An NCC Group Publication

Applying normalised compression distance for architecture classification

Prepared by:
Thomas Marcks von Würtemberg

Contents

1 Introduction

When working with malware research and black box penetration testing, it is not always clear what data you are working on. It may be that code has been obfuscated through a variety of techniques or perhaps the platform architecture is unknown to the analyst.

In order to disassemble binaries properly, one needs to know the architecture that the binary has been compiled for. This can be hard to know if there are no headers or other identifying strings to go on.

In this whitepaper, we present a technique to classify binaries and shellcode with statistical analysis using normalised compression distance.

The concept of normalised compression distance[1] was introduced in 2005 by R. Cilibrasi and P.M.B. Vitanyi. It is based on the concept of written text comparison. The author showed that it was possible to compare information distance via compression and compare computer objects to each other, for instance two computer-generated images.

Normalised compression distance has been shown[2] to be useful for classifying unknown data in the field of forensics and malware. This paper will show that it is also possible to use normalised compression distance to discern architecture classification of computer binaries.

2 Normalised compression distance

The basis of normalised compression distance (NCD) is to compare the length of two compressed objects to give an indication of how similar the objects are. The formula for this is:

$$NCDz = \frac{Z(xy) - min\{Z(x), Z(y)\}}{max\{Z(x), Z(y)\}}$$

The formula shows that if we take the length of the compressed concatenation of the string X and the string Y and subtract the length of the shortest compressed strings then divide that by the longest compressed length, it will give us the NCD value. The smaller the value, the more alike the two objects are.

The theory works on the principle that similar information objects compress similarly. This is also true regardless of their visual representation - the underlying information is still the same. Take the following example of the text *lorem ipsum* with the spaces removed:

[1] http://ieeexplore.ieee.org/document/1412045/?tp=&arnumber=1412045
[2] https://www.dfrws.org/sites/default/files/session-files/paper-the_normalised_compression_distance_as_a_file_fragment_classifier.pdf

Loremipsumdolorsitamet,consecteturadipiscingelit.Maurisvenenatistellusutanteultricesvolutpat.Pellentesquevelaliqueturna.Inrisusquam,finibusquiseuismodet,suscipitutneque.Fusceeteuismodurna.Vivamusinpuruseusembibendumblanditsitametacerat.Utornareegetenimarhoncus.Maecenasauctoraliquetleo,utmalesuadaurnaauctornon.Utrutrumpulvinarmagna,vitaedapibusvelitbibendumnec.Praesentvelfelisfinibusantebibendumsagittis.Ininterdumtortoratdignissimvestibulum.Nunctortorenim,placerateuvolutpateu,egestasnontortor.Aeneanacodioegetdolorvehiculavehiculasedvelneque.Indictumturpisidloremvenenatis,sedtinciduntleoegestas.Maurisidsemhendrerit,laoreetfelisfringilla,rhoncusest.Maecenaslaoreetportarisusneceleifend.Nullamacposuerediam.Pellentesqueullamcorperullamcorpertellus,eteuismodnisldignissimin.Quisquecursusinterdumjustoinimperdiet.Fuscemattisfermentumfelissitametsed.

Here is the same text but obfuscated in ROT13[3]

Yberzvcfhzqbybefvgnzrg,pbafrpgrghenqvcvfpvatryvg.Znhevfirarangvfgryyhfhgnagrhygevprfibyhgcng.Cryyragrfdhrirynyvdhrghean.Vaevfhfdhnz,svavohfdhvfrhvfzbqrg,fhfpvcvghgardhr.Shfprrgrhvfzbqhean.Ivinzhfvachehfrhfrzovoraqhzoynaqvgfvgnzrgnpreng.Hgbeanerrtrgravzneubaphf.Znrpranfnhpgbenyvdhrgyrb,hgznyrfhnqnheannhpgbeaba.Hgehgehzchyivanezntan,ivgnrqncvohfiryvgovoraqhzarp.Cenrfragirysryvfsvavohfnagrovoraqhzfntvggvf.Vavagreqhzgbegbengqvtavffvzirfgvohyhz.Ahapgbegberavz,cynprengrhibyhgcngrh,rtrfgnfabagbegbe.Nrarnanpbqvbrtrgqbybeiruvphyniruvphynfrqiryardhr.Vaqvpghzghecvfvqyberzirarangvf,frqgvapvqhagyrbrtrfgnf.Znhevfvqfrzuraqerevg,ynberrgsryvfsevatvyyn,eubaphfrfg.Znrpranfynberrgcbegnevfhfarpryrvsraq.Ahyynznpcbfhrerqvnz.Cryyragrfdhrhyynzpbecrehyynzpbecregryyhf,rgrhvfzbqavfyqvtavffvzva.Dhvfdhrphefhfvagreqhzwhfgbvavzcreqvrg.Shfprznggvfsrezraghzsryvffvgnzrgfrq.

The information contained in these two examples is the same. If we calculate the normalised compression distance on them, we get the following output with a normalised compression distance (NCD) of 0.8028. In this example, lzma compression implemented in python 2.7 (see appendix 8.2) was used.

Lorem Ipsum vs Lorem Ipsum ROT13
Length of compressed concatenation: 1020
Length of compressed x: 564
Length of compressed y: 568
NCD = 0.802816901408

As a comparison, consider the following random data generated via /dev/urandom:

[3] https://en.wikipedia.org/wiki/ROT13

lizsDFCXSSmuRTjzNDBnciMwwkYEhupFBAjTImpgrKMhZuHwdhZbVZYHARDK
jJMMDDOvBYMOaFVXSXOHCrnDMsZbATBJruFeREYTdDnVdTYgQSzqVpnsR
qUsfDuArcQLYmRNBkGonPqfOsRiiPRyHqmNGIIeAZJyVpomsTlFecnZEzTDmf
VVrOPnquxgBNZJvlmavReOpLFMPYYcGRbOsqWmFJejRCUbgHsNntxHqoplG
FPhJwsGlSnIDxlNpbZjSDjCkqESgnDCaCTvaNNkQbvYkOvFxlyCIXZNYFaiEtTz
PMQXGRQUGLMtFlKxLWEdEijFKWNwfRIvgrNhwagjVoTylTmTYNsqqGZRdptH
JvgwHmeAkQxaeMeHrermQGcHOfbzqzNXmNwXeyysCHljLQuhslURYHgEyKkf
StdsBIAVwabwTjjLHPZVzrKHDrQYhraAfciZgdMoJOgqMiWPKqYPNGvjpqNmkrr
uEFBxIgtbjjfVGmjUVmmfrriUIVyxuNqvrBCwlKsvMVMIUOAFJNlagEdBaVKaFWi
DeyyOrLaISMAgHetcOjfpOgqHuGpLfLtpetFYRvNaWjzBvPFuhWwBNUYwdHQP
kymzMtsxOkgeHWeGEZaDvmOoozkXyQGPZmfqIgFYpMpyjlvcElDvPIQSWNpR
xwwxCwBgrqZKDExsymXXoETwChKZVeHzyYgECUIhfWdDFHJgCvAbOQzkzX
silXbJmPgPQpNnEBRoGZQoVEeERmQnODThEwzSqESKYcluAZkQRXOLvJg
QoKqiwMlFWKnylQdGfzdWSAYZbvLEMVYXDBxFUSafKjpkNWobTtaTkRKnOS
vcvBJNbyXsQlZwMlifhTpJxxyeTKJHyhExUeyudEdpKxrBaIcqLUtigvPtbEshRTB
WBopNipxIjqZdECyEbagFJHDsiRYjafmjTgNxpVCQVWVBkQMQPjOaNAHEoGz
agEgDycBeeHAiNUVjxaOZEVJZUwCGJZKwCgqV

It is similar in length to the previous example but the information is very different, whereas, the information in the *lorem ipsum* example is well structured pseudo Latin. This example contains random letters with no structure. If we calculate the NCD of the *lorem ipsum* texts VS our random data, we will be able to see that the NCD value is greater.

Lorem Ipsum vs Random data
Length of compressed concatenation: 1332
Length of compressed x: 564
Length of compressed y: 848
NCD = 0.905660377358

In the example above, we used normal readable text but the same technique can also be applied to binaries. In the same way that our *lorem ipsum* ROT13 text was just another representation of the *lorem ipsum* text, two binaries of the same source code compiled for different architectures are also just different representations of the same underlying information.

3 Executables & shellcode

Both executable files and shellcode are a representation of machine code instructions, called “opcodes”, which are used by the CPU to decode what instructions to run. All executable files, regardless of format (such as ELF in Linux or PE in Windows), contain these instructions. These opcodes are architecture-specific and will be different depending on which architecture the code has been compiled for. Binaries may contain other information but for the purpose of classification, the opcodes are the only information we need to extract. There are multiple ways of retrieving the opcodes from a binary but in Linux the easiest way to do this is to use the objdump tool. The objdump tool can be cross-compiled to get known data samples from other architectures which might otherwise need specialised hardware to run. For an example of a script used for dumping binaries for use as known data, see the appendix (8.1).

4 Architecture classification

Let's look at the program putty and its portable 32 bit[4] binary vs putty 64 bit[5] portable binary example. They are the same program, packed in the same format (PE executable) but compiled with different opcodes for two different architectures. To see if we can tell the binaries apart, we can compare them to the known data that was previously retrieved using objdump.

In the *lorem ipsum* examples above, python was used but as the samples got larger, multiple runs were required (as will be shown later in this paper), therefore, a better solution was written in C. The amount of computation power required when running on multiple chunks is large and the zlib library in C is twice as fast as the zlib library in python.

This experiment generated the following results where we can see the putty32 bit binary (called Putty32.exe) and its corresponding NCD value with known data from the architectures listed:

	I386	86x64	Amiga
Putty32.exe	1.002155	1.000481	1.001260
Putty64.exe	0.999094	1.000949	1.001356

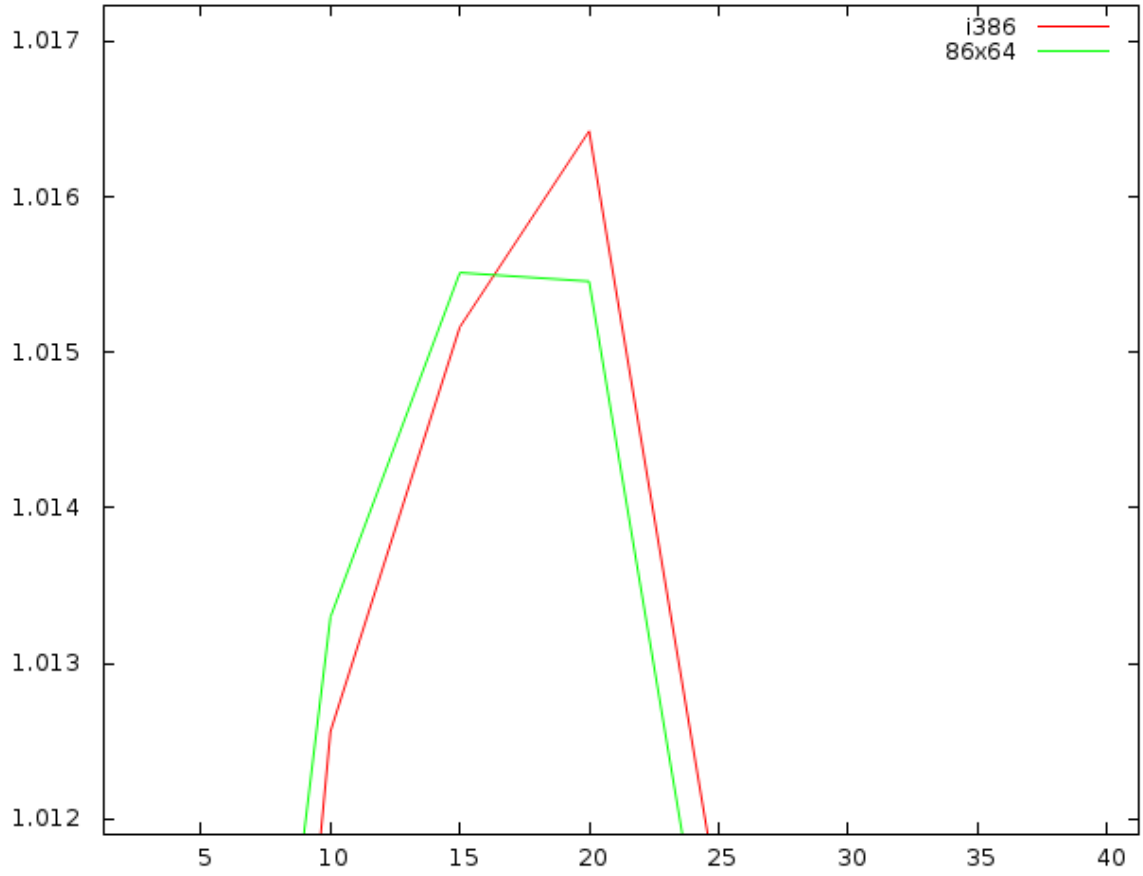
We can see that the detection is not working and in most cases the classification will be wrong. This is due to the fact that the part of the executable file that is composed of opcodes is small compared to the other supporting sections that contain information on libraries, linking tables, strings and other data required by the code. Therefore, when using known data as in the example above, only small differences will show and the risk of false positive classification is too big for reliable detection of which architecture the code was compiled for.

In order to get around this problem, we can use the law of averages and arithmetic mean to take away the large spikes that are created from parts which are not opcodes. We can split the data into small chunks and then compare the average of the NCD values of all chunks within the sample under analysis against the NCD of known files for every part of the sample file. Using this technique of taking the average of the NCD, architecture specifics become much more apparent and we reduced the risk of false positives significantly as shown in the table below (rounded to 6 decimals):

	I386	86x64	Amiga
Putty32.exe	1.012562	1.013297	1.044359
Putty64.exe	1.012363	1.009840	1.044535

[4] https://the.earth.li/~sgtatham/putty/0.68/w32/putty.exe
[5] https://the.earth.li/~sgtatham/putty/0.68/w64/putty.exe

The chunk size matters as once small enough chunks are used, we see the false classification disappear. The graphs below are all made by comparing the putty32.exe binary against known data from 32bit and 64bit Intel architectures:

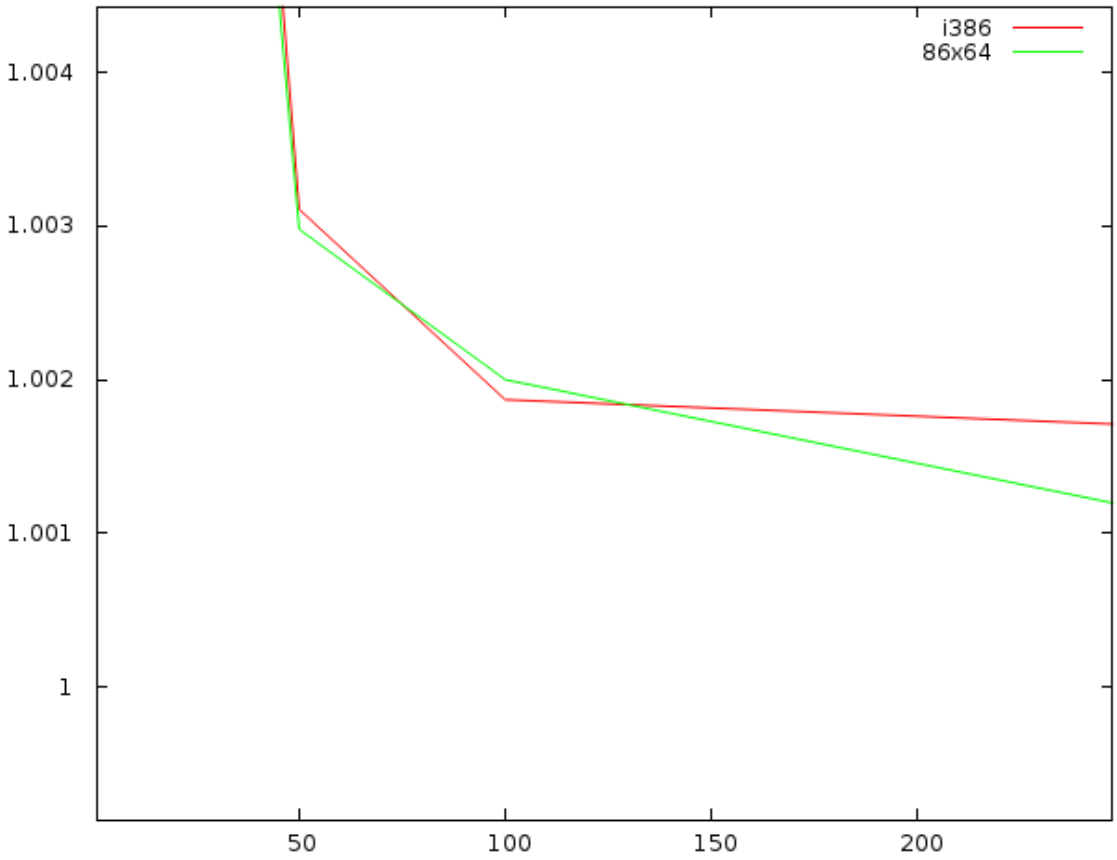


X-axis: Chunk size Y-axis: NCD value

Figure 1 - 32bit Intel exe compared to i386 and 86x64 (small chunks)

The graph in Figure 1 clearly illustrates that using a chunk size of 10 gives an optimal result.

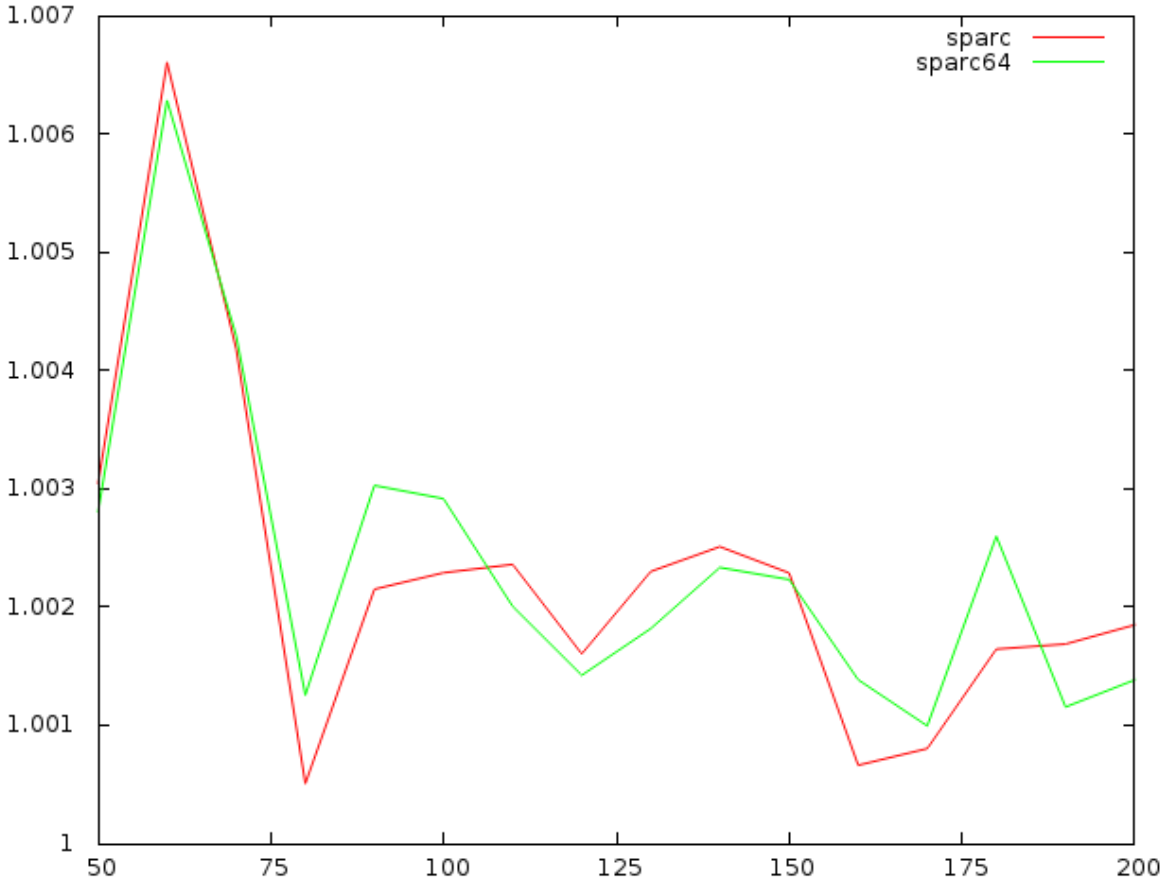
If we look at larger chunk sizes, we see that there are certain spots when it is possible that the detection is correct but it is much more unreliable.



X-axis: Chunk size Y-axis: NCD value

Figure 2 - 32bit Intel exe compared to i386 and 86x64 (large chunks)

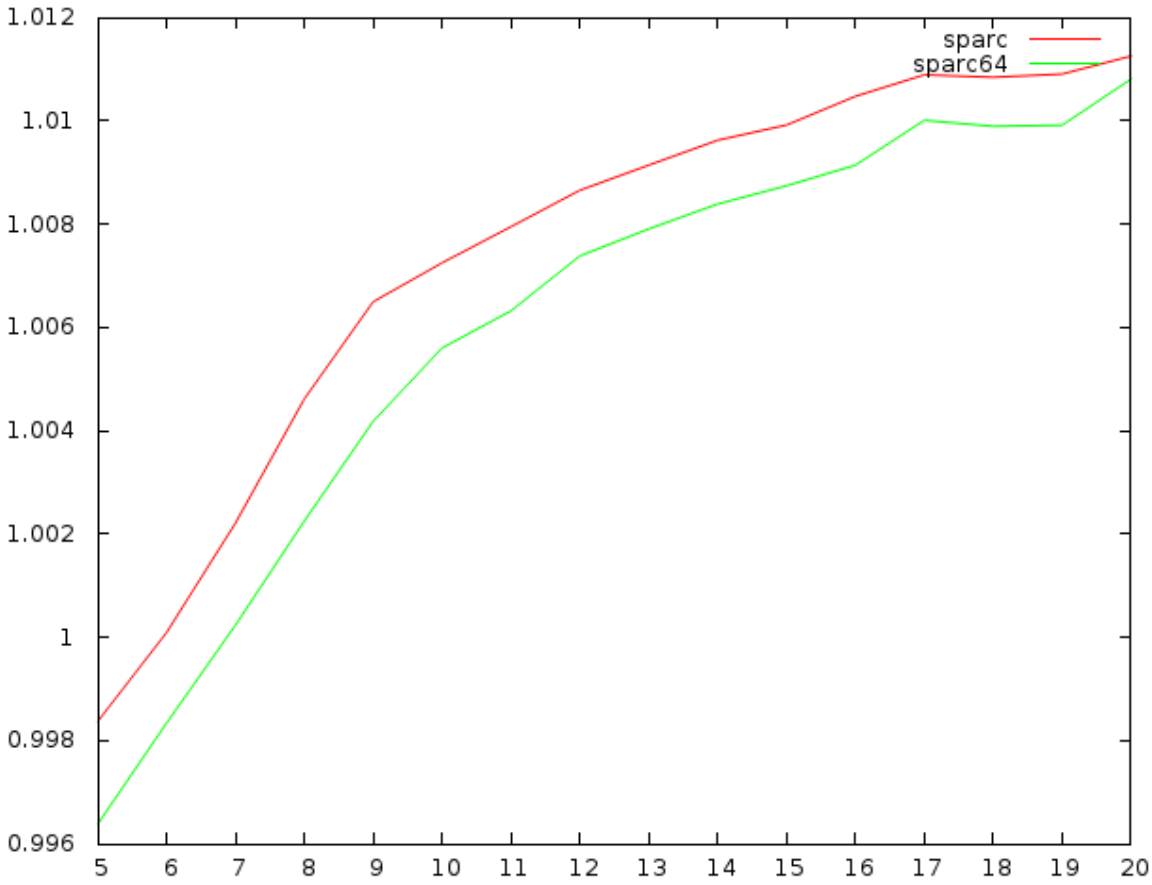
If we look at a 64bit SPARC version of the bash binary, we see that larger chunk sizes appear to be very sporadic and do not share the same patterns as the Intel binary which makes it impossible to pick a larger size for use across multiple architectures.



X-axis: Chunk size Y-axis: NCD value

Figure 3 - 32bit SPARC binary compared to 64bit SPARC binary (large chunks)

When plotting the smaller chunk size for the same 64bit bash binary, we see that the values stabilise and that 10 is a good chunk size to use.



X-axis: Chunk size Y-axis: NCD value

Figure 4 - 32bit SPARC binary compared to 64bit SPARC binary (small chunks)

Both the files in the above example are executable files and will, if tested by the "file" command in Linux show "putty32.exe: PE32 executable (GUI) Intel 80386, for MS Windows" and "putty64.exe: PE32+ executable (GUI) x86-64, for MS Windows" respectively. The detection using the "file" program is carried out on the file header. The file header contains information on how to interpret the file. However, if we remove the header by cutting away the first 100 bytes of the file, the output will simply be "data" and we have no chance of knowing what the file is by using programs like "file".

If we run our NCD analysis on these cut binaries then we can still accurately discern which architecture the binaries were compiled for, as shown in the table below (rounded to 6 decimals):

	I386	86x64	Amiga
Putty32-nohead.bin	1.012514	1.013262	1.044315
Putty64-nohead.bin	1.013123	1.010526	1.044808

We have seen that we can identify the target architectures of Intel binaries, so let's expand the scope and look at a 32bit SPARC binary[6] and other architectures. The result of which is the following table:

	Bash-static-sparc32
I386	1.01748987854251012145
x86x64	1.02056410256410256410
Amiga	1.04659919028340080971
SPARC	1.00540512820512820512
SPARC64	1.00590317642556448526
ARC	1.04603379541542085711
VAX	1.02682186234817813765

As we can see the smallest average NCD value was given when using the known data from the sparc32 bit platform. To show that this is also working for 64bit SPARC, we have another version of bash compiled for sparc64[7], which gives the following output:

	Bash-static-sparc64
I386	1.01707038681039949270
x86x64	1.01986144578313253012
Amiga	1.04466708941027266962
SPARC	1.00724819277108433734
SPARC64	1.00559701492537313432
ARC	1.04491676955170505342
VAX	1.02575459733671528218

5 Conclusion

As shown in this paper, normalised compression distance is a viable method for classifying which architecture certain code was compiled for and can even be used to detect the presence of opcodes.

Refined application of the techniques presented in this paper can be used to aide in malware detection and classification. More applications are possible even if they are not explored fully in this paper, for instance, applying these techniques to cryptography for detection of known plaintext in weak encryption ciphers.

An alternative would be to look at frequency distributions of instructions but that method has issues with some frequent opcodes having different meaning in different architectures. The method of using normalised compression distance is therefore better to use.

To be able to fully take advantage of this method the known data must be of certain quality. For example, too small a data sample and the detection will give false positives. Tests also indicated that if the known data is in hexadecimal format (\x41\x41\x41\x41) the unknown data needs to be normalised to the same format for effective comparison. The techniques shown in this paper for dumping opcodes from binaries are not effective at discerning the architecture of the compactly written

[6] http://ftp.debian.org/debian/pool/main/b/bash/bash-static_4.2+dfsg-0.1+deb7u3_sparc.deb
[7] http://ftp.ports.debian.org/debian-ports/pool-sparc64/main/b/bash/bash-static_4.4-4_sparc64.deb

shellcode that is found in some viruses and exploits. Given the right input data such as a large enough sample or samples of shellcode, the same normalised compression distance technique should apply for the detection and classification of shellcode. This is part of our future work and is beyond the scope of this initial paper.

6 References & further reading

- The similarity metric - Ming Li et al
  http://ieeexplore.ieee.org/document/1362909/?tp=&arnumber=1362909
- The Normalised Compression Distance as a file fragment classifier - Stefan Axelsson
  https://www.dfrws.org/sites/default/files/session-files/paper-the_normalized_compression_distance_as_a_file_fragment_classifier.pdf
- The Normalised Compression Distance as a Distance Measure in Entity Identification - Sebastian Klenk, Dennis Thom, Gunther Heidemann
  http://www.vis.uni-stuttgart.de/~klenksn/paper/ncd.pdf
- Clustering by compression - R. Cilibrasi, P.M.B. Vitanyi
  http://ieeexplore.ieee.org/document/1412045/?tp=&arnumber=1412045&url=http:%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1412045

7 Acknowledgements

A big thanks to my colleague Thomas Atkinson for proofreading, good comments and fantastic support as well as Chris Anley for good scientific improvement suggestions.

8 Appendix

8.1 Bash script for dumping binaries for use as known data

```
#!/bin/bash
while read file
do
     opcode=$(objdump -d $file | grep '[0-9a-f]:'| grep -v 'file'| cut -f2 -d:|
cut -f1-6 -d ' '| tr -s ' '| tr '\t' ' ' | sed 's/ $//g'|sed 's/ /\\x/g' | paste -d
'' -s| sed 's/^/"/'|sed 's/$/"/g')
     perl -e "print($opcode);" >> opcodes.bin
done < <(find ./ -executable)
```

8.2 Python script for NCD calculation

```
#!/usr/bin/python
from __future__ import division
import sys
import os
import lzma
f1 = sys.argv[1]
f2 = sys.argv[2]
fd1=open(f1,"rb")
x=fd1.read()
fd1.close()
fd2=open(f2,"rb")
y=fd2.read()
fd2.close()
xy=x+y
zxy = lzma.compress(xy)
zx = lzma.compress(x)
zy = lzma.compress(y)
print "Length of compressed concatination: %d"%len(zxy)
print "Length of compressed x: %d"%len(zx)
print "Length of compressed y: %d"%len(zy)
ncd = ((len(zxy)-min(len(zx), len(zy)))/(max(len(zx), len(zy))))
print "{} {}".format(sys.argv[2],ncd)
```