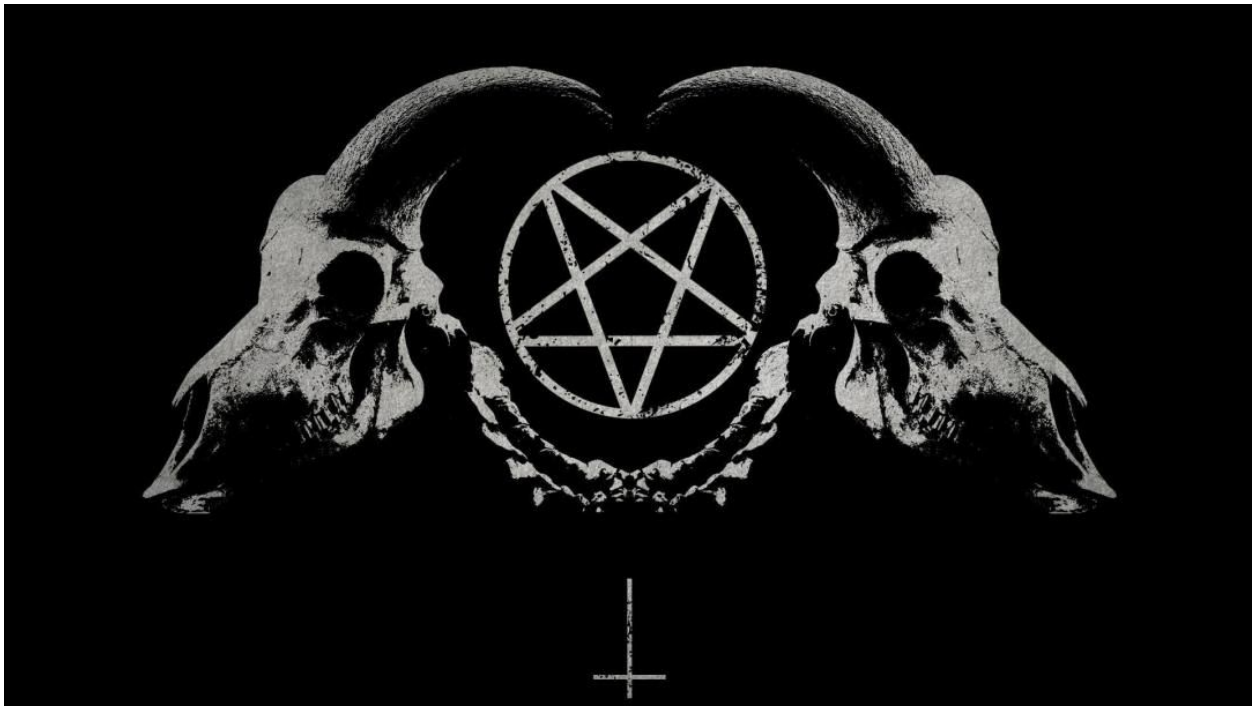
Code Transformation and Finite Automatons

vx-underground.org archive // z0mbie



There are such interesting objects as finite automata. This automaton has some internal states (just such variables), so when they change, some action is performed. So, there is a set of actions, and each action is performed when some automaton's state is changed to some specified value, and there is a matrix(es), which defines state changes.

Here is a simple task: to convert strings like "1,3-7,9-100,105" into binary form. Let's consider implementation, based on finite automata. Lets introduce single internal state S, and its values will be:

0 - before first (before '-') number
1 - inside of first number
2 - before second (after '-') number
3 - inside of second number
4 - end of string (after last char)
5 - syntax error

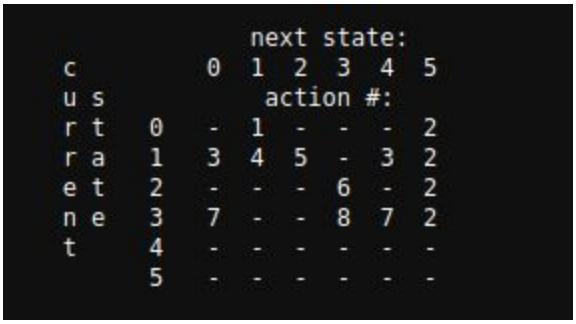
so, program will look as follows:

c - pointer to current character
*c - current character's value
A - action #

```
change   char   A  action:          situation:

0-->1    0..9   1  l=*c-'0';        new pair, got digit
0-->5    other                     new pair, got NOT a digit
1-->0     ,     2  store(l,l);      single number, ends with ','
1-->1    0..9   3  l=l*10+*c-'0'    adding new digit to first number
1-->2     -                        two numbers, 1st one ends with '-'
1-->4    eoln   2  store(l,l);      single number, ends with eoln
1-->5    other                     unavailable char within first number
2-->3    0..9   4  h=*c-'0';        two numbers, got digit after '-'
2-->5    other                     two numbers, got NOT a digit after '-'
3-->0       ,   5  store(l,h);      two numbers, ends with ','
3-->3    0..9   6  h=h*10+*c-'0';   adding new digit to second number
3-->4    eoln   5  store(l,h);      two number, ends with eoln
3-->5    other                     unavailable char within first number
4-->            7  exit();         eoln, all ok
5-->            8  error();        syntax error
```

So, matrix which describes state changes and corresponding actions, will look as follows:



current state	next state: 0	1	2	3	4	5
	action #:					
0	-	1	-	-	-	2
1	3	4	5	-	3	2
2	-	-	-	6	-	2
3	7	-	-	8	7	2
4	-	-	-	-	-	-
5	-	-	-	-	-	-

Simple source in C will look as follows:

```
int S = 0;
char*c = "1,3-7,9-100,105";

for(;;)
{
  switch(S)
  {
        case 0:
        if ( isdigit(*c)  ) { S=1; l=*c-'0';     } else
                            { S=5;               };
        break;
        case 1:
        if ( *c == ','    ) { S=0; store(l,l);    } else
        if ( isdigit(*c)  ) { S=1; l=l*10+*c-'0'; } else
        if ( *c == '-'    ) { S=2;               } else
        if ( *c == 0      ) { S=4; store(l,l);    } else
                            { S=5;               };
        break;
        case 2:
        if ( isdigit(*c)  ) { S=3; h=*c-'0';     } else
                            { S=5;               };
        break;
        case 3:
        if ( *c == ','    ) { S=0; store(l,h);    } else
        if ( isdigit(*c)  ) { S=3; h=h*10+*c-'0'; } else
        if ( *c == 0      ) { S=4; store(l,h);    } else
                            { S=5;               };
        break;
        case 4:
        exit();
        break;
        case 5:
        error();
        break;
  }
  c++;
}
```

Now, if we will make source listed above a bit more low-level, and remove state-variables, we will get the following:

```
x:
  x0:
                if ( isdigit(*c)  ) { l=*c-'0';     c++; goto x1; } else
                                    {               c++; goto x5; };
  x1:
                if ( *c == ','    ) { store(l,l);   c++; goto x0; } else
                if ( isdigit(*c)  ) { l=l*10+*c-'0'; c++; goto x1; } else
                if ( *c == '-'    ) {                     goto x2; } else
                if ( *c == 0      ) { store(l,l);   c++; goto x4; } else
                                    {                     goto x5; };
  x2:
                if ( isdigit(*c)  ) { h=*c-'0';     c++; goto x3; } else
                                    {                     goto x5; };
  x3:
                if ( *c == ','    ) { store(l,h);   c++; goto x0; } else
                if ( isdigit(*c)  ) { h=h*10+*c-'0'; c++; goto x3; } else
                if ( *c == 0      ) { store(l,h);   c++; goto x4; } else
                                    {                     goto x5; };
  x4:           exit();
  x5:           error();
      goto x
```

Now, if somebody will try to reverse this code into classical C constructions, such as for, while, if-else, there will remain many redundant goto(jmp) commands, and if the number of them will be big enough, it will be very hard to understand what this code does.

Execution graphs for such code will differ from execution graphs for classical code by a larger amount of crossed links.

To reverse this code into its source form, you should select constant code blocks, enum them, then introduce states, and only after that it will be possible to continue understanding this code. But this looks as a hard task, because while optimization, original code blocks can be divided into parts, mixed and merged.

Now, let's consider a bit more complicated source:

```
int T[256] =
{
  4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  //                    , -   0 1 2 3 4 5 6 7 8 9
  0,0,0,0,0,0,0,0,0,0,0,0,2,3,0,0, 1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};

for(;;)
{
  int A;
  switch( S )
  {
      case 0:
            switch( T[*c] )
            {
            case 1:  { S=1; A=1; }; break;
            default: { S=5; A=2; }; break;
            }
            break;
      case 1:
            switch( T[*c] )
            {
            case 2:  { S=0; A=2; }; break;
            case 1:  { S=1; A=3; }; break;
            case 3:  { S=2;    }; break;
            case 4:  { S=4; A=2; }; break;
            case 0:  { S=5;    }; break;
            }
            break;
      case 2:
            switch( T[*c] )
            {
            case 1:  { S=3; A=4; }; break;
            default: { S=5;    }; break;
            }
            break;
```

```
    case 3:
          switch( T[*c] )
          {
          case 2:  { S=0; A=5; }; break;
          case 1:  { S=3; A=6; }; break;
          case 4:  { S=4; A=5; }; break;
          default: { S=5;   }; break;
          }
          break;
    case 4:
          A = 7;
          break;
    case 5:
          A = 8;
          break;
}
switch( A )
{
    case 1: { l=*c-'0';     c++; }; break;
    case 2: { store(l,l);   c++; }; break;
    case 3: { l=l*10+*c-'0'; c++; }; break;
    case 4: { h=*c-'0';     c++; }; break;
    case 5: { store(l,h);   c++; }; break;
    case 6: { h=h*10+*c-'0'; c++; }; break;
    case 7: { exit();             }; break;
    case 8: { error();            }; break;
}
```

As you can see, this sample is written without comparison commands, and conditional jmps; everything is based on switches which is equivalent to tables.

In such form our program is a cycle, consisting of three parts: in first part, external data is analyzed (here it is string of chars), and converted into some internal variables (here it is shown indirectly using matrix T[]), then state changes are calculated (here it is first switch), and at the same time action # is calculated (here it is once again the first switch and variable A), and in the third part, some action is performed (here it is 2nd switch).

And here is interesting moment: now, instead of program, we have action #'s (indexes of code blocks), and sequence of these indexes is the core of our program.

Now let's consider the following question: how next action # is generated. It is generated depending on current automaton state values and some other data.

This means that we can use tables, which will tell us which state is next, and which action should be executed on such state change.

Such tables can be converted into a function.

Requirements for this function are simple: argument of this function is a number, which contains current state values, and probably some identifier, and result of this function is a number, which contains next state values, probably another identifier, and action index.

As such, our program will just iterate the following cycle:
1. Call main function, passing state values as argument;
   then extract new state values and action index from the result.
2. Execute action by given index,
   which will probably change state values too.

Let's consider the following sample: States are register values, from EAX to EDI, and action number is an instruction value, padded with NOP's or RETN.

Then, ALL main parts of our program will be a ...function. Lets initially pass to function:

EAX = 2, ECX = 3, other regs = 0, command is NOP.
So, argument will be:

```
            00000000...00000003000000020000C390
            <--EDI->...<--ECX-><--EAX-><--cmd->
```

Lets function result then will be:

```
            00000000...00000003000000020000C341
            <--EDI->...<--ECX-><--EAX-><--cmd->
```

which means the same state values, and command = inc ecx. So, we execute given command, and ecx state is changed, and then we pass into a function this argument:

```
      00000000...00000004000000020000C341
      <--EDI->...<--ECX-><--EAX-><--cmd->
```

and so on.

For sure, argument and result values can also contain unique instruction id, in case when some opcodes are duplicated.

Interesting fact is that our function can indirectly encode many instructions, such as jumps and arithmetic commands. The only commands can not be encoded are commands modifying memory, working with other external devices and api-calls.

So, the question appears: how to build such a function.

There can be many variants, from simple tables up to neural networks.

In the sample given above, it is impossible, for sure, because there are many registers, and they can have many values, so the function will be so big so it will not fit into existing computers.

So, let's consider a simplified sample.

Argument of function is just a current state number. Result is WORD. High result byte is the first opcode byte. Low result byte is state number, and low state bit is value of ZF flag. As such, we can indirectly encode JZ/JNZ commands. Also, the state number is equal to the action index. Program will just encrypt/decrypt some asciiz text string.

```
    Source:

          ACTIONS:          STATES/ACTIONS AND STATE CHANGES:
                                       NZ          ZR
xormsg:                          -1 --> 00                 1ST BYTE:
        xor     edx, edx         00 --> 02   01 --> 02       33
cycle1: lea     esi, msg         02 --> 04   03 --> 04       BE
        mov     ecx, msg_size    04 --> 06   05 --> 06       B9
cycle2: xor     [esi], dh        06 --> 08   07 --> 08       30
        sub     dh, dl           08 --> 10   09 --> 10       2A
        inc     esi              10 --> 12   11 --> 12       46
        dec     ecx              12 --> 06   13 --> 14       49
        jnz     cycle2
        inc     dl               14 --> 16   15 --> 16       FE
        cmp     dl, 'Z'          16 --> 02   17 --> 18       80
        jne     cycle1
        retn                     18

    Here is a function, defined by arguments and results:

       Argument   Result

        f( -1 ) = 0x3300
        f(0x00) = 0xBE02
        f(0x01) = 0xBE02
        f(0x02) = 0xB904
        f(0x03) = 0xB904
        f(0x04) = 0x3006
        f(0x05) = 0x3006
        f(0x06) = 0x2A08
        f(0x07) = 0x2A08
        f(0x08) = 0x460A
        f(0x09) = 0x460A
        f(0x0A) = 0x490C
        f(0x0B) = 0x490C
        f(0x0C) = 0x3006
        f(0x0D) = 0xFE0E
        f(0x0E) = 0x8010
        f(0x0F) = 0x8010
        f(0x10) = 0xBE02
        f(0x11) = 0x0012

    Now lets generate a function.
    Let it be polynomial.

      f(X) = SUMi( C[i]*X^i ), i=0..18
```

Now let's generate a function. Let it be polynomial.

```
f(X) = SUMi( C[i]*X^i ), i=0..18
```

Here X is a function argument, which, as we decided, is the current state, and the function result

contains the next state and its first opcode byte. Simple program to calculate coefficients is here: see (1). So, now we have everything to implement function: Here it is:

```
float calc(float X)
{
	float y = 0;
	for(int j=0; j<N; j++)
	y += pow(X,j) * C[j];

	return y;
}
```

Or, in more nice form:

```
N                   equ    19

go_next_state:      pushf  ; IN/OUT: EBX=current/next state
                    pusha
                    finit
                    fild   dword ptr [esp+4*4]   ; pusha.ebx
                    push   N
                    pop    ecx
                    lea    edx, C_table
                    fldz          ; st(1)
                    fld1          ; st(0)
__c1:               fld    st(0)
                    fld    tbyte ptr [edx]
                    fmulp
                    faddp   st(2),st(0)
                    fmul   st(0),st(2)
                    add    edx, 10
                    loop   __c1
                    fistp   dword ptr [esp+4*4]  ; pusha.ebx
                    fistp   dword ptr [esp+4*4]  ; pusha.ebx
                    popa
                    popf
                    retn

C_table             label   tbyte
  dt    4.864199999999999986e+04   ; C[ 0]
  dt    1.052028658321440037e+06   ; C[ 1]
  dt    -2.226893544084362929e+06   ; C[ 2]
  dt    1.054917653361728822e+06   ; C[ 3]
  dt    1.030581898921544049e+06   ; C[ 4]
  dt    -1.641889337850409245e+06   ; C[ 5]
  dt    1.056816179457771135e+06   ; C[ 6]
```

```
  dt    -4.22626948757782734e+05   ; C[ 7]
  dt    1.17912626433683591e+05   ; C[ 8]
  dt    -2.41895494331434752e+04   ; C[ 9]
  dt    3.74924768019917908e+03   ; C[10]
  dt    -4.44669182653933311e+02   ; C[11]
  dt    4.04463907886655141e+01   ; C[12]
  dt    -2.80102010777205398e+00   ; C[13]
  dt    1.45061080738137883e-01   ; C[14]
  dt    -5.43699937869468674e-03   ; C[15]
  dt    1.39177406756125134e-04   ; C[16]
  dt    -2.17485012359577303e-06   ; C[17]
  dt    1.56344362992516363e-08   ; C[18]

Since first opcode byte is encoded within high byte of function result,
it will be defined in our program as ?:

                        .data

msg                     db      'Hello world!',0
msg_size                equ     $-msg

start:
                        call    xormsg

                        push    0
                        push    offset msg
                        push    offset msg
                        push    0
                        callW    MessageBoxA

                        call    xormsg

                        push    0
                        push    offset msg
                        push    offset msg
                        push    0
                        callW    MessageBoxA

                        push    -1
                        callW    ExitProcess

xormsg:
                        xor     ebx, ebx        ; initial state = -1
                        dec     ebx

                        jmp     begin

cycle:                  pushf                   ; copy ZF into bit 0 of EBX
                        push    eax             ;
                        lahf                    ;
```

```
                    shr     ah, 6 ; ZF  ;
                    and     ah, 1         ;
                    and     bl, 0FEh      ;
                    or      bl, ah        ;
                    pop     eax           ;
                    popf                  ;
begin:
                    call    go_next_state ; get next state

                    cmp     bl, 18        ; exit if last state is reached
                    jae     quit

                    push    eax ecx       ; take 1st opcode byte from
                    mov     cl, bh        ; function result, and patch code
                    mov     bh, 0
                    mov     eax, jtab[ebx*4]
                    mov     [eax], cl
                    pop     ecx eax

                    jmp     jtab[ebx*4] ; go to action

quit:               retn

jtab                label   dword         ; table of actions
                                          ; index = action # = state #
                    dd      s00,s01
                    dd      s02,s03
                    dd      s04,s05
                    dd      s06,s07
                    dd      s08,s09
                    dd      s10,s11
                    dd      s12,s13
                    dd      s14,s15
                    dd      s16,s17
s00:
s01:                ; xor   edx, edx
                    db      ?,0D2h
                    jmp     cycle
s02:
s03:                ; lea   esi, msg
                    db      ?
                    dd      offset msg
                    jmp     cycle
s04:
s05:                ; mov   ecx, msg_size
                    db      ?
                    dd      msg_size
                    jmp     cycle
s06:
s07:                ; xor   [esi], dh
```

```
                    db      ?,36h
                    jmp     cycle
s08:
s09:                ; sub   dh, dl
                    db      ?,0F2h
                    jmp     cycle
s10:
s11:                ; inc   esi
                    db      ?
                    jmp     cycle
s12:
s13:                ; dec   ecx
                    db      ?
                    jmp     cycle
s14:
s15:                ; inc   dl
                    db      ?,0C2h
                    jmp     cycle
s16:
s17:                ; cmp   dl, 'Z'
                    db      ?,0FAh,'Z'
                    jmp     cycle
```

Now, to understate the logic of this compiled program, you should calculate all function results for all state values, then build a table of state changes, and insert JZ/JNZ into corresponding program places. After that, if the program is NOT builded using the method shown in the 1st part of this article, i.e. not based on finite automaton conception, it will be possible to reverse it into classical C constructions, such as if, for, while.

Now, a bit of theory.

All these code transformations can be applied to mostly all blocks of existing code, i.e. any linear code can be converted into finite automaton, and then the state change table can be converted into a function.

Let's assume that the variable state of automaton's state is not ZF flag, as it shows above, but AL register value. This means that to extract all information encoded in our function, it is required to iterate 256 variants. If this would be an AX register, then, for example on checking two first bytes of some file for MZ sign, the current state will be changed into one state in 65534 cases, and into another state in 2 cases. The bigger argument size is, the more difficult it is to iterate it; and in some cases some state changes will be lost, and part of code, which, for example, follows MZ-check will not be extracted.

Ideally, such a function is a black box, which has some value on input, and outputs some other value, but extracting all the information is impossible. Well, enough words have been said, and

i'll go drink some beer, while you will think about all that crap. I hope it will help you somewhere.

Appendix (1) - program to calculate polynomial coefficients.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <io.h>
#include <math.h>
#pragma hdrstop

#define float     long double

int N = 19;

float X[100] = {-1,0,2,4,6, 8,10,12,14,16,1,3,5,7, 9,11,13,15,17 };
float Y[100] = {
                  0+0x3300,
                  2+0xBE00,
                  4+0xB900,
                  6+0x3000,
                  8+0x2A00,
                  10+0x4600,
                  12+0x4900,
                  6+0x3000,
                  16+0x8000,
                  2+0xBE00,
                  2+0xBE00,
                  4+0xB900,
                  6+0x3000,
                  8+0x2A00,
                  10+0x4600,
                  12+0x4900,
                  14+0xFE00,
                  16+0x8000,
                  18+0x0000 };

float C[100];

float calc(float X)
{
```

```
	float y = 0;
	for(int j=0; j<N; j++)
	y += pow(X,j) * C[j];
	return y;
}


void init()
{
	float* Z = new float[ N*(N+1) ];
	assert(Z);

	#define Zyx(y,x)		Z[y*(N+1)+x]

	for(int y=0; y<N; y++)
	{
	for(int x=0; x<N; x++)
	Zyx(y,x) = pow(X[y], x);

	Zyx(y,N) = Y[y];
	}
	for(int n=0; n<N-1; n++)
	for(int y=n+1; y<N; y++)
	{
		float t = Zyx(y,n) / Zyx(n,n);
		for(int x=0; x<=N; x++)
			Zyx(y,x) -= Zyx(n,x) * t;
	}

	for(int n=N-1; n>=0; n--)
	{
	float s = 0;
	for(int t=n+1; t<=N-1; t++)
		s += Zyx(n,t) * C[t];

	C[n] = (Zyx(n,N) - s) / Zyx(n,n);
}

	delete Z;

	for(int i=0; i<N; i++)
	assert(abs(calc(X[i]) - Y[i]) < 0.0001);
```

```
}

void main()
{
	init();

	for(int n=0; n<N; n++)
	printf("f(%5.2Lf) (%02X) = %5.2Lf (%04X)\n", X[n],

	(int)ceil(X[n]), Y[n], (int)ceil(Y[n]));

	printf("f(X) = SUMi( C[i]*X^i ), i=0..%i\n", N-1);

	for(int n=0; n<N; n++)
	printf("dt %30.19Le  ; C[%2i]\n", C[n], n);

}
```