Static-analysis techniques of malware reverse engineering

Loïc Branstett – Algebra University College

Abstract

Network and system security are incredibly critical issues now. Due to the rapid proliferation of malware, traditional analysis methods struggle with enormous samples. In this paper, we will expose different static-analysis techniques to reverse engineering executable and determine if those are malware.

Keywords: Static-analysis; Malware; Reverse engineering; Elf; AI; Dynamic-analysis

1. Introduction

Nowadays malware threats were assessed by IT security organizations have been increasing by more than ten thousand every day. Symantec Internet Security Threat Report reveals that the total number of unique variants of malware in the world in 2011 amounts to around 403 million compared to 286 million variants in 2010. Use usage of avoidance techniques such as self-defending code, packing, anti-debugging and anti-virtualization techniques has a lead to more and more malware passing the detection barriers of anti-virus and researcher, this has increased the potentials entry for criminals inside corporation and individual computers. Furthermore most challenging for antivirus organization and researcher is about the threat that occurs in computer applications because of the unknown vulnerability or known as a zero-day attack. These attacks take advantage of an application vulnerability.

To defend against these malwares, analysts use different types of analysis: Static-analysis, Dynamic-analysis and Hybrid-analysis.

1.1. Static Analysis

In static analysis, malware characteristics can be collected without execution of the code itself. For Portable Executable (PE) files, which are native to Windows, analysts often use tools that can present tabular views of PE header information, disassemble machine language, extract printable strings, file hashes, etc. Some features commonly investigated as part of static analysis include file hashes, strings, op code sequences, DLL imports, API calls, and other metadata found in the PE header (6). In malware visualization, researchers propose to view the contents of software in raw numerical form, such as binary, decimal, or hexadecimal, and convert these strings of numbers into images.

1.2. Dynamic Analysis

In contrast to static analysis, dynamic analysis involves execution of the malicious code. While this is much harder to scale than static analysis, monitoring how a binary interacts with an infected system can lead to more insights. Analysts are often interested in recording API and system calls, processes, modifications to system files and registries, network communication, etc.

1.3. Hybrid Analysis

Both static and dynamic analysis have their own limitations when conducted individually. For example, packers that compress software can be used as an obfuscation tool to obscure contents of an executable. This will often necessitate the manual efforts of a malware analyst to conduct further static analysis on machine code. Dynamic analysis is not always successful since some types of malwares require a certain duration to pass or a specific event to trigger its execution. Hybrid analysis, the combination of both static and dynamic analysis, can lead to more comprehensive views of malicious programs and enable researchers to gather a larger set of features for classification.

2. Background

2.1. Machine Code and Assembly Languages

Machine code [1], also known as machine language, is the elemental language of computers. It is read by the computer's central processing unit (CPU), is composed of digital binary numbers and looks like a very long sequence of zeros and ones. Ultimately, the source code of every human-readable programming language must be translated to machine language by a compiler or an interpreter, because binary code is the only language that computer hardware can understand.

This transformation is generally called compilation, it transforms source code to the assembly language understood by the CPU, this assembly language is then itself transformed into a series of machine instructions (aka machine code).

However static-analysis generally require a higher level of understanding of the code and use a technique known as disassembly; the process of transforming machine code to a control flow graph of machine instructions, that can itself be transform to a very low language like C.

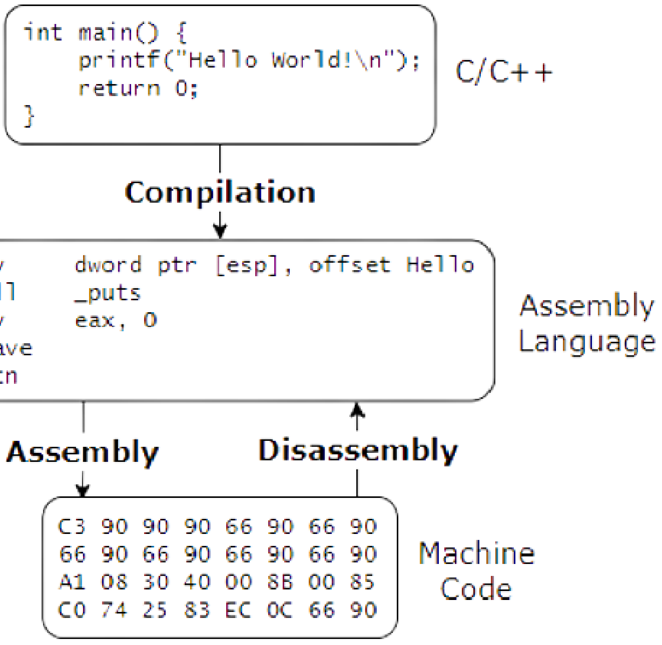

```
int main() {
    printf("Hello World!\n");
    return 0;
}
```
```
mov     dword ptr [esp], offset Hello
call    _puts
mov     eax, 0
leave
retn
```
```
C3 90 90 90 66 90 66 90
66 90 66 90 66 90 66 90
A1 08 30 40 00 8B 00 85
C0 74 25 83 EC 0C 66 90
```

Illustration 1: Compilation and assembly, disassembly

2.2. Code Obfuscation and Encryption

Since executable files can be analyzed by disassembly, new techniques called anti-reverse-engineering have been invented to obstruct the process. Not only malware authors, but also software companies use them to protect commercial software from being cracked or pirated. Code obfuscation [2] and encryption are two commonly used methods.

One commonly technique used in code obfuscation is Control flow obfuscation. It synthesizes conditional, branching, and iterative constructs that produce valid executable logic, but yields non-deterministic semantic results when de-compiled.

In a managed executable, all strings are clearly discoverable and readable. Even when methods and variables are renamed, strings can be used to locate critical code sections by looking for string references inside the binary. This includes messages (especially error messages) that are displayed to the user. To provide an effective barrier against this type of attack, string encryption hides strings in the executable and only restores their original value when needed.

Invalid Authentication - Try Again => !ù$àç_èè-'("'=):'")'

A more extreme version of string obfuscation is complete code encryption.

Unlike obfuscation, code encryption packs and encrypts executable files on the disk. They will decrypt themselves during execution. It means they are nearly impossible to analyze just by static disassembly relying instead on execution and reviewing of system logs.

2.3. Executable format

Executable formats [3] encapsulate system-related information, such as API export and import tables, resources (icons, images and audio, etc.), and the distribution of data and code. This information is critical for malware analysis. Data and executable code are stored in different sections behind headers, depending on their functions.

2.4. Portable Executable

Portable Executable (PE) [4] format is a file format for executable, object code, DLLs and others used in 32-bit and 64-bit versions of Windows operating systems.

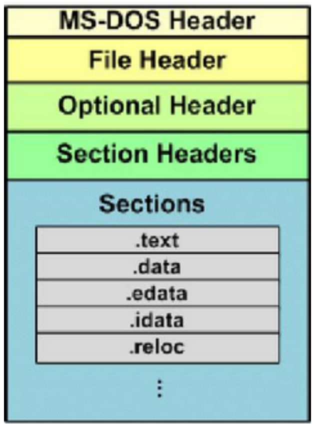


Illustration 2: Portable Executable : Arcichecture

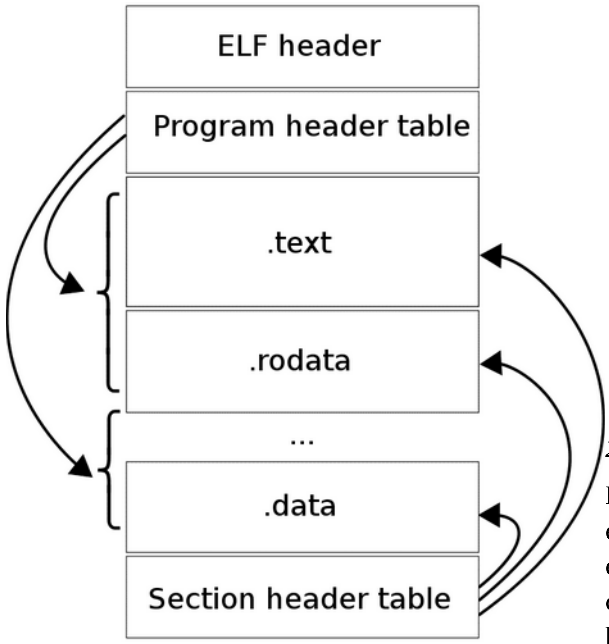


Illustration 3: ELF

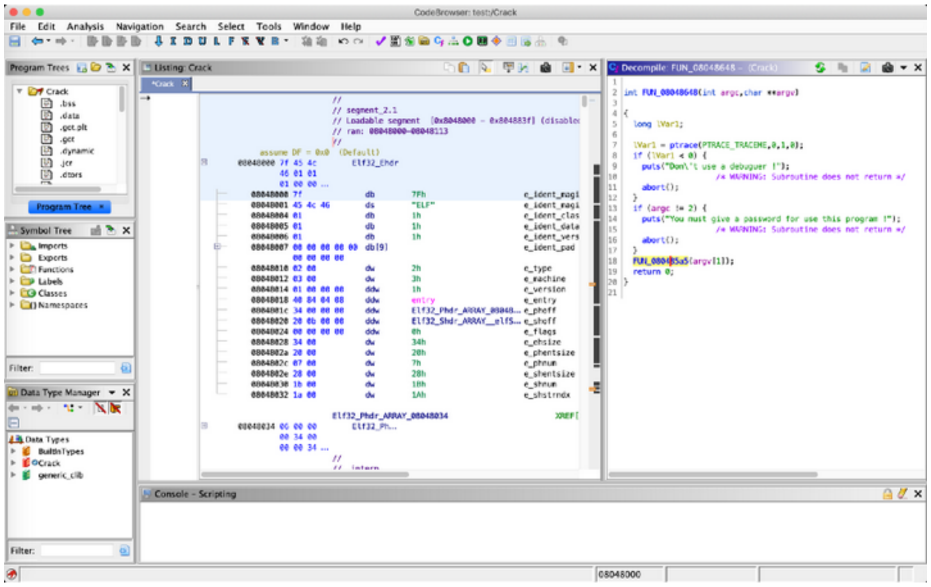
2.5. Executable and Linkable Format

Executable and Linkable Format (ELF) format is flexible, extensible, and cross-platform. For instance, it supports different endianness and address sizes so it does not exclude any particular central processing unit (CPU) or instruction set architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms.

3. Manual analysis

In this stage, analysts reverse-engineer code using debuggers, disassemblers, compilers, and specialized tools to decode encrypted data, determine the logic behind the malware algorithm and understand any hidden capabilities that the malware has not yet exhibited. Code reversing is a rare skill, and executing code reversals takes a great deal of time. For these reasons, malware investigations often skip this step and therefore miss out on a lot of valuable insights into the nature of the malware.

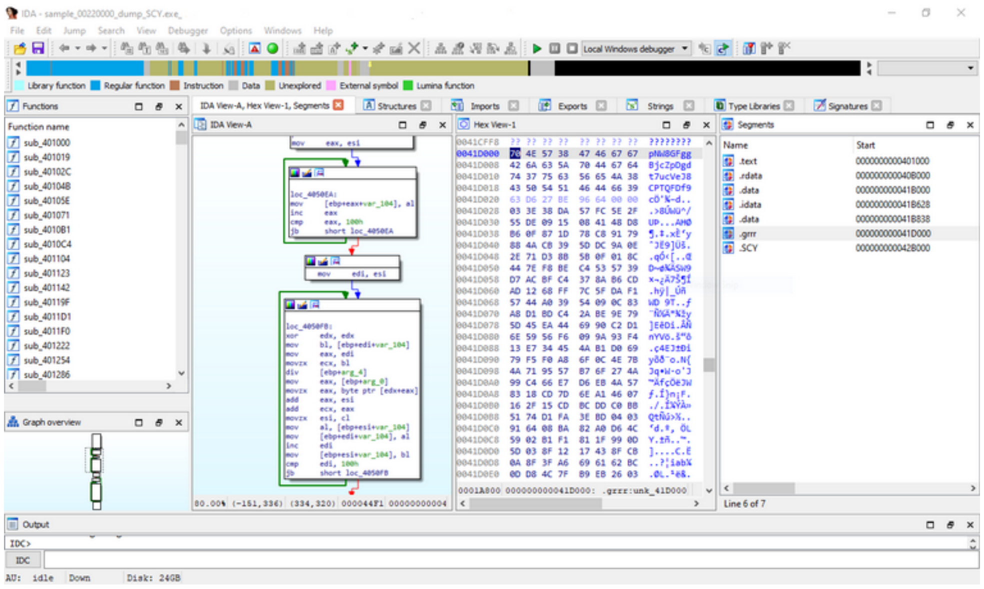
In order to manually reverse the code, malware analysis tools such as a debugger and disassembler are needed. The skills needed to complete manual code reversing are very important, but also difficult to find.



```
    //
    // segment_2.1
    // Loadable segment  [0x8048000 - 0x804883f] (disable
    // ram: 08048000-08048113
    //
        assume DF = 0x0  (Default)
    08048000 7f 45 4c        Elf32_Ehdr
             46 01 01
             01 00 00 ...
      08048000 7f            db     7Fh                   e_ident_magi
      08048001 45 4c 46      ds     "ELF"                 e_ident_magi
      08048004 01            db     1h                    e_ident_clas
      08048005 01            db     1h                    e_ident_data
      08048006 01            db     1h                    e_ident_vers
      08048007 00 00 00 00 00 db[9]                       e_ident_pad
               00 00 00 00
      08048010 02 00         dw     2h                    e_type
      08048012 03 00         dw     3h                    e_machine
      08048014 01 00 00 00   ddw    1h                    e_version
      08048018 40 84 04 08   ddw    entry                 e_entry
      0804801c 34 00 00 00   ddw    Elf32_Phdr_ARRAY_08048... e_phoff
      08048020 20 0b 00 00   ddw    Elf32_Shdr_ARRAY__elfS... e_shoff
      08048024 00 00 00 00   ddw    0h                    e_flags
      08048028 34 00         dw     34h                   e_ehsize
      0804802a 20 00         dw     20h                   e_phentsize
      0804802c 07 00         dw     7h                    e_phnum
      0804802e 28 00         dw     28h                   e_shentsize
      08048030 1b 00         dw     1Bh                   e_shnum
      08048032 1a 00         dw     1Ah                   e_shstrndx

                    Elf32_Phdr_ARRAY_08048034                    XREF[
    08048034 06 00 00        Elf32_Ph...
             00 34 00
             00 00 34 ...
    //
    //  intern
```

```
int FUN_08048648(int argc,char **argv)

{
  long lVar1;

  lVar1 = ptrace(PTRACE_TRACEME,0,1,0);
  if (lVar1 < 0) {
    puts("Don\'t use a debuguer !");
                    /* WARNING: Subroutine does not return */
    abort();
  }
  if (argc != 2) {
    puts("You must give a password for use this program !");
                    /* WARNING: Subroutine does not return */
    abort();
  }
  FUN_080485a5(argv[1]);
  return 0;
}
```

Illustration 4: Ghidra



```
mov     eax, esi

loc_4050EA:
mov     [ebp+eax+var_104], al
inc     eax
cmp     eax, 100h
jb      short loc_4050EA

mov     edi, esi

loc_4050F8:
xor     edx, edx
mov     bl, [ebp+edi+var_104]
mov     eax, edi
movzx   ecx, bl
div     [ebp+arg_4]
mov     eax, [ebp+arg_0]
movzx   eax, byte ptr [edx+eax]
add     eax, esi
add     ecx, eax
movzx   esi, cl
mov     al, [ebp+esi+var_104]
mov     [ebp+edi+var_104], al
inc     edi
mov     [ebp+esi+var_104], bl
cmp     edi, 100h
jb      short loc_4050F8
```

```
0041CFF8  ?? ?? ?? ??  ?? ?? ?? ??  ????????
0041D000  70 4E 57 38  47 46 67 67  pNW8GFgg
0041D008  42 6A 63 5A  70 44 67 64  BjcZpDgd
0041D010  74 37 75 63  56 65 4A 38  t7ucVeJ8
0041D018  43 50 54 51  46 44 66 39  CPTQFDf9
0041D020  63 D6 27 BE  96 64 00 00  cÖ'¾–d..
0041D028  03 3E 38 DA  57 FC 5E 2F  .>8ÚWü^/
0041D030  55 DE 09 15  08 41 48 D8  UÞ...AHØ
0041D038  B6 0F 87 1D  78 C8 91 79  ¶.‡.xÈ'y
0041D040  88 4A CB 39  5D DC 9A 0E  ˆJË9]Üš.
0041D048  2E 71 D3 8B  5B 0F 01 8C  .qÓ‹[..Œ
0041D050  44 7E F8 BE  C4 53 57 39  D~ø¾ÄSW9
0041D058  D7 AC BF C4  37 8A B6 CD  ×¬¿Ä7Š¶Í
0041D060  AD 12 68 FF  7C 5F DA F1  .hÿ|_Úñ
0041D068  57 44 A0 39  54 09 0C 83  WD 9T..ƒ
0041D070  A8 D1 BD C4  2A BE 9E 79  ¨Ñ½Ä*¾žy
0041D078  5D 45 EA 44  69 90 C2 D1  ]EêDi.ÂÑ
0041D080  6E 59 56 F6  09 9A 93 F4  nYVö.š"ô
0041D088  13 E7 34 45  4A B1 D0 69  .ç4EJ±Ði
0041D090  79 F5 F0 A8  6F 0C 4E 7B  yõð¨o.N{
0041D098  4A 71 95 57  B7 6F 27 4A  Jq•W·o'J
0041D0A0  99 C4 66 E7  D6 EB 4A 57  ™ÄfçÖëJW
0041D0A8  83 18 CD 7D  6E A1 46 07  ƒ.Í}n¡F.
0041D0B0  16 2F 15 CD  BC DD C0 BB  ./.Í¼ÝÀ»
0041D0B8  51 74 D1 FA  3E BD 04 03  QtÑú>½..
0041D0C0  91 64 08 BA  82 A0 D6 4C  'd.º‚ ÖL
0041D0C8  59 02 B1 F1  81 1F 99 0D  Y.±ñ..™.
0041D0D0  5D 03 8F 12  17 43 8F CB  ]....C.Ë
0041D0D8  0A 8F 3F A6  69 61 62 BC  ..?¦iab¼
0041D0E0  0D D8 4C 7F  B9 EB 26 03  .ØL.¹ë&.
0001A800 000000000041D000: .grrr:unk_41D000
```

Name	Start
.text	0000000000401000
.rdata	000000000040B000
.data	000000000041B000
.idata	000000000041B628
.data	000000000041B838
.grrr	000000000041D000
.SCY	000000000042B000

Illustration 5: IDA

4. Semi-automated analysis

Semi-automated analysis is a type of analysis that leaves only a small portion of the analysis to human.

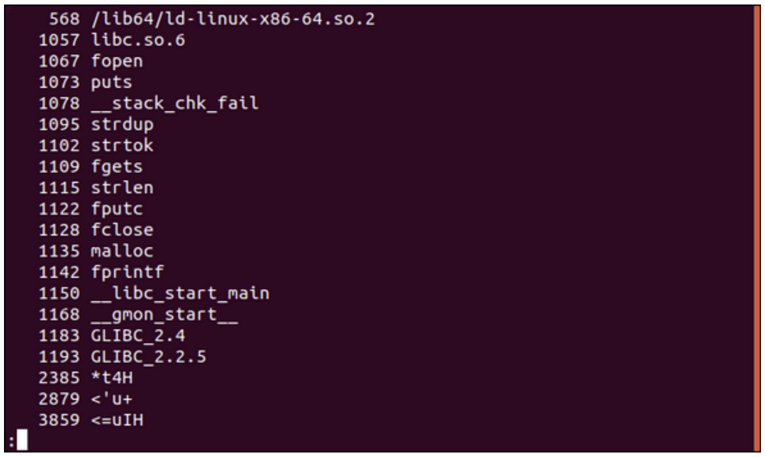
In `An Architecture for Semi-Automatic Collaborative Malware Analysis for CIs`, researcher presented an architecture that enables collaborative malware analysis by sharing resources (scripts, executable, ...) and leaving to human analysts only a small and manageable part of the whole effort.

5. Automated analysis

One of the simplest ways to assess a suspicious program is to scan it with fully-automated tools. Fully-automated tools are able to quickly assess what a malware is capable of if it infiltrated the system. This analysis is able to produce a detailed report regarding the network traffic, file activity, and registry keys. Even though a fully-automated analysis does not provide as much information as an analyst, it is still the fastest method to sift through large quantities of malware.

5.1. Using Equality

This "equality" technique uses information's such as string, hashes, ... and compares them to a preexisting database. Due to its simplicity it is one of the most used technique for a client like anti-malware solution.



```
 568 /lib64/ld-linux-x86-64.so.2
1057 libc.so.6
1067 fopen
1073 puts
1078 __stack_chk_fail
1095 strdup
1102 strtok
1109 fgets
1115 strlen
1122 fputc
1128 fclose
1135 malloc
1142 fprintf
1150 __libc_start_main
1168 __gmon_start__
1183 GLIBC_2.4
1193 GLIBC_2.2.5
2385 *t4H
2879 <'u+
3859 <=uIH
:
```

Illustration 6: Symbols of a executable

5.2. Using Machine Learning

5.2.1. Android-COCO

Android-COCO [5], is a supervised approach for detecting Android malware. It works by automatically transforming the byte and native code to program dependency graphs (PDG) and uses the embedding of these graphs to classify applications. Large-scale experiments on 100,113 samples (35,113 malware and 65,000 benign) show that this approach can detect malware applications with an accuracy of 99.86%, significantly outperforming state-of-the-art solutions.

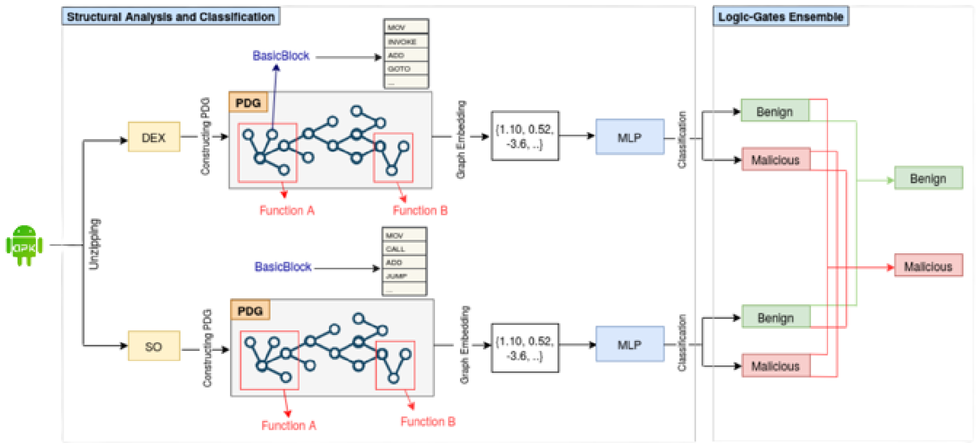


Illustration 7: Android-COCO Architecture

5.2.2. N-GRAM

Presented in `Malware Classification Using Static Disassembly and Machine Learning` [4] this novel technique uses an N-GRAM neuronal network with different inputs:

- File Size
- API
- Op Codes
- Dependencies
- Section Size
- Section Permissions
- Content complexity

The new approach features together with a classical machine learning algorithm (Random Forest) present very good accuracy at 99.40%. Higher than the previous 4-grams than only achieve 57.96% accuracy and require a more complex model.

6. Fighting limitations of static analysis

6.1. Lazy Loading

Lazy initialization/loading [6] is one of those design patterns which is in use in almost all programming languages. Its goal is to move the object's construction forward in time. It's especially handy when the creation of the object is expensive, and you want to defer it as late as possible, or even skip entirely. This implies that the object may not be precomputed in the data section like other data.

Fighting this is very hard and close to impossible. The only method who works reliably is to execute the TLS (Thread Local Storage) in a sand-boxed environment but this defeats the goal of static-analysis.

6.2. Name Mangling

Name mangling is a mechanism used by compilers to add additional characters to functions with the same name (function overloading). The goal of name mangling is to avoid any confusion when executing the program and calling a function that may have the same name as another one.

This c++ code:

```cpp
void add(int a, int b){cout << a + b << endl;}
```

Would result in these symbols inside the compiled library or binary:

```
$ nm a.out
[..]
00401382 T __Z3adddd
00401350 T __Z3addii
[..]
```

6.2.1. Jump Tables

A jump table can be either an array of pointers to functions or an array of machine code jump instructions. If you have a relatively static set of functions (such as system calls or virtual functions for a class) then you can create this table once and call the functions using a simple index into the array. This would mean retrieving the pointer and calling a function or jumping to the machine code depending on the type of table used.

The workaround used in static-analysis is to manually compute the address like the dynamic linker would do when executing the program.

7. Conclusion

We have explored in this paper different techniques from manually looking at the assembly code of a program to using artificial intelligence in order to provide an analysis of a potential malicious program. We believe that the future will be around automatic analysis with some degree of machine learning both in static analysis but also dynamic analysis.

8. Further Work

Similarity-based Android Malware Detection Using Hamming Distance of Static Binary Features [7]

Acknowledgments

We wish to thank Dublin City University and Algebra University College for having provided the resources and inspiration for this paper. As well as everyone in the security industry.

References

1: , Machine Code - Wikipedia, , https://en.wikipedia.org/wiki/Machine_code
2: , Obfuscation (software), , https://en.wikipedia.org/wiki/Obfuscation_(software)
3: , Executable formats, , https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats
4: , Malware Classification Using Static Disassembly and Machine Learning, , https://arxiv.org/abs/2201.07649
5: , Android-COCO: Android Malware Detection with Graph Neural Network for Byte- and Native-Code, , https://arxiv.org/abs/2112.10038
6: , Lazy Loading, , https://en.wikipedia.org/wiki/Lazy_loading
7: , Similarity-based Android Malware Detection Using Hamming Distance of Static Binary Features, , https://arxiv.org/abs/1908.05759