Mutational Robustness in x86 systems

Sperl Thomas
sperl.thomas@gmail.com

October 26, 2010

Abstract

The robustness of different coding styles are compared at the occurrence of bit-flip mutations. A meta-language concept with a redundant alphabeth is found to be the most robust approach.

1 Introduction

In a recent text an approach for artificial evolution in x86 systems has been presented.[1] One important property of an evolutionary system is its robustness at the occurrence of mutations. If the system is too brittle, too few neutral mutations will occure, hence there is no evolution.

An example of such a brittle environment is *CoreWorld* by Steen Rasmussen. Furthermore it has been discovered that the x86 instruction set is also too brittle to allow evolution.[2]

2 Robustness of different x86 approaches

The most direct way to define `Robustness` is the following one:

$$\text{Robustness} := \frac{\text{number of non-lethal mutations}}{\text{number of mutations}} \tag{1}$$

For example, if `Robustness`$= 1$, all mutations will be non-lethal, the opposite is `Robustness`$= 0$, where all mutations are lethal.

To compare the robustness, a simple program that copies itself in the current directory (with a specific probability of mutations) are written in four different ways:

1. Usual x86 code: The program has been written in usual assembler, without any special features

2. x86 code with metalanguage-like structure: The program is still direct x86 code (no translations), but consists of a structure that is used by the metalanguage.

```
.code
     ...
                                ; edx contains new random number
     mov  ebx, DataOffset
     add  ebx, (RandomNumber-DataOffset)
     mov  edi, ebx              ;edi=RandomNumber

     mov  ebx, edx
     mov  dword[edi], ebx       ; mov dword[RandomNumber], edx
     ...
.end code
```

This has the same structure as the translated code of the meta-language concept.

3. Meta-language without redundant alphabeth: This program uses a meta-language, the alphabeth is *not* redundant.

4. Meta-language with redundant alphabeth: This program uses a meta-language, the alphabeth is redundant. The redundancy of the alphabeth can be seen in the Appendix.

3 Results

In the experiment, one bit after another has been changed. After such a bit-flip the program has been run. If it was able to reproduce itself and execute the offspring, the mutations has been considered as non-lethal, otherwise the bit-flip was lethal.

No.	**#(bits)**	**lethal**	**non-lethal**	**Robustness**
1	2864	1916	948	33.1%
2	5400	3393	2007	37.2%
3	3472	2104	1368	39.2%
4	3472	1333	2139	61.6%

The meta-language with a redundant alphabeth is much more robust than all other approaches, thus gives a good basis for further evolutional experiments.

References

[1] Sperl Thomas, *Taking the redpill: Artificial Evolution in native x86 systems*, 2010.

[2] Dimitris Iliopoulos, Christoph Adami and Peter Ször, *Darwin inside the machines: malware evolution and the consequences for computer security*, Virus Bulletin Conference, 2008.

A Redundancy of Alphabeth

nopsA:

```
0100 0110    0100 0111
0100 1110    0100 1111
```

nopsB:

```
0100 0101    0100 1100
0100 1101
```

nopsD:

```
0000 1010    0100 1010
0100 1011
```

nopdA:

```
0000 0110    0101 0110
```

nopdB:

```
0001 1111    0010 1111
0011 1110    0011 1111
```

nopdD:

```
0001 1101    0010 1101
0011 1100    0011 1101
```

saveWrtOff:

```
0010 1100    0110 1010
0110 1100    0110 1101
0110 1110
```

saveJmpOff:

```
1000 1011    1010 1000
1010 1001    1010 1010
1010 1011
```

writeByte:

```
0011 0111    0101 0111
0111 0110    0111 0111
```

writeDWord:

```
0110 1000    0110 1010
```

save:

```
1000 1010    1000 1100
1000 1101    1000 1110
1000 1111    1001 1101
1001 1110
```

addsaved:

```
          1001 0010     1001 0011

subsaved:
          1001 0000     1001 0100

getDO:
          1000 0010     1100 0010
          1100 0011     1100 0100
          1100 0101     1100 0110
          1100 0111     1100 1000
          1101 0000     1110 0000
          1110 0010     1110 0011

getdata:
          0010 1110     0110 0111
          0110 1011     0110 1111

getEIP:
          0110 0100     0110 0110
          1010 0100     1010 0101
          1110 0100     1110 0101
          1110 0110     1110 0111
          1110 1100

zer0:
          0101 1000     0111 0000
          0111 1000     0111 1001
          0111 1010     0111 1100
          1111 1000     1111 1001
          1111 1010

push:
          0010 0001     0110 0001
          0110 0010     0110 0011
          0110 0101     0110 1001
          0111 0001     1110 0001

pop:
          0101 0010     0101 0011
          0101 0100     0101 0101
          0101 1100     0101 1101
          0101 1110

mul:
          1000 0100     1000 0110
          1010 0110     1010 0111

div:
          1000 0111     1001 0101
          1001 0110     1001 0111
```

shl:

```
        1010 1100     1010 1101
        1010 1110     1010 1111
```

shr:

```
        1001 1100     1011 1000
        1011 1010     1011 1011
        1011 1100     1011 1101
        1011 1110
```

and:

```
        1100 1001     1100 1010
        1100 1011     1100 1101
        1100 1110     1100 1111
```

xor:

```
        0101 1010     1001 1010
        1101 1000     1101 1001
        1101 1010     1101 1011
```

add0001:

```
        1111 1111
```

add0004:

```
        0111 1110     1111 1110
```

add0010:

```
        0111 1101     1111 1100
        1111 1101
```

add0040:

```
        0111 1011     1110 1101
        1111 1011
```

add0100:

```
        1011 0110     1011 0111
        1111 0110     1111 0111
```

add0400:

```
        1110 1011     1110 1101
        1110 1111
```

add1000:

```
        1001 1011     1001 1111
        1101 1100     1101 1101
        1101 1110     1101 1111
```

add4000:

```
        1011 1111
```

```
sub0001:
          0101 1111     0111 1111

nopREAL:
          0000 1001     0100 1001
          0101 1001     0101 1011
          1100 1100     1110 1110

JnzUp:
          0101 0001     1101 0001
          1101 0010     1101 0011
          1101 0111     1110 1001
          1110 1010

JnzDown:
          1011 0100     1011 0101
          1110 1000

JzDown:
          0111 0100     0111 0101
          1101 0100     1101 0101
          1101 0110     1111 0100
          1111 0101

ret:
          0000 1011     0000 1111

CallAPIGetTickCounter:
          0111 0010     0111 0011
          1011 0000     1011 0001
          1011 0010     1011 0011
          1111 0000     1111 0001
          1111 0010     1111 0011

CallAPIGetCommandLine:
          0000 0000     0000 0001
          0000 0010     0000 0100
          0000 0101     0000 1000
          0001 0000     0010 0000
          0010 0010     0010 0100
          0100 0000     1000 0000

CallAPICopyFile:
          1000 0001     1000 0011
          1000 0101     1000 1000
          1000 1001     1001 0001
          1010 0000     1010 0001
          1010 0010     1010 0011
          1100 0001
```

CallAPICreateFile:

```
0001 0100    0001 0101
0001 0110    0011 0100
0011 0101    0011 0110
```

CallAPIGetFileSize:

```
0000 1100    0000 1101
0000 1110
```

CallAPICreateFileMapping:

```
0001 1000    0001 1001
0001 1010    0001 1100
0001 1110    1001 1000
1001 1001
```

CallAPIMapViewOfFile:

```
0000 0011    0001 0001
0001 0010    0001 0011
0001 0111
```

CallAPICreateProcess:

```
0011 0001    0011 0010
0011 0011    0100 0001
0100 0010    0100 0011
0100 0100    0100 1000
0101 0000    0110 0000
1100 0000
```

CallAPIUnMapViewOfFile:

```
0000 0111    0010 0011
0010 0101    0010 0110
0010 0111
```

CallAPICloseHandle:

```
0001 1011    0010 1010
0010 1011    0011 1001
0011 1010    0011 1011
```

CallAPISleep:

```
0010 1000    0010 1001
0011 0000    0011 1000
```