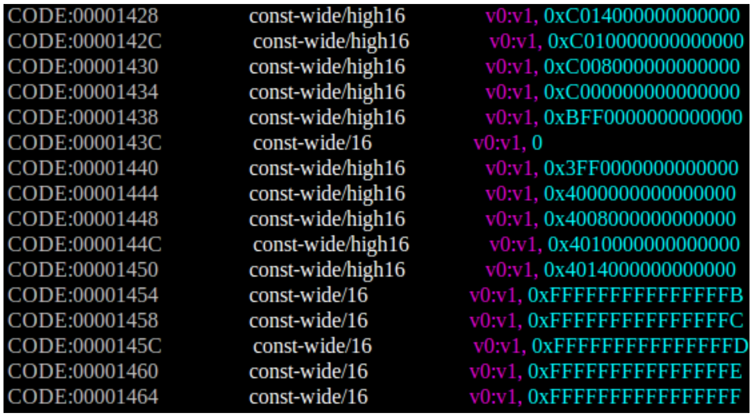
```
float fff = -5;
float ggg = -4;
float hhh = -3;
float iii = -2;
float jjj = -1;
float kkk = 0;
float lll = 1;
float mmm = 2;
float nnn = 3;
float ooo = 4;
float ppp = 5;
```

```
00155c: 1528 a0c0 |002c: const/high16 v40, #int -1063256064 // #c0a0
001560: 152e 80c0 |002e: const/high16 v46, #int -1065353216 // #c080
001564: 1535 40c0 |0030: const/high16 v53, #int -1069547520 // #c040
001568: 153a 00c0 |0032: const/high16 v58, #int -1073741824 // #c000
00156c: 153f 80bf |0034: const/high16 v63, #int -1082130432 // #bf80
001570: 1346 0000 |0036: const/16 v70, #int 0 // #0
001574: 154b 803f |0038: const/high16 v75, #int 1065353216 // #3f80
001578: 1550 0040 |003a: const/high16 v80, #int 1073741824 // #4000
00157c: 1557 4040 |003c: const/high16 v87, #int 1077936128 // #4040
001580: 155c 8040 |003e: const/high16 v92, #int 1082130432 // #4080
001584: 1561 a040 |0040: const/high16 v97, #int 1084227584 // #40a0
```

```
long aabc = 65534;
long adef = 65535;
long aghi = 65536;
long ajkl = 65537;
long amno = 32769;
long apqr = 32768;
long astu = 32767;
long avwx = 32766;
```

```
0015cc: 1702 feff 0000 |0064: const-wide/32 v2, #float 0.000000
0015d2: 1708 ffff 0000 |0067: const-wide/32 v8, #float 0.000000
0015d8: 170a 0000 0100 |006a: const-wide/32 v10, #float 0.000000
0015de: 170c 0100 0100 |006d: const-wide/32 v12, #float 0.000000
0015e4: 170e 0180 0000 |0070: const-wide/32 v14, #float 0.000000
0015ea: 1710 0080 0000 |0073: const-wide/32 v16, #float 0.000000
0015f0: 1612 ff7f      |0076: const-wide/16 v18, #int 32767 // #
0015f4: 1614 fe7f      |0078: const-wide/16 v20, #int 32766 // #
```

Figure 11: Source & dexdump output

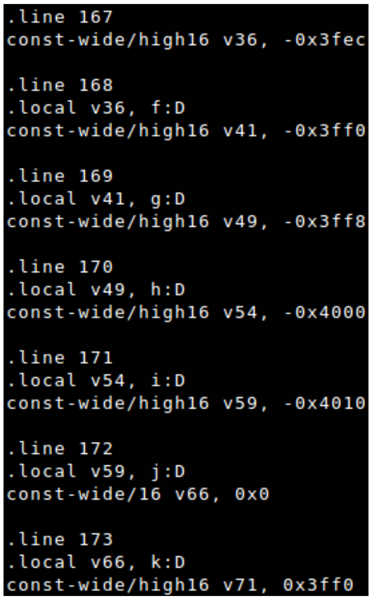


```
CODE:00001428       const-wide/high16      v0:v1, 0xC014000000000000
CODE:0000142C       const-wide/high16      v0:v1, 0xC010000000000000
CODE:00001430       const-wide/high16      v0:v1, 0xC008000000000000
CODE:00001434       const-wide/high16      v0:v1, 0xC000000000000000
CODE:00001438       const-wide/high16      v0:v1, 0xBFF0000000000000
CODE:0000143C       const-wide/16          v0:v1, 0
CODE:00001440       const-wide/high16      v0:v1, 0x3FF0000000000000
CODE:00001444       const-wide/high16      v0:v1, 0x4000000000000000
CODE:00001448       const-wide/high16      v0:v1, 0x4008000000000000
CODE:0000144C       const-wide/high16      v0:v1, 0x4010000000000000
CODE:00001450       const-wide/high16      v0:v1, 0x4014000000000000
CODE:00001454       const-wide/16          v0:v1, 0xFFFFFFFFFFFFFFFB
CODE:00001458       const-wide/16          v0:v1, 0xFFFFFFFFFFFFFFFC
CODE:0000145C       const-wide/16          v0:v1, 0xFFFFFFFFFFFFFFFD
CODE:00001460       const-wide/16          v0:v1, 0xFFFFFFFFFFFFFFFE
CODE:00001464       const-wide/16          v0:v1, 0xFFFFFFFFFFFFFFFF
```

Figure 12: IDA disassembly

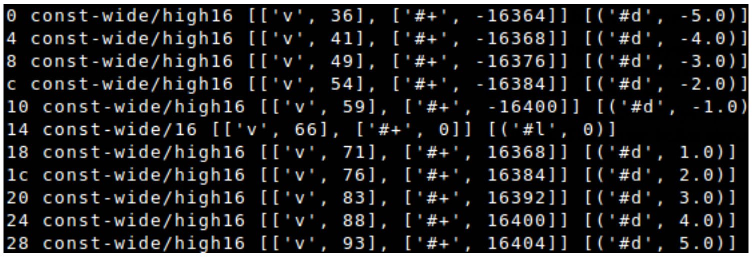
- the second one is control-flow analysis.

The data flow analysis goal is to improve the intermediate representation built from the bytecode. In order to do this, compilations techniques are used on the IR, such as constant propagation, common subexpressoin



```
.line 167
const-wide/high16 v36, -0x3fec

.line 168
.local v36, f:D
const-wide/high16 v41, -0x3ff0

.line 169
.local v41, g:D
const-wide/high16 v49, -0x3ff8

.line 170
.local v49, h:D
const-wide/high16 v54, -0x4000

.line 171
.local v54, i:D
const-wide/high16 v59, -0x4010

.line 172
.local v59, j:D
const-wide/16 v66, 0x0

.line 173
.local v66, k:D
const-wide/high16 v71, 0x3ff0
```

Figure 13: Baksmali disassembly



```
0 const-wide/high16 [['v', 36], ['#+', -16364]] [('#d', -5.0)]
4 const-wide/high16 [['v', 41], ['#+', -16368]] [('#d', -4.0)]
8 const-wide/high16 [['v', 49], ['#+', -16376]] [('#d', -3.0)]
c const-wide/high16 [['v', 54], ['#+', -16384]] [('#d', -2.0)]
10 const-wide/high16 [['v', 59], ['#+', -16400]] [('#d', -1.0)
14 const-wide/16 [['v', 66], ['#+', 0]] [('#l', 0)]
18 const-wide/high16 [['v', 71], ['#+', 16368]] [('#d', 1.0)]
1c const-wide/high16 [['v', 76], ['#+', 16384]] [('#d', 2.0)]
20 const-wide/high16 [['v', 83], ['#+', 16392]] [('#d', 3.0)]
24 const-wide/high16 [['v', 88], ['#+', 16400]] [('#d', 4.0)]
28 const-wide/high16 [['v', 93], ['#+', 16404]] [('#d', 5.0)]
```

Figure 14: androguard disassembly

elimination, dead code elimination,. . .

The control flow analysis goal is to structure the control flow graph of our methods. This phase modifies the graph to identify structures like loops and conditions.

4.1 DVM representation

We chose to represent an android application by building an abstraction of the Dalvik machine. Our DVM object contains all the classes of the android application. Classes are represented by a an object which contains all the information of a class : visibility, type, name, methods,. . .

Methods are also abstracted by an object. As with the classes, methods objects contain all the information of a method : visibility, type, name, parameters, . . . They are also given a memory, which is used to perform some data flow analysis. Methods also contains the representation of the bytecode instructions.

Hence, we are able to decompile only one method of an application, and, if we set the memory right, we can decompile only a single bytecode instruction. This can be use to do some debugging of the application.

4.2 Data and Control flow

An internal memory state is assigned to each methods. This memory is used to represent the registers of the Dalvik virtual machine. The first thing we do to be able to do some data flow analysis is to construct the control flow graph of the method. This graph is constructed by partitioning the bytecode instructions of the method into basic blocks. A basic block is a sequence of instruction which has only one entry point and one exit point.

When building the CFG, exceptions have to be treated a special way. Due to the nature of exceptions, their basic blocks do not have parents. Indeed, there is no derivation of the control flow to an exception(Figure 15), because an exception is triggered when executing. So for each basic blocks we need to check if the block may trigger an exception, and if it is the case, we modify our graph accordingly(Figure 16).

After having built the CFG, we can perform data flow analysis as presented in [10]. These analysis transform the intermediate representation in order to propagate information accross different blocks and optimize the result of the decompilation.

Once the data flow analysis and optimizations are done, we can perform an analysis of the CFG to recover higher structures of control like conditionnal statements, loops,. . . To do such analysis, we calculate dominators and intervales of the graph as presented in [10]

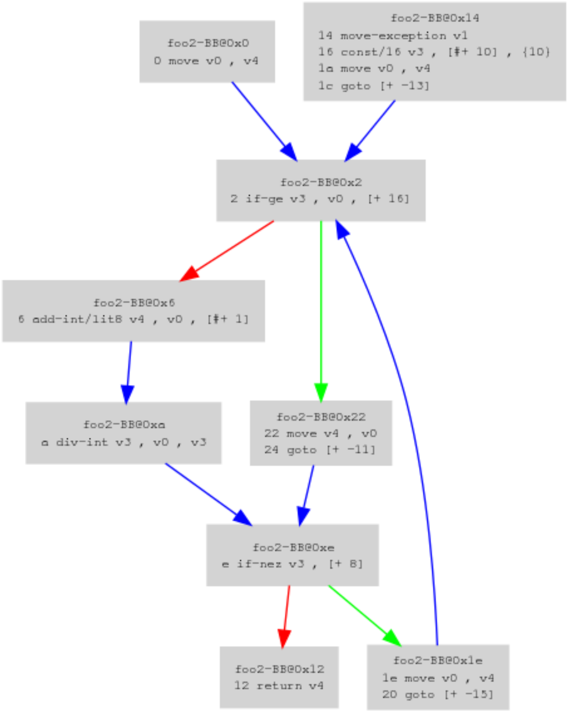


Figure 15: Basic blocks with exception

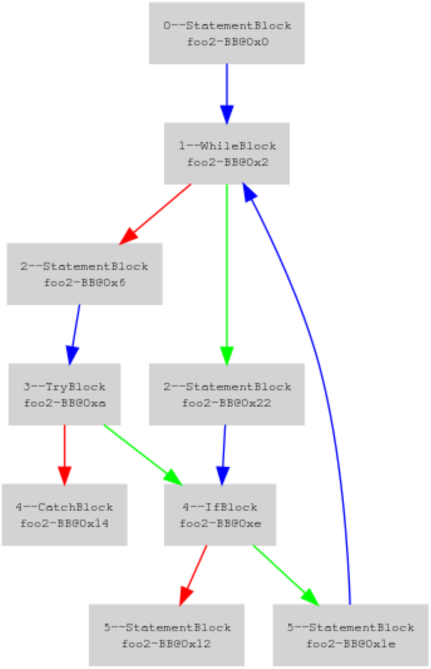


Figure 16: CFG with exception

The next step is to build an AST from the graph in order to remove cycles and to be able to perform some other optimizations.

4.3 Writing/Executing source

To write the source of a method, the prototype (visibility, type, name. . .) of the method is written, the AST built previously is traversed and each node of the AST print its representation. To write the source of a class, we write its prototype and we write the source of all its methods.

Currently, no optimization is done to recover the value of the attributes of a class, so each method's constructors initalize the attributes, in the same way it is done at the bytecode level.

5 Conclusion and Future Work

In the first part of the paper we presented several algorithms to compare applications to identify their similarities or differences. To do that we applied an original selected compressor (Snappy) obtain a real usable tool to improve the time of comparison with good results. With this similarity distance, we created a tool to determine whether a version of application has potentially been pirated. Next we applied this technique to design a tool to measure the efficiency of an obfuscator, and we demonstrated that there is a lack of proven tools in this domain. Then we described a new algorithm to find and visualize dissimilarities between versions of an application.

While this paper demonstrate progress in this area, we need to continue investigating this domain to compare one application against multiple applications by using clustering. An additional area which merits additional research relates to the possible use of similarity distance to create a database of applications [13] to check whether an identified application is present in a large set of applications. While preliminary findings are promising, there remains much work to be done in this important area.

In the last part of the paper we presented a problem we found in disassemblers (even in the one provided by Google) during the implementation of our decompiler, concerning the recovery of numbers. Then we described different phases which are generally done in a decompiler. We implemented some of them in order to have a "minimal" working decompiler, that is to say, whose output can be compiled. Some optimisation phases still need to be done, mostly to increase the readability of the output, but also to handle special constructs of the language. The decompiler is built in such a way that it can be used to reverse parts of an application with the source code generated on the fly. It can also be used to do basic debugging, by extending the operations of our intermediate representation.

The tools and the framework are open source and can be downloaded on the website [13].

References

[1] http://en.wikipedia.org/wiki/longest_common_subsequence_problem.

[2] http://www.7-zip.org/.

[3] http://www.bzip.org/.

[4] G. Gueguen A. Desnos. Android malwares : is it a dream ? *EICAR*, 2011.

[5] R. Erra A. Desnos, B. Caillat. Binthavro: towards a useful and fast tool for goodware and malware analysis. *EICAR*, 2010.

[6] AndroidPolice. http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/.

[7] Brutall. http://code.google.com/p/android-apktool/.

[8] Aureliano Calvo. Showing differences between disassembled functions. In *Hack.lu*, 2010.

[9] Silvio Cesare. Classification of malware using structured control flow. In *AusPDC*, 2010.

[10] Cristina Cifuentes. Reverse compilation techniques, 1994.

[11] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[12] Lasse Collin. http://tukaani.org/xz/.

[13] Anthony Desnos. http://code.google.com/p/androguard/.

[14] Emmanuel Dupuy. http://java.decompiler.free.fr/?q=jdgui.

[15] Emmanuel Dupuy. http://java.decompiler.free.fr/?q=jdgui.

[16] C. Eagle. *The IDA PRO Book*. No Starch Press, 2008.

[17] Jesus Freke. http://code.google.com/p/smali/.

[18] Google. http://code.google.com/p/snappy/.

[19] Google. http://developer.android.com/sdk/index.html.

[20] Google. http://www.android.com/.

[21] Hex-Rays. http://www.hex-rays.com/products/ida/index.shtml.

[22] Don Kellogg. http://blog.nielsen.com/nielsenwire/?p=28628.

[23] Pavel Kouznetsov. http://www.varaneckas.com/jad.

[24] Eric Lafortune. http://proguard.sourceforge.net/.

[25] Jean loup Gailly and Mark Adler. http://www.zlib.net/.

[26] McAfee. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf.

[27] Nomair A. Naeem and Laurie Hendren. Programmer-friendly decompiled java. Technical Report SABLE-TR-2006-2, Sable Research Group, McGill University, March 2006.

[28] Damien Octeau, William Enck, and Patrick McDaniel. The ded Decompiler. Technical Report NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2010.

[29] Oracle. http://www.java.com/.

[30] Panxiaobo. http://code.google.com/p/dex2jar/.

[31] Andy Rubin. https://twitter.com/#!/arubin/status/85660213478309888.

[32] Skype. http://blogs.skype.com/security/2011/04/privacy_vulnerability_in_skype_1.html.

[33] PreEmptive Solutions. http://www.preemptive.com/products/dasho.