POLLACK PERIODICA
An International Journal for Engineering and Information Sciences
DOI: 10.1556/606.2018.13.1.1
Vol. 13, No. 1, pp. 3–20 (2018)
www.akademiai.com

AUTOMATIC TRANSLATION OF ASSEMBLY SHELLCODES TO PRINTABLE BYTE CODES

[1] Zsolt GÉCZI, [2] Péter IVÁNYI

Faculty of Engineering and Information Technology, University of Pécs
Boszorkány u 2, 7624, Pécs, Hungary
e-mail: [1]geztaap@mik.pte.hu, [2]ivanyi.peter@mik.pte.hu

Received 11 March 2017; accepted 3 January 2018

Abstract: The generation of printable shellcode is an important computer security research area. The original idea of the printable shellcode generation was to write a binary, executable code in a way that the generated byte code contains only bytes that are represented by the English letters, numbers and punctuation characters. In this way unfortunately only a limited number of CPU instructions can be used. In the originally published paper a small decoder is written with instructions represented by printable characters and the shellcode is decoded on the stack to be executed later. This paper, however describes a proof of concept project, which converts the source code of a full assembly program or shellcode to a new source code, whose compiled binary code contains only printable characters. The paper also presents new, printable character implementation of some CPU instructions.

Keywords: Shellcode, Source to source conversion, Printable ASCII characters

1. Introduction

The process of executable code generation contains a step where the source code is translated into a binary code. This binary code is a sequence of CPU instructions and they can be directly executed by a CPU. Every CPU instruction can be represented by one or more bytes and they form the machine language. However some of these bytes represent the English letters (41h-5Ah (A-Z) and 61h-7Ah (a-z)), numbers (30h-39h), punctuation characters, brackets, etc. (20h-2Fh, 3Ah-40h, 5Bh-60h, 7Bh-7Eh), according to the ASCII table. The small 'h' letter after the numbers denotes that the

number is a hexadecimal number. These characters between 20h-7Eh are the printable bytes (sometimes called alphanumeric characters) and they are the basis of normal texts.

Naturally not only the printable bytes, but the full range of bytes between 00h-FFh are compiled into a normal program. This observation is usually a very simple way to recognize an executable program or code, since a program may contain any of the 256 characters, while a normal text may contain only the above listed printable characters.

All this discussion is important, since a special technique has been proposed in the Phrack magazine [1] in 2001. The name of the proposed technique was 'Writing ia32 alphanumeric shellcodes'. Shellcodes are a form of attack against computer systems, where a code is injected and later executed. Usually the injection occurs when some data is received by a program. The program stores the received data, that is in reality an executable code and later, by exploiting some vulnerability of the system, the execution is redirected to the injected code. As described above the recognition of a code injection can be very simple by the non-printable characters, however if the injected code contains only printable characters then the injection cannot be detected in this way [2]-[4].

The original technique [1] proposed that only a small decoder is written in the special way using only bytes representing letters and numbers - hence the name of alphanumeric shellcode - and the rest of the actual executable code is decoded on the stack by the decoder. At the end of decoding the execution is passed to the decoded exploit. This method has been improved and a 'looped decoder' [5] has been proposed. This technique is built into a fully automatic alphanumeric shellcode generator [6], however this decoder has been further improved recently [7].

A very interesting variation of the alphanumeric shellcode has been described by Mason et al. [8]. They have created an alphanumeric binary code in such a way, that the resulting byte series is very similar to an English text. This technique makes the recognition of an executable code even more difficult. Another variation is where the shellcode is 'Unicode-proof' [9] or where the shellcode is UTF-8 compatible [10]. The technique has also been extended to the ARM computer architecture [11].

It is important to mention here that the use of shellcode as an exploit is not possible in itself, since currently all modern operation systems are using Data Execution Prevention (DEP), which makes it impossible to rewrite and execute data in runtime. A working exploit with shellcode has to switch off the DEP runtime.

However this paper only concentrates on describing a conversion technique for shellcodes. The technique is based on the original idea [1], but uses a different approach, where the source code of the full program is converted to another source code with instructions, whose generated byte code falls in the range of printable characters.

2. Basic assumptions

The main purpose of this paper is to describe a technique, which can convert a 16 bit x86 assembly source code to another source code. When the converted source code is compiled by an assembler the resulting executable code must contain only printable characters. The resulting executable code is a 16 bit COM program that can run only under Windows XP or earlier operating systems. The choice of a target system seems to

be limited and outdated, but this was an intentional decision, since the paper describes a proof of concept project, where the feasibility of the conversion of full programs is investigated. Furthermore generating only COM programs simplifies the conversion process, since the program always starts at the first instruction in the source code and once the program is loaded into memory the entry point of the execution is always at 100h.

As described above, only a limited number of the x86 instructions can be used in the resulting program. However, the limitation also applies to the numbers that are directly stored in the program. This means that a 16 bit number must fall in the range of 2020h-7E7Eh and an 8 bit number must fall in the range of 20h-7Eh.

It should also be mentioned, that the paper presents one approach, however other conversion methods are possible. During the development several conversion techniques have been found for several instructions, but only one of the techniques will be described in this paper.

As in the original technique the stack is going to be heavily utilized, but no executable code will be created on the stack. The main benefit of using the stack is that during stack operations the status register is not modified.

In this paper the Intel syntax and the syntax of the NASM [12] assembly compiler will be used.

2.1. Self-modifying code

It can be stated in advance, that at the moment there are instructions for which no equivalent alternative has been found. In this paper the approach to solve this problem is to use a self-modifying code. In higher level programming languages it is not very common to use this kind of technique, but in assembly it is possible to intermix data and code, since they are just a series of bytes and the only difference is whether it is get loaded and executed by the CPU or not. An example for the self-modifying code is:

```
       ADD [addr], byte 07h
addr: db 00h
```

where the ADD instruction adds the number 07 to the byte found at the address denoted as ‘addr’. The result is that after the execution of the ADD instruction the next instruction will be executed, which will have the bytecode of 07 and this is equivalent to the POP ES instruction.

Although this seem straightforward, but attention has to be paid to the address of the modified byte and to the instructions as well that are performing the modification, since all of them should be printable. First let’s consider the following code fragment:

```
       PUSH SI
       PUSH byte 033h
       POP  SI
       SUB  [addr], byte 07h
       ...
addr:  db   040h
       POP  SI
```

The bytecode of all instructions of this fragment is in the range of printable ASCII characters and these instructions are also equivalent to the POP ES instruction. Theoretically it is a solution; however there are two problems with this code:

- First the 'addr' address must be positioned in a way that the number of the address will also be in the range of printable ASCII characters.
- Second this set of instructions can be executed only once and this can cause a serious problem if it should be executed in a loop. The problem is that this set of instructions has a side effect. They do more than perform the POP ES instruction, since the byte at the 'addr' address will be modified.

To solve the second problem the following general code fragment can be considered:

```
        PUSH SI
        PUSH AX
        MOV AX, 0047h
        PUSH AX
        POP  SI
        POP  AX
        XOR  [a1], SI
        POP  SI
a1:     db   040h
        PUSH SI
        PUSH AX
        MOV AX, 0047h
        PUSH AX
        POP  SI
        POP  AX
        XOR  [a1], SI
        POP  SI
```

In this code fragment there are two instructions that are marked by bold letters. The reason for this that the bytecode of these instructions does not necessarily generate printable bytecodes, therefore another technique will be described later, which generates these instructions with the appropriate effects and printable bytecodes.

Another point to consider in the code fragment is that when the XOR instruction is applied first then it modifies the byte at address 'a1' and the byte afterwards since the SI register is 16 bit wide. However, the content of SI is 0047h and using the XOR operator between a value and 00h does not change the value, therefore the second byte will be unchanged. Furthermore when the XOR instruction is applied the second time to the byte at address 'a1' then the original value (0040h) will be restored.

This solution can be used for instructions with a single bytecode, however the same technique can be used for instructions with multiple bytecodes. In those cases the XOR instructions have to be applied to address 'a1', 'a1+1', …, 'a1+n'.

3. Implementation of the instructions

3.1. Frequently used code fragments

There are several code fragments that are used during the implementation of other assembly instructions.

Create zero on the top of the stack

In several cases it will be necessary to generate the zero value on the top of the stack. The following code fragment

```
PUSH 2020h
POP  AX
XOR  AX, 2020h
PUSH AX
```

is equivalent to the instruction PUSH word 0000h. The code will use register AX, therefore saving it before and restoring afterwards is mandatory.

Setting a register to zero

This is a variation of the previous code segment, however in this case the value of a register must be set to zero. This is also achieved through the stack. The code fragment that is equivalent to XOR reg, reg is:

```
PUSH 2020h
POP  AX
XOR  AX, 2020h
PUSH AX
POP  register
```

Setting register BP to any value

In several cases it will be necessary to access data on the stack, for example using the BP register, like:

```
MOV reg, [BP+offset]
```

The problem in this case is with the ‘offset’, since that number will become the part of the compiled bytecode of the instruction, therefore this number must also be printable. To solve this situation the data will always be shifted on the stack by a printable value, for example 20h. To achieve this shifting the following code fragment can be used:

```
PUSHA      ; SUB BP, 16
PUSHA      ; SUB BP, 16
PUSH BP    ; SUB BP, 2
PUSH SP    ; MOV BP, SP
POP  BP
```

In the code fragment next to the instructions the equivalent instruction is also shown as a comment after the semi-column. The PUSHA instruction uploads the AX, CX, DX, BX, SP, BP, SI and DI registers and reduces the value of SP by 16. After this code fragment data can be accessed on the stack, for example using the [BP+34] address.

After the actual operation the stack should be restored using the POPA and the POP instructions. However, when the result of the operation is stored in a register a different method is required. In the current implementation the selected method is to execute the POP instruction several times for individual registers. An example can be seen later.

The following sections discuss the details of the implementations of the most important instructions.

3.2. The MOV instruction

The MOV instruction is probably the most important instruction, since this instruction moves data between the registers and/or the memory. Its general form is:

```
MOV op1, op2
```

where the contents of ‘op2’ is copied into ‘op1’. ‘op1’ and ‘op21 can be a register or a memory location, but both of them cannot denote a memory location at the same time. ‘op2’ can also be a constant number, but ‘op1’ cannot be a constant. To create equivalent instructions with printable bytecodes several cases have to be considered.

Copying data between 16 bit registers

This is the simplest case when the full content of a 16 bit register has to be copied to another register. For example for the instruction:

```
MOV AX, BX
```

first the content of register BX must be uploaded to the stack, then popped to register AX. Therefore the equivalent code fragment is:

```
PUSH BX
POP  AX
```

This method can be extended to other registers and all these instructions result in printable bytecodes.

Copying a 16 bit value to a 16 bit register

In this case a 16 bit number is must be copied to a 16 bit register and the general syntax of the instruction is:

```
MOV BX, number
```

where the number can be in the range of 0 and 65535. Technically there is an easy solution, where zero is copied into the AX register then incremented as many times as required. For example:

```
MOV BX, 0003
```

can be implemented in an equivalent way as follows:

```
PUSH AX          ; save register AX
PUSH 8224
POP  AX
XOR  AX, 8224    ; set register AX to zero
INC  AX
INC  AX
INC  AX          ; increment three times
PUSH AX
POP  BX          ; copy to register BX
POP  AX          ; restore register AX
```

Unfortunately this implementation is very inefficient especially for large numbers. To avoid this problem several techniques have been employed for different ranges of numbers, for example for the following ranges:

- When number < 127 then the above described straightforward technique is used.
- When 127 < number < 8225 then two operations (an AND and a XOR) are used to set the register to the required value. The following code fragment demonstrates the method that is developed in this paper for this case:

```
MOV AX, 0ffffh  ; AX = 65535
AND AX, op1
XOR AX, 16254
```

  To determine the operand of the AND operation a mathematical formula can be used: op1 = (63 − d) × 256 + 94 where d = int(number/256). The formula ensures that op1 16 bit number can be represented by two printable characters. The division to determine value d is an integer division and the remainder (r = rem(number/256)) is also required, since the XOR operations with its constant operand does not provide the exact number in register AX. There are three sub-cases depending on the remainder:

  - if the remainder is smaller than 32 then instruction DEC AX is repeated for 32 - r number of times;
  - if the remainder is smaller than 143 then instruction INC AX is repeated for r - 32 number of times;
  - otherwise instruction DEC AX is repeated for 288−r number of times;

- When 8225 < number < 32382 then there are cases when a single operation is enough to assign the required value to register AX. The reason for this is that the numbers that can also be represented by printable characters fall in this range;
- For other ranges of numbers similar techniques are used but they are not presented here in detail for the sake of brevity.

Copying a 8 bit (byte) value to an 8 bit register

This kind of data copying requires a different approach, since by default there are 16 bit values on the stack. When the data should be copied to the high 8 bit register (e.g. AH or BH, etc), then the following code fragment can be the base:

```
PUSH CX       ; register for intermediate storage
PUSH AX       ; ***
PUSHA
PUSHA
PUSH BP
PUSH SP
POP  BP
XOR  AH, [BP + 35]  ; AH = 0
POP  BP
POP  CX
...                 ; 16 times POP CX
POP  CX
```

In this code fragment the XOR instruction accesses the high byte of the instruction marked by three stars, therefore register AH becomes zero.

After this an addition is simulated between the 8 bit register and the 8 bit value. This is discussed later. When the data should be copied to the low 8 bit register the approach is similar, but instruction XOR changes as:

```
XOR  AL, [BP + 34]
```

Other addressing modes

There are several other addressing modes that can be solved by the combination of the previously described methods. For the sake of brevity they are not discussed here.

3.3. The arithmetic instructions

There are several arithmetic operations, like addition, subtraction, multiplication and division, but basically all of them can be simulated by addition. Addition is also one of the most common operations, therefore only its implementation will be discussed here. The syntax of the addition instruction is:

```
ADD  op1, op2
```

where ‘op1’ and ‘op2’ can be registers and memory addresses, but they cannot be memory addresses at the same time. This instruction adds ‘op2’ to ‘op1’ and stores the result in ‘op1’. As an example the implementation of an instruction that adds together 16 bit registers will be discussed. In this case register AX must be used to create instructions with printable bytecodes. The addition will be simulated by three subtractions and the stack will also be utilized. In this regard the following values are uploaded to the stack: 0, ‘op1’ and ‘op2’. Using this stack the following instructions can simulate the addition: ADD DX, BX:

```
PUSH AX           ; saving AX
PUSH 8224
POP  AX
XOR  AX, 8224     ; AX = 0
PUSH AX           ; 0 on stack
PUSH DX           ; op1 on stack
PUSH BX           ; op2 on stack
PUSHA
PUSHA
PUSH BP
PUSH SP
POP  BP
SUB AX, [BP + 34]
SUB AX, [BP + 36]
SUB [BP + 38], AX
POP BP
POPA
POPA
POP BX          ; remove op2 from stack
POP DX          ; remove value from stack
POP DX          ; setting DX to result from stack
POP AX          ; restore AX
```

where the meaning of the first subtraction is 0 - op2, the meaning of the second subtraction is (0 - op2) - op1 and the meaning of the third subtraction is 0 - ((0 - op2) - op1), which is equal to op2 + op1. The result of the third subtraction is stored on the stack and later can be popped from it.

When a 16 bit number must be added to a register, the technique is the same, since the only thing that has to be done is to copy the number to a temporary register. Similarly copying an address to a register can be easily solved with this technique. Furthermore the 8 bit variation can be implemented in the earlier discussed way.

From this discussion, it can be seen that instruction SUB is compatible with printable bytecodes, therefore they can be used without any translation. Similarly the instructions to increment or decrement registers or memory places can be replaced by addition or subtraction.

3.4. The CMP instruction

A basic instruction to control the execution of a program is instruction CMP, which compares two operands. The general form of the instruction is:

```
CMP op1, op2
```

It should be noted, that this instruction is equivalent to a subtraction (op1 - op2) and it sets all of the status bits at the same time, which means that technically all combinations of comparisons is performed: equal, not equal, smaller, greater, smaller or equal and greater or equal. Once the status bit is set a conditional jump can be performed.

In the implementation of instruction CMP it is important that the status bits do not change, therefore only stack operations should be used. Another important note that register AX is used as one of the operands in every comparison. For the second operand a value on the stack is used, therefore the technique presented in Section 3.1 can be used again. For example the instruction CMP BX, CX can be implemented in an equivalent way:

```
PUSH AX
PUSH BX
POP  AX     ; op1 in AX
PUSH CX     ; op2 on stack
PUSHA
PUSHA
PUSH BP
PUSH SP
POP  BP
CMP  AX, [BP + 34]
POP  BP   ; no status bit should change below
POPA
POPA
POP  CX
POP  AX
```

The other comparison instructions can be implemented in the same way, for example comparison of a 16 bit register with a number, comparison of an 8 bit registers and comparison of an 8 bit register with a number.

3.5. Handling of interrupts

The COM programs use software interrupts to communicate with the operating system. The most common operations performed with interrupts are reading character(s) and printing on the screen. Unfortunately at the moment the only method to perform equivalent operation with printable bytecodes is to use a self-modifying code. In this case however two bytes has to be created. Generally instruction INT XX generates two bytecodes: CD XX, where XX is the number of the interrupt. The aspect that the self-modifying code has to consider is that the Intel architecture uses little-endian storage, therefore in the memory actually XX CD will be stored. To create an equivalent instruction set with printable bytecodes the following should be considered:

```
PUSH SI
...                ; MOV SI, (16525 + 256 * XX)
XOR  [intaddr], SI
POP  SI
intaddr: dw 4040h
PUSH SI
  ...                ; MOV SI, (16525 + 256 * XX)
XOR  [intaddr], SI
POP  SI
...
```

where the setting of register SI is implemented by the previously described techniques. The code fragment contains the second XOR instruction as well, which restores the bytes at address 'ntaddr' to their original values. As discussed above it is necessary if the code fragment is used for example in a loop.

It should be noted that this technique works for software interrupts between 00-60. Although there are 256 different interrupts in the interrupt table, but the most common interrupts fall in this range, therefore only this variation has been implemented.

3.6. The jump instructions

In the assembly language the unconditional jump instruction is also important to control the execution of programs. The general syntax of the instruction is:

```
JMP op1
```

where 'op1' can be an address or a register. In the following the instruction with explicit address will be discussed, since this is the most often used version of the instruction. Theoretically it is possible to replace the unconditional jump instructions with a conditional jump instruction, since the conditional jump instructions has printable bytecode. However in this case the appropriate condition has to be created before every jump instruction. For example when the result of the last operation is zero then the JZ instruction will jump to the given address. Although this method may work, but the other problem with the default unconditional and the conditional jump instructions is that they use a relative address. This means that the assembler will calculate the distance between the address of the actual position and the address of the jumping position. There are two problems with this approach. One of the problems is that this distance is not known, since the assembly code is transformed to a new source code. At the time of the compilation the transformed code after the current position is not known. The second problem is that operand must also be printable byte.

The current solution to the above described problem is to use an unconditional jump instruction, which uses absolute addresses, with segments and offsets. The byte codes of the instruction are: EAh, XXh, XXh, YYh, YYh, where XXh and YYh are hexadecimal numbers. Furthermore on the Intel architecture the segment and offset data is store with the little-endian convention. This instruction can only be implemented by a self-modifying code. Since other instructions also utilize this solution therefore the size of the code must be smaller than 127 bytes, as discussed in the following subsection. Finally this instruction may also occur inside a loop, therefore the code has to modify itself and then modify back to its original bytecode, and thus it can be called repeatedly. The following code is proposed:

```
PUSH
PUSH 02020h
POP AX
PUSH AX
POP BX
PUSH BX
POP DI
```

```
    PUSH 03e3eh
    POP DX
    SUB AX, 03536h          ;1
    XOR [jump1], AH         ;2
    XOR [jump3], DH         ;3
jump3: db 020h
   POP SI                   ;4
   XOR [jump3], DH          ;5
   XOR [jump1 + 3], SI      ;6
   PUSH arg2
   POP SI
   XOR [jump1 + 1], SI      ;7
   XOR [jump1], DI          ;8
   XOR [jump1 + 2], DI      ;9
   PUSH byte 020h
   POP DI
   XOR [jump1 + 4], DI      ;10
   JP jump2                 ;11
   XOR [jump1], AH          ;12
   XOR [jump4], DH
jump4: db 020h
   POP SI
   XOR [jump4], DH
   XOR [jump1 + 3], SI
   PUSH arg2
   POP SI
   XOR [jump1 + 1], SI
   PUSH BX
   POP DI
   XOR [jump1], DI
   XOR [jump1 + 2], DI
   PUSH byte 020h
   POP DI
   XOR [jump1 + 4], DI
jump2:
   POPA
jump1: db 020h, 020h, 020h, 020h, 020h
```

After the semicolon a number is placed as a comment, which is referred below:

1. From the beginning of the code to comment 1 the code only saves the registers and sets the required values in them;
2. This instruction creates the byte code EAh, which is the instruction code;
3. The PUSH DS instruction is created at position `jump3';
4. The segment register DS, which is on the stack, is stored in register SI;
5. The byte code at address `jump3' is restored to its original byte;
6. The segment value is stored in the last two bytes of the jump instruction;
7. The offset address, used in the original jump instruction, is stored in the second and third bytes of the jump instruction;

8-11. These lines create the final byte code of the jump instruction. At address 'jump1' the original bytes are 20h. These values are XOR-ed with the required

value by earlier instructions. Here this new value is XOR-ed with 20h again, therefore only the required values will remain at the address. Furthermore there are three XOR instructions here. Their order can be arbitrary, thus the last XOR instruction is used to determine whether this code segment is in an odd or even iteration;

12. Instructions between comment 1 and 10 are repeated here.

Conditional jump instruction

The bytecode of the conditional jump instructions are printable, therefore theoretically there is no problem with them. However in the case of conditional jump instructions the second byte, after the instruction byte, is the distance between the current position and the target position. This byte actually represents a signed integer; therefore its range is from -128 to +127. Naturally not all these values are printable, which is the source of the problem in this case. One of the possible solutions is to rewrite a conditional jump instruction into an equivalent set of instructions. For example the instruction JZ addr (jump if zero) is equivalent to:

```
JNZ @@over
JMP addr
@@over:
```

In this case the above discussed implementation of the unconditional jump instruction can be used. If the length of this implementation is designed carefully, then the byte after instruction JNZ, which represent the relative distance of the jump, will be a printable byte. Another benefit of this method is that in this case the implementation is direction independent. It means that it does not matter whether the original instruction JZ jumps forward or backward, since this problem is handled by the implementation of the unconditional jump instruction implementation.

Handling of addresses

Labels are important in assembly programs, since they represent memory addresses symbolically. In an assembly program labels are used by conditional or unconditional jump instructions and they can also denote the address of a data. In both cases there are two difficulties. First of all in the source code labels are only symbolic. This means that modern compilers usually use a two pass technique to determine their actual value. In the first pass all instructions are translated to byte code and position and references of labels are only remembered. In the second pass when the byte code of all instructions is known the compiler fills the missing information in the byte code. The second difficulty with labels is that the address that they represent becomes a part of the instructions. Naturally in the current method the address should have a printable byte code.

The current solution for the first problem is that there is a direct translation between the original instruction and the instructions with the printable byte code, therefore the length of the transformed code is always known. Furthermore all addresses are placed at the end of the code at a position which has an address that is printable bytecode. The

space between the end of the code and position of the first labels are filled with some random printable characters.

3.7. Instruction LAHF

This instruction loads the status bits into register AH. This instruction can be important since this can be used to store the status bits that are used by the conditional jump instructions. On the other hand the implementation of the instruction is difficult since for example a self-modifying code cannot be used as the used instructions may also modify the status bits.

The implementation of this instruction uses conditional statements and depending on the value of the examined status bit different values are pushed on the stack. When the value of the status bit is one then 8225 otherwise 8224 are pushed on the stack. The implementation should not contain any jumping backward and the jumping forward should cover larger distance than 20h, as only these values will result in printable bytes. However to fill the distances only stack operations can be used as they are the only instructions that do not modify the status bits. The implemention of the instruction therefore is the following:

```
    PUSH BX
    PUSH AX          ; save the registers
    JNC s1           ; examining carry bit
    PUSH 8224
    jc s2
    ...              ; 30 bytes of filling
s1:
    PUSH 8225
s2:
    JNZ s3           ;examining zero bit
    PUSH 8224
    JZ s4
    ...              ; 30 bytes of filling
s3:
    PUSH 8225
s4:
    JNP s5           ; examining parity bit
    PUSH 8224
    JP s6
    ...              ; 30 bytes of filling
s5:
    PUSH 8225
s6:
    JNS s7           ; examining sign bit
    PUSH 8224
    JS s8
    ...              ; 30 bytes of filling
s7:
    PUSH 8225
```

```
s8:
   PUSH 8736
   POP ax
   SUB ax, 8224
   PUSH ax
   POP bx          ; initial value of BX (512)
   POP ax          ; sign bit in AX
   CMP ax, 8224
   JNE s9
   ...             ; add bx, (128 * 256)
   ...             ; 20 bytes of filling
s9:
   POP ax          ; value of parity bit
   CMP ax, 8224
   JNE s10
   ...             ; add bx, (4 * 256)
   ...             ; 20 bytes of filling
s10:
   POP ax          ; value of zero bit
   CMP ax, 8224
   JNE s11
   ...             ; add bx,(64 * 256)
   ...             ; 18 bytes of filling
s11:
   POP ax          ; value of carry bit
   CMP ax, 8224
   JNE s12
   ...             ; add bx,(1 * 256)
   ...             ; 20 bytes of filling
s12:
   POP ax
   ...             ; mov ah, bh
   POP bx
```

The triple dots with comments represent instructions in the comment that should be implemented with printable byte codes.

4. Implementation of the compiler

The above described method has been implemented in a compiler, which reads in an assembly source and outputs another assembly source code. The input and output source code can be compiled by the NASM assembler [12]. The current implementation uses the GNU flex [13] for the tokenization and GNU bison [14] for the parsing phase. After parsing the compiler directly translates the instruction to a series of instructions with printable bytecode. There is no optimization during the generation.

The implementation of the compiler has been extensively tested on three levels. First every instruction has been tested individually. In this case the content of the registers has been recorded before and after the instruction and compared with the required results. On the second level several combinations of the instructions have been tested.

Unfortunately in this case the transformed code may contain several hundreds of instructions, therefore only the input and the output have been compared with the required results. Finally several full working assembly programs have been transformed and tested. In this case again only the input the output effect has been observed and compared.

5. Testing and examples

The following assembly code will print an 8x8 square:

```
org 100h
   MOV AH,2
   MOV CX, 8
outer:
   PUSH CX
   MOV CX, 8
inner:
   MOV DL, 58h
   INT 21h
   LOOP inner
   ; new line
   MOV DL, 0dh
   INT 21h
   MOV DL, 0ah
   INT 21h
   POP CX
   LOOP outer
   INT 20h
```

The size of the original compiled code is 38 bytes. After the transformation the compiled assembly program has the size of 9837 bytes. Since the compiled version of the converted program is printable therefore it can be presented in this paper, however for the sake of brevity only the beginning of the code is presented here. It has to be noted that unfortunately the text processor does omit some of the spaces at the end of the line, therefore this representation is not perfectly reproduced here:

```
SP``UT]2f#][[[[[[[[[[[[[[[[[[SPh                                    X5
H%^=5~?HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHP[XPh                        X5
PS``UT]+F"+F$)F&]aa[XX[Ph                                            X5
@@@@@@@@PYX`V^V[R^PZPYPZSZRXRZP^RXSZQXR^RXP[RYS[P[VXQXP[SYQYQYR^S
YRZPXSXQXS^VYVZVXRYSYPZPYP^P[Q^VZS[PXSZS[SYR^P^VYVXR^Q[QXR^Q[PYP^
SXSYQZP[S[RXRXV[RZPYVZVXV[PZR[RZSXPXQXPZPXRYR^S^QZVZQ[S[S^Q[RXS^P
XS[S^PZQ[VZR[QXSZVZS^S[VYVZQ^QXPXQXR^S^R^P^RYV[SZQ^V[PZVZQ^S^Q^QX
P^SXQXP[S[RYSYPZRYQZQZPYV^QYP[VXS^QZVXQYQXSZVXSXP^Q^QXRZVZP[R[QZS
^PXV[SYRZVZR[P^PZS^P^RXS[Q[PYR^R^RXR[Q[VXR^S^PZR^RZP[P[SYR[V^V^V[
Q[R^V[VZSXRYQ[P[QYSYP^Q[P[R[QXS[V[RYQYPXSXQXR[S^VYQ^QZP[SYVYQYQXS
YPXS[VZS[SYQXRYV[SYQ[Q^S[Q^Q[PXQ^Q^V[P[QXR[PZS^PYRZQ[RXR^PYP^SYSY
PYRZVXQYQZSZPYVXPYVYV[S[QZVZR[V[S^Q[V^VYV^P^VXVXVZSYV^VZP[SZRYQ[P
[PZRZV[P^SXVYP[PZS^P[QZQ^RZR^RYQ[R[QZPYQZQXP[PXS[RZVXPZV^SYR^QYRX
```

```
V[QZVZPYS^Q^PXRYRYSYPXR^P[RYPYSXSYP^PYSZVXQYP^SYP[RYQXQYSZRZR^RXS
XVXP^QZQ^VYSXPXSYQZV^S[SXPZVXVZV[SZSXSYS^QYR[PXVXQXQXQYP[PXP^QZP[
S[S[R^V^P[RYVXRZQYRYSXQ[QZQ[SZVZPXRXP[V^Q^Q^V[R^QXQ^PXPYSYQ^PZSXR
YQXRZRZQ[QXR^SXQ[SYPXSZS^VZR[QYR^PXRXQZV^Q^VXRYQXSYS[SYPZR[PYRXR^
VYSXSZRZV^V^R[V^PXV^PYQ^QYQXS^S^P^QYR[R[S^V[QYPZP^Q^QYS^QZPXRZQ[P
XRZP^S^V^V[P[R[QXVZQZS[R^R^VYV[SXSYQXSXVZVXSYRXP[VXSZQ^S^V[P[VYQ^
R[RXR[V^PYP^SXQYR^VYP^RYPZPYPZRYV^QYVXQZQYPZQ^PXPYS[QYV
```

Conclusions

The paper has presented a method to convert full assembly programs into another assembly program, which can be compiled into only printable bytes. The method has been implemented as a proof of concept. The number of implemented instructions is 60 and 40 different variations of the MOV instruction are also implemented. Naturally the method can be extended to further instructions. More importantly the implementation of the instructions can be optimized or in other words, their size can be reduced by 20-50% according to a preliminary analysis.

It must be mentioned, that an interesting future application of this method is to use it as an obfuscator. An obfuscator performs a code transformation or conversion, but in this case the output result would be very difficult to be read and understood by a human. Looking at the final result of the above discussed method it can be stated that the method is a good starting point for a code obfuscator.

References

[1] Rix, Writing ia32 alphanumeric shellcodes, *Phrack Magazine*, Vol. 57, 2001, http://phrack.org/issues/57/15.html, (last visited 2 January 2017).

[2] Verma N., Mishra V., Singh V. P. Detection of alphanumeric shellcodes using similarity index, *International Conference on Advances in Computing, Communications and Informatics* (ICACCI), delhi, India, 24-27 Sept. 2014, pp. 1573−1577.

[3] Polychronakis M., Anagnostakis K. G., Markatos E. P. Network-level polymorphic shellcode detection using emulation, *Journal in Computer Virology*, Vol. 2, No. 4, 2007, pp. 257–274.

[4] Khodaverdi J. Enhancing the effectiveness of shellcode detection by new run-time heuristics, *International Journal of Computer Science Research and Application*, Vol. 3, No. 2, 2013, pp. 2–11.

[5] http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/ whitepaper shell code.html.php, (last visited 13 May 2015).

[6] ALPHA3 - Alphanumeric shellcode encoder, https://code.google.com/p/alpha3/, (last visited 13 May 2013).

[7] Basu A., Mathuria A., Chowdary N. Automatic generation of compact alphanumeric shellcodes for x86, *Proceedings of the 10th International Conference on Information Systems Security*, ICISS 2014, A. Prakash and R. Shyamasundar (Eds.) Springer, 2014, pp. 399–410.

[8] Mason J., Small S., Monrose F., Macmanus G. English shellcode, *Proceedings of the 16th ACM conference on Computer and communications Security*, S. Jha and A. Keromytis, (Eds.) ACM Press, 2009, pp. 524–533.

[9] obscou, Building ia32 'unicode-proof' shellcodes, *Phrack Magazine*, Vol. 61, 2003, http://phrack.org/issues/61/11.html, (last visited 2 January 2017).
[10] Wana T. Writing utf-8 compatible shellcodes, *Phrack Magazine*, Vol. 62, 2004, http://phrack.org/issues/62/9.html, (last visited 2 January 2017).
[11] Kumar P., Chowdary N., MathuriaA. Alphanumeric shellcode generator for ARM architecture, *Third International Conference on Security, Privacy, and Applied Cryptography Engineering*, SPACE 2013, Kharagpur, India, 19-23 October 2013, pp. 38−39.
[12] *The Netwide Assembler*, http://www.nasm.us, (last visited 2 January 2017).
[13] *The Fast Lexical Analyzer*, http://flex.sourceforge.net/, (last visited 2 January 2017).
[14] *GNU Bison - The Yacc-compatible Parser Generator*, https://www.gnu.org/software/bison/, (last visited 2 January 2017).