TECHNICAL FEATURE

FLIBI: EVOLUTION

Peter Ferrie
Microsoft, USA

The Flibi virus demonstrated that a virus can carry its own 'genetic code' (see *VB*, March 2011, p.4), and if the codons[1] (the p-code form of the virus), the tRNA[2] (the translator function), or the corresponding amino acids[3] (the native code) are mutated in some way, then interesting behaviours can arise.

Each codon is used as a relative offset into a table of amino acids. There is a single pointer to the table. Mutation of a codon might cause a new amino acid to be produced, since it might now point to a different entry in the table. Mutation of the pointer would almost certainly be fatal since many codons would not be translated into the correct amino acids. Mutation of the amino acid itself might produce new behaviour, depending on the change. For example, a shift could become a rotate.

The virus has the ability to move a sequence of codons to a later position in the stream[4], and then fill the gap with no-operation instructions. In most cases, this simply results in the replacement of the codons at the destination[5]. Of course, if the selected sequence appears at the end of the defined stream (there is a lot of slack space after the last meaningful codon), then the size of the defined stream will increase slightly each time that condition occurs. However, the size of the buffer remains fixed. Therefore, new sequences can only appear when the translator code is modified to increase the number of codons that are translated, thus 'translating' garbage beyond the original end of the stream. That garbage could potentially be modified over time to eventually produce meaningful functionality. Its location in the virus body would change over time as a result of the codon deletion, allowing the new amino acids to 'migrate' to a final position where they become truly useful. The human eye did not spring fully formed from the dust of the earth but was the result of gradual refinements in image accuracy. Something similar can occur here, where the sequence of amino acids does not need to work completely (or even at all) in order to be useful (or just retained). As unlikely as these things are, millions of computer years from now, we might see some of the following transformations.

The aim of this article is to demonstrate how some instructions from the original set might be removed by replacing them with functionally equivalent code sequences using the remaining instructions. One advantage of a smaller instruction set is that it allows an increase in the number of codons that can map to a single amino acid, thus making the body more resilient to corruption. Further, a sequence of instructions has a smaller risk of lethal mutation than a single instruction, because the risk is spread over a wider area.

We begin with a brief overview of the language itself. There are 45 commands in the release version (there were only 43 in the preview version). There are three general-purpose registers ('A', 'B' and 'D', which correspond to the 'eax', 'ebp' and 'edx' CPU registers); one temporary register, upon which all operations are performed (which corresponds to the 'ebx' CPU register); one 'operator' register, which holds the value for any operation that requires a parameter (which corresponds to the 'ecx' CPU register); and two buffer registers, one of which holds the destination for branching instructions (which corresponds to the 'esi' CPU register), and the other holds the destination for write instructions (which corresponds to the 'edi' CPU register).

The language supports the following commands:

- _nopsA, _nopsB, _nopsD, _nopdA, _nopdB, _nopdD
- _saveWrtOff, _saveJmpOff
- _writeByte, _writeDWord
- _save, _addsaved, _subsaved
- _getDO, _getdata, _getEIP
- _push, _pop, _pushall, _popall
- _zer0
- _mul, _div, _shl, _shr, _and, _xor
- _add0001, _add0004, _add0010, _add0040, _add0100, _add0400, _add1000, _add4000, _sub0001
- _nopREAL
- _JnzUp, _JzDown, _JnzDown
- _CallAPILoadLibrary, _CallAPIMessageBox, _CallAPISleep (release version), _call
- _null (release version, it has no actual name)

This set can be reduced in several ways. The most obvious candidates for removal are the three API calls (two in the

[1] A codon is a trinucleotide sequence of DNA or RNA (the nucleic acids that contain the genetic instructions used in the development and functioning of living organisms) that corresponds to a specific amino acid. See http://www.genome.gov/Glossary/index.cfm?id=36.
[2] Transfer RNA, or tRNA, is a small RNA molecule that is involved in protein synthesis. See http://www.wiley.com/college/boyer/0470003790/structure/tRNA/trna_intro.htm.
[3] Amino acids are the building blocks of proteins. See http://en.wikipedia.org/w/index.php?title=Amino_acid&oldid=412676887.
[4] There is a bug in this code, which can result in attempting to copy more bytes than exist in the source.
[5] There is an additional case where the destination is the same as the source, in which case the codons are deleted.

preview version[6]). The APIs can be called using the '_call' command if the API addresses are placed in the data section in this way:

```
_getDO      ;get data offset
_addnnnn    ;adjust ebx as appropriate to reach the
            ;required offset
_call       ;call the API
```

This leaves 42 commands remaining (41 in the preview version).

The '_zer0' command can be removed by using this code:

```
_save   ;ecx = ebx
_xor    ;ebx = 0
```

41 (40) commands now remain.

The '_subsaved' command (which performs the action 'ebx = ebx – ecx') can be removed, and the '_addsaved' command (which performs the action 'ebx = ebx + ecx') can be used instead, with a slight change. Specifically, the new value of the 'ecx' register is '-ecx' (such that 'ebx = ebx + -ecx'). However, there is no negate command, so an equivalent result must be achieved using the combination of operations that perform a 'not' and an 'add 1'. The problem is that a 'not' operation uses the value '0xffffffff', which requires many steps to construct. Given the existing instruction set, it would be simplest to place the value '0xffffffff' in the data section[7]. It must be placed at the start of the data section, because the '_addnnnn' commands can be removed, leaving no way to select another offset. This algorithm can then be used:

```
xor     ebx, 0xffffffff
inc     ebx
```

which we translate into this code:

```
_push
_getDO          ;get data offset
_getdata        ;fetch 0xffffffff
_xor            ;logically 'not' ebx
_add0001        ;increment result to complete negate
_save           ;replace ecx
_pop
_addsaved       ;ebx = ebx + ecx
```

40 (39) commands remain.

In the same way, the '_sub0001' command can be removed by using this code:

```
_push
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
_pop
_addsaved       ;ebx = ebx - 1
```

39 (38) commands remain.

The '_addnnnn' commands exist for convenience, but all of the commands apart from '_add0001' can be constructed using the '_add0001' command. Thus, the '_add0004', '_add0010', '_add0040', '_add0100', '_add0400', '_add1000' and '_add4000' commands can be removed.

32 (31) commands remain.

The '_add0001' command can also be removed, because the number '1' can be recovered from the value '0xffffffff' by using this code:

```
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff

;here is a horrible trick:
;modern CPUs limit the shift-count to 0x1f by taking
;the low five bits for the count and simply discarding
;the rest of the value internally, this performs a
;cl & 0x1f and it's exactly what we need

_shr            ;ebx = ebx >> cl
_save           ;ecx = 1
```

From then on, the '_addsaved' command can be used to increment the 'ebx' register as needed.

31 (30) commands remain.

Of course, it would require very many uses of the '_addsaved' command in order to construct large values, but value construction can be accelerated by using the '_shl' and '_xor' commands.

For example, constructing the value '2' is a matter of the following:

```
_shl            ;ebx = ebx << cl (ebx and ecx are '1'
                ;from above)
```

Constructing the value '3', beginning with the 'ebx' and 'ecx' registers holding the value '1', as above, is a matter of the following:

```
_shl            ;ebx = ebx << cl
_xor            ;ebx = ebx ^ ecx
```

Constructing the value '4', beginning with the 'ebx' and 'ecx' registers holding the value '1', as above, is a matter of the following:

```
_shl            ;ebx = ebx << cl
_shl            ;ebx = ebx << cl
```

And so on. Given this algorithm, we can see that the value '0xffffffff' is not the only possible 'base constant'. The value '1' could be used instead, since the value '0xffffffff' could be produced from it in the following way:

[6] The '_CallAPISleep' command was added to the release version because the API resolver code could not resolve the Sleep() API on certain platforms. The reason is described in detail in the previous article (*VB*, March 2011, p.4).

[7] It would be even simpler to introduce an instruction which performs a 'mov ebx, 0xffffffff'.

```
_getDO          ;get data offset
_getdata        ;ebx = 1
_save           ;ecx = 1
_shl            ;ebx = 2
_xor            ;ebx = 3
_shl            ;ebx = 6
_xor            ;ebx = 7
_shl            ;ebx = 0x0e
_xor            ;ebx = 0x0f
... [54 steps]
_shl            ;ebx = 0xfffffffe
_xor            ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
```

Clearly, it is far simpler to go from '0xffffffff' to '1' than the other way around. Note that values can also be constructed using the 'reverse' of this technique, to reduce the number of shifts required. For example, constructing the value '0x80000000' is a matter of the following:

```
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
_shl            ;ebx = 0x80000000
```

Constructing the value '0x40000000' is a matter of the following:

```
_push
_shr            ;ebx = ebx >> cl (ebx = 0x80000000,
                ;ecx = 0xffffffff from above)
_save           ;ecx = 1
_pop
_shr            ;ebx = 0x40000000
```

However, setting additional bits in the upper region requires more than just the '_xor' command. Here are two examples that set the same value, one using the '_shl' command and one using the '_shr' command. To construct a value such as '0xf0000000', beginning with the 'ebx' register holding the value '0x80000000' and the 'ecx' register holding the value '0xffffffff', as above, the following can be used:

```
_push
_shr            ;ebx = ebx >> cl
_save           ;ecx = 1
_pop            ;ebx = 0x80000000 again
_push
_shr            ;ebx = 0x40000000
_push
_shr            ;ebx = 0x20000000
_push
_shr            ;ebx = 0x10000000
_save           ;ecx = 0x10000000
_pop
_xor            ;ebx = 0x30000000
_pop
_xor            ;ebx = 0x70000000
_pop
_xor            ;ebx = 0xf0000000
```

Whereas, to construct the value '0xf0000000', beginning with the 'ebx' and 'ecx' registers holding the value '0xffffffff', as above, the following can be used:

```
_push
_shr            ;ebx = ebx >> cl
_save           ;ecx = 1
_shl            ;ebx = 2
_xor            ;ebx = 3
_shl            ;ebx = 6
_xor            ;ebx = 7
_shl            ;ebx = 0x0e
_shl            ;ebx = 0x1c
_save           ;ecx = 0x1c
_pop
_shl            ;ebx = 0xf0000000
```

Thus, depending on the value, the '_shl' method is the simplest.

Astute readers will have noticed that none of the value constructions above use the '_addsaved' command. This shows that constants can be constructed without using any form of 'add'. However, it is also possible to perform the addition of arbitrary values without using any form of 'add', resulting in the removal of the '_addsaved' command by using this algorithm (edx and ebp holding the values to add together):

```
eax = edx ^ ebp
do
{
        ebp = (ebp & edx) << 1
        edx = eax
        eax = edx ^ ebp
}
while (edx & ebp)
ebx = eax
```

which we translate into this code:

```
[construct here the first value to add, not shown]
_nopdD      ;edx = ebx
[construct here the second value to add, not shown]
_nopdB      ;ebp = ebx
_save       ;ecx = ebx
_nopsD      ;ebx = edx
            ;optional, depending on the order of the
            ;constructions above
_xor        ;ebx = edx ^ ebp
_nopdA      ;eax = edx ^ ebp
_getEIP
_push       ;top of do-while loop
            ;ebx points to a hidden 'pop ebx'
            ;instruction as part of _getEIP so there
            ;is no explicit 'pop' instruction inside
            ;the loop that corresponds to this 'push'
            ;instruction
_saveJmpOff ;esi = ebx
_nopsD;ebx = edx
_save       ;ecx = edx
```

```
_nopsB      ;ebx = ebp
_and        ;ebx = ebp & edx
_push
_getDO      ;get data offset
_getdata    ;ebx = 0xffffffff
_save       ;ecx = 0xffffffff
_shr        ;ebx = 1
_save       ;ecx = 1
_pop
_shl        ;ebx = (edx & ebp) << 1
_nopdB      ;ebp = (edx & ebp) << 1
_save       ;ecx = ebx
_nopsA      ;ebx = eax
_nopdD      ;edx = eax
_push
_xor        ;ebx = edx ^ ebp
_nopdA      ;eax = edx ^ ebp
_pop
_and        ;ebx = edx & ebp
_JnzUp      ;loop while ((edx & ebp) != 0)
_pop        ;discard loop address
_nopsA      ;ebx = eax
```

30 (29) commands remain.

The replacement code for the '_addsaved' command requires the use of the base constant from the data section (and here, the value '1' would result in shorter code).

The value '1' can be constructed dynamically instead, in the following way:

```
_getEIP
_getdata        ;ebx=0xxxxxxx5b
_save           ;ecx=0xxxxxxx5b
_shl
_shr            ;ebx=0x1b
_save           ;ecx=0x1b
_addsaved       ;ebx=0x36
_addsaved       ;ebx=0x51
_addsaved       ;ebx=0x6c
_addsaved       ;ebx=0x87
_save           ;ecx=0x87
_shr            ;ebx=1
```

However, that algorithm prevents the removal of the '_addsaved' command. The two concepts seem to be mutually exclusive.

It is unclear whether the '_nopREAL' command could be removed, since there is no other single-byte command that might take its place in the event that a true 'no-operation' command were required. Its current purposes are to pad the unused slots following codon deletion and to fill the unused slot(s) that follow the '_JnzDown' command (since the '_JnzDown' command skips three slots). Note that the current implementation of the '_JnzDown' command contains a bug, which is that the destination of the branch is not the start of a slot. Instead, the command branches to two bytes past the start of the slot. The result is that the '_nopREAL' command must be used to fill that destination slot, otherwise a crash could occur because the branch might land in the middle of a command. However, the '_JnzDown' command can be removed by using alternative code, and any non-stack and non-memory instruction can be used for tail padding. Thus it appears that, given its current uses, the '_nopREAL' command can be removed.

29 (28) commands remain.

In the release version a '_null' command exists, which emits a single zero into the stream, followed by the 'nop' padding. Its existence is the result of a bug. The execution of such an instruction is likely to cause an exception. It is possible on *Windows XP* and later to register a vectored exception handle using the existing language, and that could intercept the exception, but this is quite outside the 'style' of the language. The command can be removed without any problem.

28 commands remain.

The '_JnzDown' command could be removed by using a careful implementation of '_JnzUp' (given that the meaning is reversed), but perhaps not without the loss of some functionality. It requires knowledge of the location of a forward branch destination. This interferes with command reordering if the buffer size is fixed, because there might not be enough slots available to construct the required 'add' value (unless the maximum number of slots was reserved each time in order to construct any possible number). It does, however, extend the functionality in a different way, since the '_JnzDown' command can skip only three commands at a time, requiring its use multiple times in order to execute larger conditional blocks. The '_JnzDown' command also places severe restrictions on what can appear in those conditional blocks, since an arithmetic operation might clear the Z flag, causing the branch to be taken instead of skipped. In contrast, the use of the '_JnzUp' command can skip an arbitrary number of commands without restriction. The difference can be demonstrated easily. We begin with some code that calls the GetTickCount() API to fetch a 'random' number (for ease of demonstration, the offset of the GetTickCount() API is set arbitrarily to the value '0x0c'), using the '_JnzDown' command:

```
;construct pointer to GetTickCount()
;construct the value "0x0c"

_getDO       ;get data offset
_getdata     ;ebx = 0xffffffff
_save        ;ecx = 0xffffffff
_shr         ;ebx = 1
_save        ;ecx = 1
_shl         ;ebx = 2
_xor         ;ebx = 3
```

```
_shl          ;ebx = 6
_shl          ;ebx = 0x0c
_nopdB        ;ebp = 0x0c
_save         ;ecx = 0x0c

;add to data offset

_getDO        ;get data offset
_nopdD        ;edx = data offset
_xor          ;ebx = edx ^ ebp
_nopdA        ;eax = edx ^ ebp
_getEIP
_push         ;top of do-while loop
              ;ebx points to a hidden 'pop ebx'
              ;instruction as part of _getEIP
              ;so there is no explicit 'pop'
              ;instruction inside the loop
              ;that corresponds to this 'push'
              ;instruction
_saveJmpOff   ;esi = ebx
_nopsD        ;ebx = edx
_save         ;ecx = edx
_nopsB        ;ebx = ebp
_and          ;ebx = ebp & edx
_push
_getDO        ;get data offset
_getdata      ;ebx = 0xffffffff
_save         ;ecx = 0xffffffff
_shr          ;ebx = 1
_save         ;ecx = 1
_pop
_shl          ;ebx = (edx & ebp) << 1
_nopdB        ;ebp = (edx & ebp) << 1
_save         ;ecx = ebx
_nopsA        ;ebx = eax
_nopdD        ;edx = eax
_push
_xor          ;ebx = edx ^ ebp
_nopdA        ;eax = edx ^ ebp
_pop
_and          ;ebx = edx & ebp
_JnzUp        ;loop while ((edx & ebp) != 0)
_pop          ;discard loop address
_nopsA        ;ebx = eax

;call GetTickCount()

_call
```

Then the choice is made, and the branch might be taken (seven in eight chances to take it):

```
;construct the value '7'

_getDO           ;get data offset
_getdata         ;ebx = 0xffffffff
_save            ;ecx = 0xffffffff
_shr             ;ebx = 1
_save            ;ecx = 1
_shl             ;ebx = 2
_xor             ;ebx = 3
_shl             ;ebx = 6
_xor             ;ebx = 7
_save            ;ecx = 7

;'and' with result from GetTickCount()

_nopsA
_and             ;ebx = ebx & 7
_JnzDown
[conditional command 1]
[conditional command 2]
[conditional command 3]
_nopREAL         ;work around '_JnzDown' bug
```

The replacement code might look something like this, beginning immediately after the call to the GetTickCount() API:

```
    ;save result from GetTickCount()

    _nopsA
    _push

    ;construct pointer to l2

    _getDO        ;get data offset
    _getdata      ;ebx = 0xffffffff
    _save         ;ecx = 0xffffffff
    _shr          ;ebx = 1
    _save         ;ecx = 1
    ... ['_shl' and '_xor' as needed to produce the
    value ((lines(l1...l2) * 8) + 3)]
    _nopdB        ;ebp = offset of l2
    _save         ;ecx = offset of l2
    _getEIP
l1: _nopdD        ;edx = eip
    _xor          ;ebx = edx ^ ebp
    _nopdA        ;eax = edx ^ ebp
    _getEIP
    _push         ;top of do-while loop
                  ;ebx points to a hidden 'pop ebx'
                  ;instruction as part of _getEIP
                  ;so there is no explicit 'pop'
                  ;instruction inside the loop
                  ;that corresponds to this 'push'
                  ;instruction
    _saveJmpOff   ;esi = ebx
    _nopsD        ;ebx = edx
    _save         ;ecx = edx
    _nopsB        ;ebx = ebp
    _and          ;ebx = ebp & edx
    _push
    _getDO        ;get data offset
    _getdata      ;ebx = 0xffffffff
    _save         ;ecx = 0xffffffff
    _shr          ;ebx = 1
    _save         ;ecx = 1
    _pop
    _shl          ;ebx = (edx & ebp) << 1
    _nopdB        ;ebp = (edx & ebp) << 1
    _save         ;ecx = ebx
    _nopsA        ;ebx = eax
```

```
    _nopdD        ;edx = eax
    _push
    _xor          ;ebx = edx ^ ebp
    _nopdA        ;eax = edx ^ ebp
    _pop
    _and          ;ebx = edx & ebp
    _JnzUp        ;loop while ((edx & ebp) != 0)
    _pop          ;discard loop address
    _nopsA        ;ebx = eax
    _saveJmpOff

    ;restore result from GetTickCount()

    _pop
    _nopdA        ;eax = GetTickCount()

    ;construct the value '7'

    _getDO        ;get data offset
    _getdata      ;ebx = 0xffffffff
    _save         ;ecx = 0xffffffff
    _shr          ;ebx = 1
    _save         ;ecx = 1
    _shl          ;ebx = 2
    _xor          ;ebx = 3
    _shl          ;ebx = 6
    _xor          ;ebx = 7
    _save         ;ecx = 7

    ;'and' with result from GetTickCount()

    _nopsA        ;ebx = GetTickCount()
    _and          ;ebx = ebx & 7
    _JnzUp
    [conditional command 1]
    [conditional command 2]
    [conditional command 3]
    ...
    [conditional command n]
l2:     ;branch destination is here
```

27 commands remain.

In the same way as for the '_JnzDown' command, the '_JzDown' command can be removed.

26 commands remain.

Normally, the 'ecx', 'esi' and 'edi' registers are write-only (technically, the 'ecx' register only becomes write-only after the '_addsaved' command is removed), leaving the 'eax', 'ebx', 'edx' and 'ebp' registers as general-purpose registers. However, there is a way to read these registers again after they have been written. The '_pushall' command pushes the registers onto the stack in this order: eax, ecx, edx, ebx, esp, ebp, esi, edi. The registers can then be popped individually from the stack, by using the '_pop' command, in the following way:

```
_pushall  ;save all registers
_pop      ;edi
_pop      ;esi
_pop      ;ebp
_pop      ;esp (useful for reading stack parameters,
          ;using the '_getdata' command, see below)
_pop      ;ebx
_pop      ;edx
_pop      ;ecx
_pop      ;eax
```

A smaller set of '_pop' commands can be used to access particular registers, leaving the others for removal later, if necessary. The popped registers can also be modified and pushed back onto the stack, allowing the '_popall' command to be used to pop all of them. This allows multiple values to be assigned simultaneously. By combining several of these tricks, it becomes possible to remove the '_mul' command (edx:eax = eax * ebx). A working solution can be downloaded from http://pferrie.tripod.com/misc/flibi_mul.zip.

25 commands remain.

Interestingly, by reordering the register initialization code for the first addition block to remove one instruction, the code actually increases in size because the branch to l4 requires more instructions to construct it as a result. This brings us to a special-case problem of dynamic pointer construction. There is a particular problem when the code at l2 branches to l4 and the code at l3 branches to l1, but where l1 < l2 and l4 > l3, as shown here:

```
l1:     [code]
l2:     jz       l4
l3:     jnz      l1
l4:     [code]
```

First, construct the branch from l2 to l4:

```
l1:     [code]

        ;construct relative to l2 (two instructions)

        mov      reg, 1
        shl      reg, 1
l2:     jz       l2+reg
l3:     jnz      l1
l4:
[code]
```

Then construct the branch from l3 to l1:

```
l1:     [code]
        mov      reg, 1
        shl      reg, 1
l2:     jz       l2+reg

        ;construct relative to l3 (four lines)
        ;[code] at l1 is a single instruction to keep
        ;the example simple

l3:     mov      reg, 1
        shl      reg, 1
        shl      reg, 1
        jnz      l3-reg
l4:     [code]
```

Now the branch at l2 is affected, and no longer points to l4, so reconstruct it:

```
l1:     [code]
        ;construct relative to l2 (five instructions)
        mov     reg, 1
        shl     reg, 1
        shl     reg, 1
        add     reg, 1
l2:     jz      l2+reg
l3:     mov     reg, 1
        shl     reg, 1
        shl     reg, 1
        jnz     l3-reg
l4:     [code]
```

But now the branch at l3 is affected, and no longer points to l1, so reconstruct it:

```
l1:     [code]
        mov     reg, 1
        shl     reg, 1
        shl     reg, 1
        add     reg, 1
l2:     jz      l2+reg
        ;construct relative to l3 (six instructions)
l3:     mov     reg, 1
        shl     reg, 1
        add     reg, 1
        shl     reg, 1
        jnz     l3-reg
l4:     [code]
```

Again, the branch at l2 is affected and no longer points to l4, so reconstruct it:

```
l1:     [code]
        ;construct relative to l3 (six instructions)
        mov     reg, 1
        shl     reg, 1
        add     reg, 1
        shl     reg, 1
l2:     jz      l2+reg
l3:     mov     reg, 1
        shl     reg, 1
        add     reg, 1
        shl     reg, 1
        jnz     l3-reg
l4:     [code]
```

Finally, the instructions are reordered but not inserted, and the combination works. The limitation is that the lines in the construction must converge on a multiple of each other. Such a value might not exist without the explicit insertion of 'alignment' lines. The '_nop' command could be used for this purpose, but any 'harmless' instruction can be used, such as moving to/from the same register from/to which a value was just moved (more specifically, if the previous move instruction was from ebx to eax, then it is harmless to move from eax back into ebx). By combining several of these tricks, it becomes possible to remove the '_div' command (eax, edx = edx:eax / ebx) as well. A working solution can be downloaded from http://pferrie.tripod.com/misc/flibi_div.zip.

24 commands remain.

The '_writeDWord' command can be removed by using this algorithm:

```
mov     [edi], bl
inc     edi
shr     ebx, 8
mov     [edi], bl
inc     edi
shr     ebx, 8
mov     [edi], bl
inc     edi
shr     ebx, 8
mov     [edi], bl
```

which we translate into this code:

```
;construct the value '8'
_push
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
_shr            ;ebx = 1
_save           ;ecx = 1
_shl            ;ebx = 2
_shl            ;ebx = 4
_shl            ;ebx = 8
;save in ecx for later
_save           ;ecx = 8
_pop
;write byte 0
_writeByte
;increment edi
_pushall
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
_shr            ;ebx = 1
_nopdB          ;ebp = 1
_save           ;ecx = 1
_pop            ;ebx = edi
_nopdD          ;edx = edi
_xor            ;ebx = edx ^ ebp
_nopdA          ;eax = edx ^ ebp
_getEIP
_push   ;top of do-while loop
        ;ebx points to a hidden 'pop ebx' instruction
        ;as part of _getEIP so there is no explicit
        ;'pop' instruction inside the loop that
        ;corresponds to this 'push' instruction
```

```
_saveJmpOff     ;esi = ebx
_nopsD          ;ebx = edx
_save           ;ecx = edx
_nopsB          ;ebx = ebp
_and            ;ebx = ebp & edx
_push
_getDO          ;get data offset
_getdata        ;ebx = 0xffffffff
_save           ;ecx = 0xffffffff
_shr            ;ebx = 1
_save           ;ecx = 1
_pop
_shl            ;ebx = (edx & ebp) << 1
_nopdB          ;ebp = (edx & ebp) << 1
_save           ;ecx = ebx
_nopsA          ;ebx = eax
_nopdD          ;edx = eax
_push
_xor            ;ebx = edx ^ ebp
_nopdA          ;eax = edx ^ ebp
_pop
_and            ;ebx = edx & ebp
_JnzUp          ;loop while ((edx & ebp) != 0)
_pop            ;discard loop address
_nopsA          ;ebx = eax

;update edi

_push
_popall         ;edi = eax and rebalance stack

;shift ebx right by 8

_shr            ;ebx = ebx >> 8

;write byte 1

_writeByte

[repeat twice more, beginning with 'increment edi'
from above, to write the remaining bytes]
```

Of course, if there were a command to write a new value for the stack pointer, then the stack could be moved to the destination address, and a '_push' command could be used to write the value. However, there would need to be a corresponding command to read the previous value for the stack pointer in order to restore it afterwards. This is quite outside the 'style' of the language.

23 commands remain.

Another instruction that can be removed is the '_call' command. A subroutine call is equivalent to pushing the return address onto the stack, and then jumping to the location of the subroutine. It can be replaced by the '_JnzUp' command in the following way (again, calling the GetTickCount() API, as above):

```
    ;construct pointer to l2

    _getDO      ;get data offset
    _getdata    ;ebx = 0xffffffff
    _save       ;ecx = 0xffffffff
    _shr        ;ebx = 1
    _save       ;ecx = 1
    ... ['_shl' and '_xor' as needed to produce the
    value ((lines(l1...l2) * 8) + 3)]
    _nopdB      ;ebp = offset of l2
    _save       ;ecx = offset of l2
    _getEIP
l1: _nopdD      ;edx = eip
    _xor        ;ebx = edx ^ ebp
    _nopdA      ;eax = edx ^ ebp
    _getEIP
    _push       ;top of do-while loop
        ;ebx points to a hidden 'pop ebx' instruction
        ;as part of _getEIP so there is no explicit
        ;'pop' instruction inside the loop that
        ;corresponds to this 'push' instruction
    _saveJmpOff ;esi = ebx
    _nopsD      ;ebx = edx
    _save       ;ecx = edx
    _nopsB      ;ebx = ebp
    _and        ;ebx = ebp & edx
    _push
    _getDO      ;get data offset
    _getdata    ;ebx = 0xffffffff
    _save       ;ecx = 0xffffffff
    _shr        ;ebx = 1
    _save       ;ecx = 1
    _pop
    _shl        ;ebx = (edx & ebp) << 1
    _nopdB      ;ebp = (edx & ebp) << 1
    _save       ;ecx = ebx
    _nopsA      ;ebx = eax
    _nopdD      ;edx = eax
    _push
    _xor        ;ebx = edx ^ ebp
    _nopdA      ;eax = edx ^ ebp
    _pop
    _and        ;ebx = edx & ebp
    _JnzUp      ;loop while ((edx & ebp) != 0)
    _pop        ;discard loop address
    _nopsA      ;ebx = eax

    ;save return address on stack

    _push

    ;construct pointer to GetTickCount()

    _getDO      ;get data offset
    _getdata    ;ebx = 0xffffffff
    _save       ;ecx = 0xffffffff
    _shr        ;ebx = 1
    _save       ;ecx = 1
    _shl        ;ebx = 2
    _xor        ;ebx = 3
    _shl        ;ebx = 6
    _shl        ;ebx = 0x0c
    _nopdB      ;ebp = 0x0c
    _save       ;ecx = 0x0c
    _getDO      ;get data offset
```

```
    _nopdD       ;edx = data offset
    _xor         ;ebx = edx ^ ebp
    _nopdA       ;eax = edx ^ ebp
    _getEIP
    _push      ;top of do-while loop
               ;ebx points to a hidden 'pop ebx'
               ;instruction as part of _getEIP so
               ;there is no explicit 'pop'
               ;instruction inside the loop that
               ;corresponds to this 'push' instruction
    _saveJmpOff  ;esi = ebx
    _nopsD       ;ebx = edx
    _save        ;ecx = edx
    _nopsB       ;ebx = ebp
    _and         ;ebx = ebp & edx
    _push
    _getDO       ;get data offset
    _getdata     ;ebx = 0xffffffff
    _save        ;ecx = 0xffffffff
    _shr         ;ebx = 1
    _save        ;ecx = 1
    _pop
    _shl         ;ebx = (edx & ebp) << 1
    _nopdB       ;ebp = (edx & ebp) << 1
    _save        ;ecx = ebx
    _nopsA       ;ebx = eax
    _nopdD       ;edx = eax
    _push
    _xor         ;ebx = edx ^ ebp
    _nopdA       ;eax = edx ^ ebp
    _pop
    _and         ;ebx = edx & ebp
    _JnzUp       ;loop while ((edx & ebp) != 0)
    _pop         ;discard loop address
    _nopsA       ;ebx = eax
    _getdata     ;ebx = offset of GetTickCount()
    _saveJmpOff  ;esi = offset of GetTickCount()

    ;clear Z flag

    _save        ;ecx = ebx
    _and         ;ebx = ebx & ebx (known non-zero from
                 ;above)

    ;jump to GetTickCount()

    _JnzUp
 l2:;execution resumes here
```

Local subroutines can be called in the same way; however there is no 'return' command. The equivalent for a 'return' command is the following:

```
 ;retrieve return address from stack

 _pop
 _saveJmpOff     ;esi = return address

 ;clear Z flag, if required

 _save   ;ecx = ebx
 _and    ;ebx = ebx & ebx (known non-zero from above)

 ;return to caller

 _JnzUp
```

22 commands remain.

The following are two useful tricks just for the sake of interest. The first one demonstrates how to read parameters directly from the stack:

```
 [push parameters here, not shown]
 _pushall
 _pop            ;edi
 [_nopdA         ;eax = edi, if needed]
 _pop            ;esi
 [_nopdD         ;edx = esi, if needed]
 _pop            ;ebp (discard)
 _pop            ;esp
 _push           ;esp
 _push           ;ebp
 [_nopsD         ;ebx = original esi, if needed]
 _push
 [_nopsA         ;ebx = original edi, if needed]
 _push
 _popall         ;ebp = esp
 [add to ebp as needed to reach required variable]
 _nopsB          ;ebx = ebp
 _getdata        ;read from stack
```

Then, simply by replacing the '_getdata' command with the '_call' command, function pointers on the stack can be called.

The '_push' command can be removed, but the replacement code is ugly. It would look like this:

```
 _nopdA     ;place into eax in order to appear at the
            ;top of the stack
 _pushall
 _pop            ;discard edi
 _pop            ;discard esi
 _pop            ;discard ebp
 _pop            ;discard esp
 _pop            ;discard ebx
 _pop            ;discard edx
 _pop            ;discard ecx
 ;eax remains as the only register on the stack
```

21 commands remain.

The '_popall' command can be removed. The '_popall' command pops the registers from the stack in the following order: edi, esi, ebp, esp, ebx, edx, ecx, eax. The command can be replaced by popping and assigning the registers individually, in the following way:

```
 _pop            ;edi
 _saveWrtOff
 _pop            ;esi
 _saveJmpOff
 _pop            ;ebp
```

```
_nopdB
_pop            ;esp (discard)
_pop            ;ebx
_pop            ;edx
_nopdD
_pop            ;ecx
_save
_pop            ;eax
_nopdA
```

20 commands remain.

The '_nopdB', '_saveWrtOff' and '_saveJmpOff' commands can be removed if the '_push' and '_popall' commands are retained. Replacement of the '_saveWrtOff' command would look like this:

```
_pushall
_pop            ;discard existing edi
[construct value to place into edi, not shown]
_push
_popall
```

Replacement of the '_saveJmpOff' command would look like this:

```
_pushall
_pop            ;edi
[_nopdD         ;preserve edi if needed]
_pop            ;discard existing esi
[construct value to place into esi, not shown]
_push
[_nopsD         ;restore edi if needed]
_push
_popall
```

Replacement of the '_nopdB' command would look like this:

```
_pushall
_pop            ;edi
[_nopdD         ;preserve edi if needed]
_pop            ;esi
[_nopdA         ;preserve esi if needed]
_pop            ;discard existing ebp
[construct value to place into ebp, not shown]
_push
[_nopsA         ;restore esi if needed]
_push
[_nopsD         ;restore edi if needed]
_push
_popall
```

19 commands remain.

Two other commands can be removed, but they cannot be replaced using existing instructions. Instead, the replacement code requires the introduction of another instruction. The two commands are '_shl' and '_shr'. The replacement instruction is '_rot' ('rotate'). The direction of the rotate is not important, as long as it is known, since all values can be constructed by using it in conjunction with the '_and' instruction. However, it requires the use of the value '1' as the 'base constant'. The value '1' would be used to construct the values '0x7fffffff' (if rotating shifts to the right) or '0xfffffffe' (if rotating shifts to the left). This is the mask value that is used by the '_and' command to zero the appropriate bit in order to simulate a shift. This is the simplest implementation that would rotate a value only once per use without reference to the value in the 'ecx' register. Multi-bit rotates could be supported, too, but then the 'and' mask would no longer be a constant. Instead, it would be specific to the number of bits that are being rotated. So, shifting the value in the 'eax' register left by '3' times, using the single-bit rotate command, would look like this:

```
;construct the value '0xfffffffe'

_getDO          ;get data offset
_getdata        ;ebx = 1
_save           ;ecx = 1
_rot            ;ebx = 2
_xor            ;ebx = 3
_rot            ;ebx = 6
_xor            ;ebx = 7
_rot            ;ebx = 0x0e
_xor            ;ebx = 0x0f
[repeat seven more times, but omit the final xor]
_save           ;ecx = 0xfffffffe

;rotate left and zero the overflow bits

_nopsA          ;ebx = eax
_rot            ;ebx = rol(ebx, 1)
_and            ;ebx = ebx & 0xfffffffe
_rot            ;ebx = rol(ebx, 1)
_and            ;ebx = ebx & 0xfffffffe
_rot            ;ebx = rol(ebx, 1)
_and            ;ebx = ebx & 0xfffffffe
_nopdA          ;eax = shl(eax, 3)
```

18 commands remain.

- _nopsA, _nopsB, _nopsD, _nopdA, _nopdD
- _writeByte
- _save
- _getDO, _getdata, _getEIP
- _push _pop, _pushall, _popall
- _rot, _and, _xor
- _JnzUp

Many years from now, our distant descendants might stumble upon a codon stream that describes only 18 amino acids – and we might be looking at its origin. Imagine that.