data preprocessing

Andreu Girones de la Fuente

December 2022

1 Extraction

The data has been extracted from VirusTotal. It consists of 1683 malicious ELF files.
Using *objdump* the target instruction set has been analysed for each one of the files.

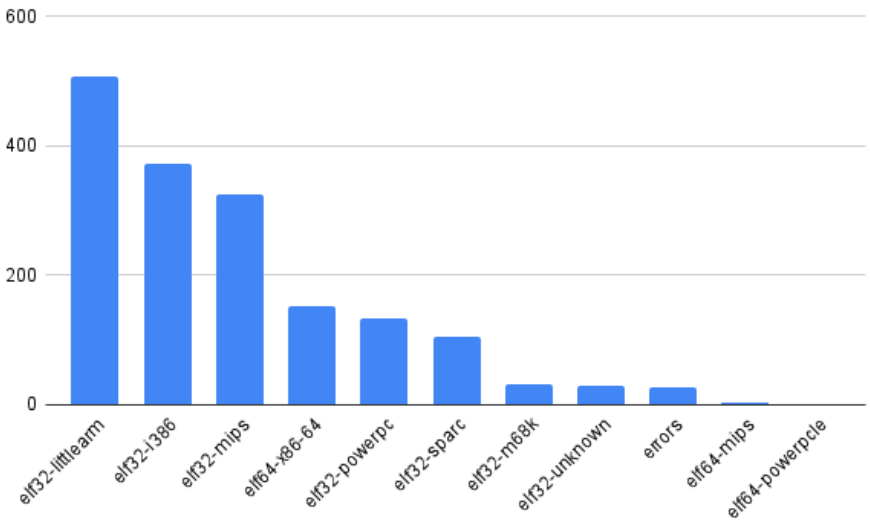

Figure 1: Occurrences of each ISA in the VirusTotal dataset.

Notice that 26 of the files *objdump* were not able to extract the target ISA.

As the number of files with errors is not significant, they will simply be discarded.

To extract the opcodes for each of the files, again, *objdump* is used. However, not all ISAs are supported by *objdump*. As it is shown in the table below.

```
section header table goes past the end of the file: e_shoff = 0x231768
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x4b050
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x23fa40
section table goes past the end of file
section header table goes past the end of the file: e_shoff = 0xebb64
section header table goes past the end of the file: e_shoff = 0xcd54
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0xcc4b8
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x29f1c0
section header table goes past the end of the file: e_shoff = 0xc705c
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x27b54
section header table goes past the end of the file: e_shoff = 0x23fa40
section header table goes past the end of the file: e_shoff = 0x18cf0
```

Figure 2: Errors shown when trying to extract the target ISA.

ISA	objdump supported
elf32-i386	yes
elf32-littlearm	yes
elf32-m68k	no
elf32-mips	no
elf32-powerpc	no
elf32-sparc	no
elf32-unknown	no
elf64-mips	no
elf64-powerpcle	no
elf64-x86-64	yes

The elf64-x86-64 files had been chosen to extract the opcodes. Here is an image that shows the output of the *objdump -d* command.

```
fee1339d401f0b6de238dd09624621b5ceaa0076dc5bff96124051443f8e4b5c:     file format elf64-x86-64


Disassembly of section .init:

00000000004008d8 <_init>:
  4008d8:       48 83 ec 08             sub    $0x8,%rsp
  4008dc:       48 8b 05 3d 11 20 00    mov    0x20113d(%rip),%rax        # 601a20 <__gmon_start__>
  4008e3:       48 85 c0                test   %rax,%rax
  4008e6:       74 05                   je     4008ed <_init+0x15>
  4008e8:       e8 c3 00 00 00          call   4009b0 <__gmon_start__@plt>
  4008ed:       48 83 c4 08             add    $0x8,%rsp
  4008f1:       c3                      ret

Disassembly of section .plt:

0000000000400900 <.plt>:
  400900:       ff 35 2a 11 20 00       push   0x20112a(%rip)        # 601a30 <_GLOBAL_OFFSET_TABLE_+0x8>
  400906:       ff 25 2c 11 20 00       jmp    *0x20112c(%rip)        # 601a38 <_GLOBAL_OFFSET_TABLE_+0x10>
  40090c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400910 <free@plt>:
  400910:       ff 25 2a 11 20 00       jmp    *0x20112a(%rip)        # 601a40 <free@GLIBC_2.2.5>
  400916:       68 00 00 00 00          push   $0x0
  40091b:       e9 e0 ff ff ff          jmp    400900 <.plt>

0000000000400920 <pthread_create@plt>:
  400920:       ff 25 22 11 20 00       jmp    *0x201122(%rip)        # 601a48 <pthread_create@GLIBC_2.2.5>
```

Figure 3: Raw output for the objdump command.

After the command, the output text is processed, to isolate the opcodes. Nonetheless, I have to revise some operations as this padding nopw. Because the opcode name extracted is *data16* instead of nopw.

```
  400a9a:       48 85 c0                test   %rax,%rax
  400a9d:       74 11                   je     400ab0 <deregister_tm_clones+0x30>
  400a9f:       5d                      pop    %rbp
  400aa0:       bf 08 1b 60 00          mov    $0x601b08,%edi
  400aa5:       ff e0                   jmp    *%rax
  400aa7:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  400aae:       00 00
  400ab0:       5d                      pop    %rbp
  400ab1:       c3                      ret
  400ab2:       66 66 66 66 66 2e 0f    data16 data16 data16 data16 cs nopw 0x0(%rax,%rax,1)
  400ab9:       1f 84 00 00 00 00 00

0000000000400ac0 <register_tm_clones>:
  400ac0:       be 08 1b 60 00          mov    $0x601b08,%esi
  400ac5:       55                      push   %rbp
  400ac6:       48 81 ee 08 1b 60 00    sub    $0x601b08,%rsi
  400acd:       48 c1 fe 03             sar    $0x3,%rsi
  400ad1:       48 89 e5                mov    %rsp,%rbp
  400ad4:       48 89 f0                mov    %rsi,%rax
  400ad7:       48 c1 e8 3f             shr    $0x3f,%rax
  400adb:       48 01 c6                add    %rax,%rsi
  400ade:       48 d1 fe                sar    %rsi
  400ae1:       74 15                   je     400af8 <register_tm_clones+0x38>
  400ae3:       b8 00 00 00 00          mov    $0x0,%eax
  400ae8:       48 85 c0                test   %rax,%rax
  400aeb:       74 0b                   je     400af8 <register_tm_clones+0x38>
  400aed:       5d                      pop    %rbp
```

Figure 4: Padding nopw (400ab2) that breaks the text processing to get the opcodes.

Also, it is needed to revise other errors after the text processing. After processing all the files, 12605345 opcodes are collected. In the dataset appear 556 different opcodes.

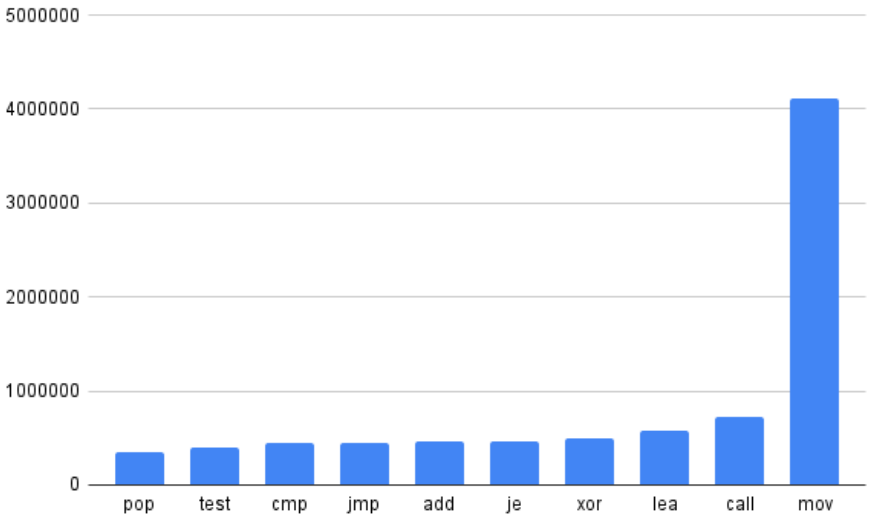

Figure 5: Top 10 opcodes that appear the most in the dataset.

Checking the distinct opcodes that appear in the dataset, I found that there is one in particular that suggests an error in the process, *(bad)*. It is produced by two files in the dataset. And it is due to *objdump* do not know how to disassemble an instruction.

```
610660:       61                      (bad)
610661:       03 00                   add    (%rax),%eax
610663:       00 00                   add    %al,(%rax)
610665:       00 00                   add    %al,(%rax)
610667:       00 61 03                add    %ah,0x3(%rcx)
61066a:       00 00                   add    %al,(%rax)
61066c:       00 00                   add    %al,(%rax)
61066e:       00 00                   add    %al,(%rax)
610670:       61                      (bad)
610671:       03 00                   add    (%rax),%eax
610673:       00 00                   add    %al,(%rax)
610675:       00 00                   add    %al,(%rax)
610677:       00 61 03                add    %ah,0x3(%rcx)
61067a:       00 00                   add    %al,(%rax)
61067c:       00 00                   add    %al,(%rax)
61067e:       00 00                   add    %al,(%rax)
610680:       61                      (bad)
```

Figure 6: *Objdump* (bad) opcode output.

As just two files contain the *(bad)* opcode, those would be discarded.

```
[andreugirones@Andreus-MacBook-Air opcodes % grep --exclude=\*.txt -rlw . -e "(bad)"
./67e38438759f34eaf50d8b38b6c8f18155bcc08a2e79066d9a367ea65e89aa3d
./76210f0a7710b40095d32f81bfb5d0576f81ac7cbdc63cf44ababb64cb8e65b7
```

Figure 7: Files containing the (bad) opcode.

To check if all the opcodes are opcodes existing in the x86 ISA, I compared the opcodes against the list that I found here: `https://www.felixcloutier.com/x86/`. As the list is not complete, I have found that 275 do not have a match in the list. I need to find a complete list to verify the process.

2 N-grams

As to extract the information gain of each of the n-grams, the n-grams from both classes (benign and malicious) are needed. I have downloaded from here: `https://archlinux.org/packages/core/x86_64/coreutils/`, the coreutils package with 103 x86-64 benign ELF files.

631235 opcodes are extracted from all the benign files. Where, there are 233 unique opcodes.

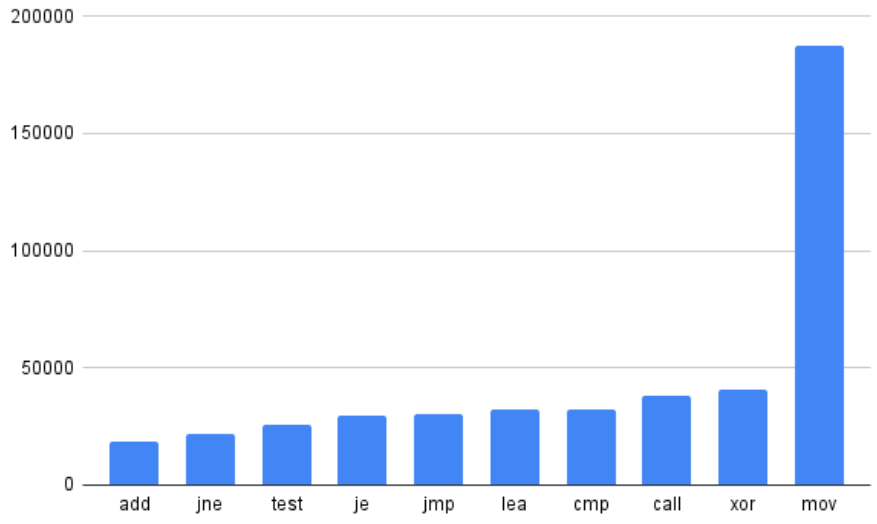

Figure 8: Top 10 opcodes that appear the most in the benign files.

On the other hand, as said above, I have obtained a total of 12605345 opcodes from the malicious files. Where, there are 556 unique opcodes.

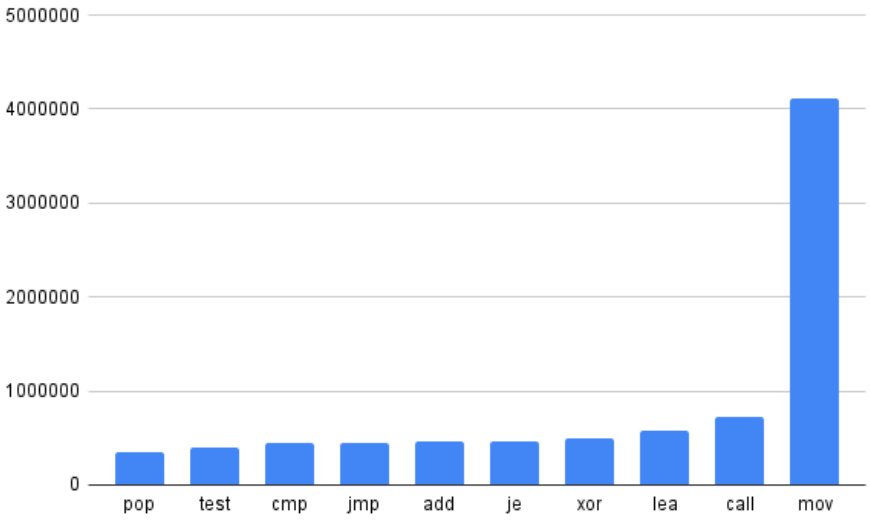

Figure 9: Top 10 opcodes that appear the most in the malicious files.

The *preprocessing.py* python script extracts the n-grams and the features from the dataset. It could be found on: `https://github.com/agirones/malware_detector`.
The script expects that files containing opcodes are stored in separated directories for malicious and benign files. In other words, there is no directory containing both, malicious and benign opcode files.
I have divided the files containing the opcodes in two directories i.e. *benign_opcodes/* and *malicious_opcodes/*

```
endbr64
sub
mov
test
je
call
add
ret
push
call
call
call
call
call
call
call
call
nopw
endbr64
push
push
push
push
```

Figure 10: Sample of a file containing opcodes.

All files are scanned, and the n-grams are extracted for each of the files. To do so, the *nltk* library is used.

```
('endbr64', 'sub', 'mov')
('sub', 'mov', 'test')
('mov', 'test', 'je')
('test', 'je', 'call')
('je', 'call', 'add')
('call', 'add', 'ret')
('add', 'ret', 'push')
('ret', 'push', 'call')
('push', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'call')
('call', 'call', 'nopw')
('call', 'nopw', 'endbr64')
('nopw', 'endbr64', 'push')
('endbr64', 'push', 'push')
('push', 'push', 'push')
('push', 'push', 'push')
('push', 'push', 'push')
('push', 'push', 'push')
```

Figure 11: Ngrams obtained after processing the opcodes shown above.

Once all the n-grams of a file are extracted, all the duplicated n-grams are filtered. For each one of them, an occurrence in a benign or malicious file is counted. Obtaining at the end, for each of the unique n-grams in the dataset, in how many benign and malicious files appear each one. Also, the number of benign and malicious analyzed files are recorded.
Finally, the document frequency is calculated for each of the unique n-grams, see Figure 12. The L n-grams with the highest document frequency are chosen as features i.e. the top $L/2$ from benign class and the top $L/2$ from malicious class.

$$DF(ng, S_C) = \frac{\lg|S_{Vng} = 1|}{\lg|S_C|}$$

Figure 12: Document frequency; where ng is a n-gram, S_C is the partition of the dataset that belongs to class C, S_{Vng} is the partition of S_C where ng appears.

Once the features are chosen, a feature vector could be extracted from an specific file.