CodeBreakers Journal
Security & Anti-Security - Attack & Defense

Virtual Machine Rebuilding by Maximus

Vol. 1, No. 1, 2006

Abstract:
Virtual Machines are one of the most controversial protection methods used nowadays. I try to show how virtual machines are created by examining the full reversed source code of the VM used in the T2'06 challenge, worth $1'500. It explains how to code a VM and helps those who wanted to analyse such challenge by giving direct access to its source code and its structures. The RE-built source code of T2'06 is in appendix.

Virtual Machine Rebuilding

By Maximus

Virtual Machines are one of the most controversial protection methods used nowadays. I try to show how virtual machines are created by examining the full reversed source code of the VM used in the T2'06 challenge, worth $1'500. It explains how to code a VM and helps those who wanted to analyse such challenge by giving direct access to its source code and its structures. The RE-built source code of T2'06 is in appendix.

1 Introduction

Reversing a Virtual machine do rarely requires a total understanding of the VM structure. Usually, we use the disassembler to perform a quick analysis of the machine structure when possible, then we debug to see the 'live' data moving, how they fit our initial description, what they really 'do'. Being the T2'06 a small application, I just reversed it in full, offering you its rebuilt source code. There should be no reason for doing this to solve it. My main reason for it is related to my prior tutorial on Virtual Machines. I got many requests of more complex and more simpler things, as there were good Reverser's which still have problems approaching this new technology without theoretical support. So, this essay goes directly to the point, offering *both* a reversing essay on Virtual Machines and a deeper look on VM structure, their (RE)coding and (RE)creation.

For a basic approach to a simple VM (acronym of Virtual Machine), I strongly suggest you to read my prior basic tutorial on the subject. You can find it on Code-Breakers Magazine (CBM). Please note that I suppose you to have a basic IDA/WDASM knowledge, because in the article, as I'll directly refer to the target application's code, either to its reversed C shape or its assembly. Many of the 'sanity checks' performed in original code are omitted in the reversed C code, as they are not useful to code comprehension and so are omitted for brevity. Also, note that I did not debug the T2'06 challenge: I used only IDA 4.3 – the one without the debugger, and did not bother to run Olly in parallel for it.[1]

Said this, let me raise the volume of my MP3 player and start.

2 General Approach, Structures

Few seconds after I opened the T2'06 I recognized I were facing a VM: how, you may ask. If you look closely to the _main() function, you may notice a generic loop with a sort of dispatcher which calls a a function chosen within a list of functions. And this sounds terribly like a basic VM core.

A Virtual Machine is usually built around a main loop that decodes and executes the VM instructions. The generic picture of it might be drawn this way:

```
Virtual Machine Loop Start
        ...
        ...
        ...
        --> Decode VM Instruction's
        ...
        --> Execute the Decoded VM Instruction's
        ...
        ...
        --> Check Special Conditions
        ...
        ...
        ...
Virtual Machine Loop End
```

The T2'06 Machine is built around a VM Context structure, which holds the machine registers as well as a set of references and pointers to the VM data. Another important structure of this VM is the transport layer used to interface the VM Decoder to the VM Instructions. This layer is just a structure where the decoder places the decoded data of the instruction flow, so that the functions which implements them can easily access such information's.

Let's give a quick look to the VM Context structure, the machine 'core':

```
struct TVMContext {
       int Register_IOType,                // generic register,
can held i/o direction
       int Register_IO,                    //
       int Register_IOAddress,             // generic register,
can address memory referenced
       int Register_IOCount,               // generic register,
can bytes transferred
       int GenericRegisters[12],           // other general
registers
       int *Registers,                     // pointer to
registers area
       int VM_ESP ,                        // virtual stack
pointer
       int VM_EIP ,                        // virtual
Instruction's pointer
       int VMEIP_Saved_Prior_InstrExec,    // keep track of
currently executed instruction's IP
       TVMFLAGS VM_EFLAGS,                 // flags of the VM
       int InstructionCounter,             // number of
instructions
       int InitCode,
       int MemorySize,
       void* ProgramMemoryAddr,
       int Original_ESP,
       int StackMemorySize,
       void* StackMemoryAddr,
       int MachineControl,                 // keep track of the
VM Machine Status
       int VM_ResumeExec                   // a sort of machine's
exception handler
}
```

The VM Context contains a bunch of general and specific registers, the reference to the virtual IP and SP, the flags

1 To be honest, I fired Olly the very fist time I found the challenge, for checking if it were packed or not.

and the reference to the virtual memory areas of this VM (this VM uses private, virtual memory spaces with its own addressing mode).

On the other hand, the VM Decoder fills the following structure during the decoding stage:

```
struct TInstructionBuffer{
       int Length,                // Length of the processed
instruction
       int InstructionData,       // passed by lookup table
       char InstrType,
       //char Fillers1[3],        // if structure alignment is 1
       int Operand_Size,
       char InstructionParamsCount,
       //char Fillers2[3],        // if structure alignment is 1
       SubInstr ParamDest,
       SubInstr paramSrc,
       SubInstr ParamThird,
       SubInstr *WorkSubField     // pointer to the SubInstr
parameters (like “Registers[]” do)
};
```

It contains mainly the parameters of the decoded instructions, as well as few specific data the decoder 'fills' up. These two structures are passed to the VM instructions, which operates on them to make the Virtual machine 'acts'.

For example, let's imagine how a NOP instruction might be implemented. We know the effect of a NOP instruction: it simply skips to the next instruction in the code flow -in other words, it just updates EIP to the next instruction.

On this VM, a NOP instruction could simply be implemented as:

```
TVMContext.VM_EIP = TVMContext.VM_EIP +  TInstructionBuffer.Length
```

The IP register of the VM gets incremented by the length of the processed instruction, so that it will point to the next one.

A VM usually takes advantage of an Instruction's table to dispatch the instruction execution. It is the point that -scrolling the code- leaded me to find the Opcode table (instead of just scrolling the data area for a big list of pointers). It is often, but not always, implemented with an indexed jump/call:

```
                    Opcode = *RealEIP;
MachineCheck = (*OpcodeProc[(char)Opcode])(&InstBuff,&VMContext);
// VM dispatcher
```

As we can also see in the following IDA snip:

```
Execute_VM_Opcode:                      ; CODE XREF: _main+13D#j
.text:004021B8 0D4                mov     edx,
[esp+0D4h+VMInstructionBuff_VM_Opcode]
.text:004021BF 0D4                lea     eax,
[esp+0D4h+VM_Context] ; Load Effective Address
.text:004021C3 0D4                lea     ecx,
[esp+0D4h+VM_InstructionBuff_Body_Ptr] ; Load Effective Address
.text:004021CA 0D4                and     edx, 0FFh       ; VM
Opcode is 1 byte only
.text:004021D0 0D4                push    eax             ; VM
Context
.text:004021D1 0D8                push    ecx             ; VM
Instr Ptr
.text:004021D2 0DC                call    VM_Opcode_Table[edx*4]
; Indirect Call Near Procedure
.text:004021D2
.text:004021D9 0DC                add     esp, 8          ; Add
.text:004021DC 0D4                test    eax, eax        ;
Logical Compare
.text:004021DE 0D4                jz      VM_Loop_Head_Default ;
Jump if Zero (ZF=1)
```

You may wonder how I decoded the “RealEIP” name in the prior code. You can understand this by examining the value used to jump to the VM Opcode Table in the IDA snip. The index is given by a byte that is being taken from an address. Obviously, this byte discriminates which instruction *is* executed , and so it can only be the Instruction Opcode itself. Also, it is shaped to be a direct index for the Dispatcher within the VM Instruction Table.

Now, we just need to tag our VM Instruction Table and move to the virtual Opcodes, trying to locate the Context object of the VM, some of the most common VM registers like the Instruction Pointer, some instruction and so on, until we start getting a clear picture. Below you can see a snippet of this structure, taken from IDA source:

```
.data:00407430     VM_Opcode_Table dd offset VM_MOV        ; DATA
XREF: _main+162#r
.data:00407434                     dd offset VM_Multiple_op2
.data:00407438                     dd offset VM_Multiple_op2
.data:0040743C                     dd offset VM_Multiple_op2
.data:00407440                     dd offset VM_Multiple_op2
.data:00407444                     dd offset VM_Multiple_op2
.data:00407448                     dd offset VM_PUSH
.data:0040744C                     dd offset VM_POP
.data:00407450                     dd offset VM_JMP
.data:00407454                     dd offset VM_CALL
...                                ...
```

The Instruction's Table of this VM contains a long list of functions. One thing that stoke me at first glance was the fact that some of them were repeating. This can mean that some Opcode is missing, or that multiple VM instructions are encoded within the same procedure. So, I just tagged the repeated ones assigning them a couple of temporary names. Then, I started to look for the most simple VM instructions referenced by the table (I considered 'simple' an instruction which made little use of internal calls).

By scanning the VM Instruction Table, you can quickly reach the following implementation:

```
; int __cdecl VM_NOP(int Param1,int Param2)
.text:00401F80     VM_NOP          proc near               ; CODE
XREF: _main+162#p
.text:00401F80                                             ; DATA
XREF: .data:0040746C#o ...
.text:00401F80
.text:00401F80     Param1 = dword ptr  4
.text:00401F80     Param2 = dword ptr  8
.text:00401F80
.text:00401F80 000                mov     ecx, [esp+Param1]
```

```
.text:00401F84 000                mov     eax, [esp+Param2]
.text:00401F88 000                mov     edx, [ecx]
.text:00401F8A 000                mov     ecx,
[eax+Param2.Something]
.text:00401F8D 000                add     ecx, edx        ; Add
.text:00401F8F 000                mov
[eax+Param2.Something], ecx
.text:00401F92 000                xor     eax, eax        ;
Logical Exclusive OR
.text:00401F94 000                retn                    ;
Return Near from Procedure
.text:00401F94
.text:00401F94     VM_NOP         endp
```

It wasn't very difficult to understand it was our NOP instruction. I had chosen it at first sight because it was a very 'favourable' instruction: it does not call internal functions, and it is short. As you can notice it:

1. Take two parameters: most probably one is the VM Context.
2. Take the first value pointed by Param1 (mov edx,[ecx]).
3. Take a value at a certain offset from Param2 (mov ecx, [eax+Param2.Something]).
4. Add the value obtained from Param1 to the value read from Param2
5. Save the sum of the prior point within, and terminates.

This code leads to some considerations: we know that the Virtual IP must be incremented somewhere -this step could be accomplished in several ways. And the sequence is in plain C:

```
Struct2.FieldX = Struct2.FieldX + Struct1.Field0
```

Looks really like

```
VMContext.VM_EIP = VMContext.VM_EIP + TInstructionBuffer.Length
```

A good way for solving this issue is to examine other instructions and see if this is a recurrent pattern. Just roaming here and there among the VM Instruction Table will tell you that this code fragment is very used, usually at the end of instructions. So, you can bet it is our VM EIP increment. This also can give us an idea of which structure might contain the general VM registers.

If you are smarter than me, you might have noticed that the structure which contains VM_EIP is allocated on the caller's stack space, which means the VM structure is held as a local variable of the _main() function. Having not noticed this at first, I had to rename all the IDA fields of the _main() module to match recovered VM structure names. It made the _main() function much more understandable, by the way.

Decoding our VM_NOP instruction helped me to recover the following elements:

1. The VM Context: because we know the parameter that holds it, so we can backtrack it in the main and we can label properly the parameter of the other instructions.
2. The Virtual IP (VMContext.VM_EIP): because it gets incremented in the NOP instruction.
3. The area where the decoded instruction is put. If you examine other VM instructions, you can notice that the 2nd parameter is referenced as a structure. And since the VM Opcode Dispatcher makes no differentiation among instructions and passes to them all the very same parameters, we can bet (but it is *not* sure!) it is a structure.

However, this is not even the tip of the iceberg. Complexities had yet to come, especially if the Zen does not help in time (as it happened to me, requiring a bunch of more hours of reversing time). We could move in various directions now. For example, we could examine what procedures are called just before the Opcode Dispatcher: the Instruction's decoder must be there, as they are not decoded 'on the fly' within each instruction. I have chosen to examine deeply the instruction set, and this is the knowledge of the machine I got in the process. I hope it can help you not only to understand the T2'06 VM, but can give you an idea of a way it can be analysed (I think it required about 6 working days for general reversing, 4-6 for full C rewriting and article, heh!).

During the analysis of this VM, I tried to locate the Jump(s) instruction: it can be useful to gain an idea of the Virtual Flags structure. The VM_JCC is located at .text:00401C80. It can be easily recognised as a JCC because it perform a lot of conditional testing and, moreover, it uses our VM_EIP by incrementing/altering it in response to the test's results. The VM Jcc instruction is a 'multivalent' one, as it spans more than one Opcode in the VM Instruction table.

Scrolling the code, we can see many instructions that calls internal functions to do some unknown work. It is a good idea to leave them off when analysis is still at start, concentrating on more easier functions. This does not mean, however, that we should renounce to look for interesting pattern on those functions!

For example, we can find a complex instruction that performs many math operations, differentiating them on byte/word/dword basis. It is the operand I initially called VM_MultipleOp2. One of the most interesting points of this instruction is that it features this code:

```
case 2: // XOR
        switch(DecodedInstr->OperandSize) {
                        case 1:      VMValueEval = (char)VMValueSrc ^
(char)VMValueThird; break;
                        case 2:      VMValueEval = (word)VMValueSrc ^
(word)VMValueThird;break;
```

```
                    case 4:      VMValueEval = VMValueSrc ^
VMValueThird;
        }
```

As you can notice, it checks pattern of values (1-2-4) and act accordingly to the byte-size value, performing the same relevant operations on different data-size. When you see such pattern, you can easily recognize their usage and so you can 'spot' instructions usage and parameters. Clearly, the name of the parameters in the above code is given after that a good knowledge of the machine has been gathered, but you can notice even in the initial steps of reversing that such parameter (DecodedInstr->OperandSize) is used to differentiate the operand size, and you can see that the code structure is used to perform an XOR operation on multiple operand size. In IDA, it looks that way:

```
 VM_XOR_case_multi_3:                    ; CODE XREF:
VM_Multiple_op2+70#j
.text:00402336                                          ; DATA
XREF: .text:004026E8#o
.text:00402336 014                mov     eax,
[ebx_is_InstrBuf+VM_InstrBuffer.Operand_Size]
.text:00402339 014                dec     eax           ;
Decrement by 1
.text:0040233A 014                jz      short loc_40236E ;
Jump if Zero (ZF=1)
.text:0040233A
.text:0040233C 014                dec     eax           ;
Decrement by 1
.text:0040233D 014                jz      short loc_402355 ;
Jump if Zero (ZF=1)
.text:0040233D
.text:0040233F 014                sub     eax, 2        ;
Integer Subtraction
.text:00402342 014                jnz
Finalize_Instruction_end_of_case_0Ch ; Jump if Not Zero (ZF=0)
.text:00402342
.text:00402348 014                mov     eax,
[esp+14h+Hold_34h_param]
.text:0040234C 014                mov     esi, edi_is_Param_24h
.text:0040234E 014                xor     esi, eax      ;
Logical Exclusive OR
.text:00402350 014                jmp
Finalize_Instruction_end_of_case_0Ch ; Jump
.text:00402350
.text:00402355     ;
-------------------------------------------------------------------
----------
.text:00402355
.text:00402355     loc_402355:                          ; CODE
XREF: VM_Multiple_op2+12D#j
.text:00402355 014                mov     esi,
[esp+14h+Hold_34h_param]
.text:00402359 014                mov     ecx, edi_is_Param_24h
.text:0040235B 014                and     esi, 0FFFFh   ;
Logical AND
.text:00402361 014                and     ecx, 0FFFFh   ;
Logical AND
.text:00402367 014                xor     esi, ecx      ;
Logical Exclusive OR
.text:00402369 014                jmp
Finalize_Instruction_end_of_case_0Ch ; Jump
.text:00402369
.text:0040236E     ;
-------------------------------------------------------------------
----------
.text:0040236E
.text:0040236E     loc_40236E:                          ; CODE
XREF: VM_Multiple_op2+12A#j
.text:0040236E 014                mov     esi,
[esp+14h+Hold_34h_param]
.text:00402372 014                mov     edx, edi_is_Param_24h
.text:00402374 014                and     esi, 0FFh     ;
Logical AND
.text:0040237A 014                and     edx, 0FFh     ;
Logical AND
.text:00402380 014                xor     esi, edx      ;
Logical Exclusive OR
.text:00402382 014                jmp
Finalize_Instruction_end_of_case_0Ch ; Jump
```

However, this discovery is not enough to get a deeper knowledge of the VM. I have used IDA 4.3, so I cannot debug and this means I cannot understand what the parts of code that comes before and after this snippet do, especially those who call some functions. Not a great problem -I could use Olly which is a *far* better debugger, but I didn't want to. This made reversing a bit more complex, but surely more interesting. So, let us recap what we discovered about this VM by examining the code:

1. We know where the VM Context structure is, and we know at least a field of it: VM_EIP.
2. We know where the decode VM Instruction is put, and we know where the instruction's byte length is placed -in the first field of this structure.
3. We discovered the “OperandSize” sub-field of the VM Decoder structure, by locating a code that seems to perform XOR on different data size.

Other interesting instructions that we might examine at the start of analysis are the VM_DEC and the VM_INC ones. They are pretty recognizable by the use of ...inc(x) and dec(x), of course. Also, the VM_NOT instruction can be found this way.

But, before or later, we must start picking the real VM rock. So, it is wise to get back and examine the main VM loop, trying to locate and understand its procedures. If you look the _main() procedure, you can notice it elaborates a call just prior the execution of the VM opcode... you can bet it is the VM Instruction Decoder.

```
.text:00402196 0D4                mov     eax,
[esp+0D4h+RealVMAddr__and_decoded_VMEIP]
.text:0040219A 0D4                lea     ecx,
[esp+0D4h+VM_InstructionBuff_Body_Ptr]
.text:004021A1 0D4                push    eax
.text:004021A2 0D8                push    ecx
.text:004021A3 0DC                call    VMInstructionDecoder ;
Call Procedure
.text:004021A3
.text:004021A8 0DC                add     esp, 8          ; Add
.text:004021AB 0D4                test    eax, eax        ;
Logical Compare
.text:004021AD 0D4                jnz     short
Execute_VM_Opcode ; Jump if Not Zero (ZF=0)


if (!MachineCheck) {
                MachineCheck =
VMInstructionDecoder(&InstBuff,RealEIP);
                if (!MachineCheck) // check for opposite
behavior...
```

Following our analysis, we back-trace the _main() function up to the small function called before our decoder. Let's look to the general sequence of actions in the main: one of these functions take our VM_EIP address and test it against 2 blocks bounds. Then, it deference it to a memory address.

I made my bet: one of the block represents the VM Memory space, the other the VM Stack space. The address is a *generic parameter* of the function, which

takes a virtual address and test it against VM Memory space for its correct interpretation. I were right, of course.

So, even if in the actual call from the main we see the parameter as receiving the Virtual EIP, the function accepts any generic VM pointer and dereferences it to the 'real' x86 pointer. This way, we uncovered the function used to dereference memory:

```
bool VMAddress2Real(TVMContext *VMContext,int VMAddress,int
*RealAddr) { // .text:004011D0
        if( RANGE(VMAddress,VMContext->InitCode,VMContext-
>MemorySize) ) {
                *RealAddr = (VM_Address-VMContext-
>InitCode)+VMContext->ProgramMemoryAddr;
                return 1;
        }
        if( RANGE(VMAddress,VMContext->Original_ESP,VMContext-
>StackMemorySize) ) {
                *RealAddr = (VM_Address-VMContext-
>Original_ESP)+VMContext->StackMemoryAddr;
                return 1;
        }
        VMContext->MachineControl = mcWrongAddress;
        return 0;
}
```

As you can notice by examining the code, we have a 'MachineControl' register uncovered, and one of its status. Such register get set in various places, where an error condition appears (this means that I cross-referenced this before making the supposition!), so it got natural to me pairing it with a MachineControl register. It should be noted that it is not only an 'error status' register, as it gets used for indicating other machine conditions different from errors : for example, input/output of VM with the outside world is signalled using a MachineControl value, as we can see from implementation:

```
// :00401F60     VM_ALLOW_IO
int __cdecl VM_ALLOW_IO(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {

        VMContext->MachineControl = mcInputOutput;
        NextInstr(VMContext,DecodedInstr);
        // very ugly: for having IO you must force a
MachineControl check, as if error were in.
        return 1;
}
```

The above C code is the parallel of:

```
:00401F60     VM_ALLOW_IO     proc near               ; CODE
XREF: _main+162#p
.text:00401F60                                        ; DATA
XREF: .data:0040752C#o
.text:00401F60
.text:00401F60     arg_0           = dword ptr  4
.text:00401F60     arg_4           = dword ptr  8
.text:00401F60
.text:00401F60 000                 mov     eax, [esp+arg_4]
.text:00401F64 000                 mov     ecx, [esp+arg_0]
.text:00401F68 000                 mov
[eax+VM_Context.maybe_MachineControl], mcInputOutput
.text:00401F6F 000                 mov     edx, [ecx]
.text:00401F71 000                 mov     ecx,
[eax+VM_Context.VM_EIP]
.text:00401F74 000                 add     ecx, edx        ; Add
.text:00401F76 000                 mov
[eax+VM_Context.VM_EIP], ecx
.text:00401F79 000                 mov     eax, 1
```

```
.text:00401F7E 000                 retn                    ;
Return Near from Procedure
.text:00401F7E
.text:00401F7E     VM_ALLOW_IO     endp
```

A personal comment: I don't like the solution used by the this VM creator. I think it is an inelegant solution, because it uses the same check both for errors and for non-errors, and requires the instruction to return an 'error code' (the '1' value) for processing correctly the different machine status. The following snippet shows you how this machine performs the Machine's control check:

```
.text:004021E4 0D4                 cmp
[esp+0D4h+Ctx_var_54_zeroed_on_loop_head_R70_MachineControl],
edi_mcInputOutput ; Compare Two Operands
.text:004021EB 0D4                 jnz
VM_MachineErrCheck_OrEndOfVM ; jump to test if we need NOT to
read/write output!
.text:004021EB
.text:004021F1 0D4                 lea     edx,
[esp+0D4h+VM_Context] ; Load Effective Address
.text:004021F5 0D4                 push    edx             ;
VM_Context_Ptr
.text:004021F6 0D8                 call    CheckForInputOutput ;
Call Procedure


if (MachineCheck && VMContext.maybe_MachineControl==c2) { // VM
loop end. c2==mcInputOutput
         CheckForInputOutput(&VMContext);
         continue;
```

The original application does not seem to use the “MachineCheck” as I did. The code in the _main() made me think a lot about the original code. I hope this is the result of #defining out code, and that no *goto* semantic is present there, as it really seems to me. That would represents a bad design (and coding) choice.

Continuing with the analysis of this Virtual Machine, we can try to locate the instructions that appears to reference a stack structure, hoping that our initial bet on the stack presence is fulfilled: this way, instruction that alters VM_ESP and that manages the stack comes out. We might not be able to understand in detail what the VM instructions that makes use of the stack do, as they calls internal procedures that have to be understood, but we can at least understand on what they act, and label them appropriately. Locating the CALL/RET pair isn't very easy until we start examining the internal functions, as you can understand by examining the VM_RET instruction below:

```
int __cdecl VM_RET(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue;

        Read_VMMemory_To(VMContext->VM_ESP, &VMValue);
        VMContext->VM_ESP-=4;
        VMContext->VM_EIP = VMValue;
}
```

```
.text:00401EC0     VM_RET          proc near               ; CODE
XREF: _main+162#p
.text:00401EC0                                             ; DATA
```

```
XREF: .data:0040745C#o
.text:00401EC0
.text:00401EC0     VMContext_Ptr   = dword ptr  0Ch
.text:00401EC0
.text:00401EC0 000                 push    esi
.text:00401EC1 004                 mov     esi,
[esp+VMContext_Ptr]
.text:00401EC5 004                 push    esi             ;
vm_context_ptr
.text:00401EC6 008                 lea     eax,
[esp+4+VMContext_Ptr] ; Load Effective Address
.text:00401ECA 008                 mov     ecx,
[esi+VM_Context.VM_ESP]
.text:00401ECD 008                 push    4               ;
AddressDataSize
.text:00401ECF 00C                 push    eax             ;
Write_VMValue_in_LE_At
.text:00401ED0 010                 push    ecx             ;
VMAddress
.text:00401ED1 014                 call    Read_VMMemory_To ; was
Set_RealAddress_To
.text:00401ED1
.text:00401ED6 014                 add     esp, 10h        ; Add
.text:00401ED9 004                 test    eax, eax        ;
Logical Compare
.text:00401EDB 004                 jnz     short loc_401EE4 ;
Jump if Not Zero (ZF=0)
.text:00401EDB
.text:00401EDD 004                 mov     eax, 1
.text:00401EE2 004                 pop     esi
.text:00401EE3 000                 retn                    ;
Return Near from Procedure
.text:00401EE3
.text:00401EE4     ;
-------------------------------------------------------------------
----------
.text:00401EE4
.text:00401EE4     loc_401EE4:                             ; CODE
XREF: VM_RET+1B#j
.text:00401EE4 004                 mov     eax,
[esi+VM_Context.VM_ESP]
.text:00401EE7 004                 mov     edx,
[esp+VMContext_Ptr]
.text:00401EEB 004                 add     eax, -4         ; Add
.text:00401EEE 004                 mov
[esi+VM_Context.VM_EIP], edx
.text:00401EF1 004                 mov
[esi+VM_Context.VM_ESP], eax
.text:00401EF4 004                 xor     eax, eax        ;
Logical Exclusive OR
.text:00401EF6 004                 pop     esi
.text:00401EF7 000                 retn                    ;
Return Near from Procedure
.text:00401EF7
.text:00401EF7     VM_RET          endp
```

The recovery of the value from the virtual stack is performed by a specific procedure, and it deals with the Virtual Memory allocated by this VM.

Without figuring this out, reversing this kinds of instructions might be a bit problematic. Nonetheless, if you look carefully, you can notice that such called function returns a value that is placed in EIP. And since the function seems to operate on our possible VM_ESP, you can start to catch the picture even without knowledge of the internal function actions (also, we might notice that the supposed stack uses a 'different' grow-up direction!).

I followed another way: I attacked the internal functions directly, to see what they really do. During the analysis, when the whole picture was quite missing, I had a problem understanding the reason of the DEC EAX in the instruction that starts at text:00401E10 -I just dropped it after few minutes. You might wish to have a look to it and try to guess what such instruction does – it might be a nice training session.

I wish to end this generic introduction to this VM by highlighting two passages that occurred in my analysis.

The first one is a thing that fuzzed me at the very start of the analysis, and were the initial values assigned to the stack and application's area. For example, the stack start is 0x80000000, which sounded really an *odd* value to me. My Zen did not help here, so I were forced to look for the answer other ways.

Reversing here and there, I came to the following code snippet:

```
.text:00401261 004                 jnz     short loc_4012AD ;
here below, operandsize is 4
.text:00401261
.text:00401263 004                 mov     ecx, ebx_param_vmvalue
.text:00401265 004                 mov     edx, ebx_param_vmvalue
.text:00401267 004                 mov     eax, ebx_param_vmvalue
;
.text:00401267                                             ; this
code simply swap ebx bytes
.text:00401267                                             ; from
4321 Little endian to 1234 big endian,
.text:00401267                                             ; and
write VMAddress2RealAddr the BE value
.text:00401269 004                 and     ecx, 0FF0000h   ; take
3rd byte
.text:0040126F 004                 shr     edx, 10h        ;
Shift Logical Right
.text:00401272 004                 and     eax, 0FF00h     ; take
2nd byte
.text:00401277 004                 or      ecx, edx        ;
Logical Inclusive OR
.text:00401279 004                 mov     edx,
[esp+4+Ptr_ValueToWriteAndSwap]
.text:0040127D 004                 shl     ebx_param_vmvalue, 10h
; Shift Logical Left
.text:00401280 004                 or      eax, ebx_param_vmvalue
; Logical Inclusive OR
.text:00401282 004                 pop     ebx_param_vmvalue
.text:00401283 000                 shr     ecx, 8          ;
Shift Logical Right
.text:00401286 000                 shl     eax, 8          ;
Shift Logical Left
.text:00401289 000                 or      ecx, eax        ;
Logical Inclusive OR
.text:0040128B 000                 mov     eax, 1
.text:00401290 000                 mov     [edx], ecx
.text:00401292 000                 retn                    ;
Return Near from Procedure
```

If you pay attention, this code swaps the bytes of a double word from Little Endian to Big Endian (and vice versa)[2]. After I found (and understood) this code, it immediately gave me a different picture of the 0x80000000 value. It is written in Big Endian, not in Little Endian! I was very unhappy that I did not understand it at sight -oh well, nobody's perfect, after all[3].

The last point I wish to underline is the addressing modes of this Virtual Machine. As common processors, this VM uses different addressing modes -i.e. direct addresses, registry addresses etc.

A good approach to understand the way values are used within instructions is to use the Virtual Machine's Jump instruction as a reference: there you can surely find addressing modes in action: if you examine the VM_JMP

2 The Endianness is the order of bytes (and bits) in a word's machine. X86 are Little Endian processors, and keep the least significant byte as last (the same happens for Bit order). Big Endian machines keeps higher bytes last, but usually keeps a Little Endianness at bit's level anyway.

3 This is doubly true as -my mistake- I erroneously inverted in my mind the Endianness of bits in article's beta.

instruction, you can notice how the Virtual EIP is changed, and decode at least part of the addressing modes the VM uses. For example, by examining this VM Jump instruction, we can notice that a parameter is being tested and, if the test is 'ok', it is *added* to the current Virtual EIP. Does not it sound like a jump by displacement? See it with your eyes:

```
.text:00401D79     loc_401D79:                             ; CODE
XREF: VM_JMP+1F#j
.text:00401D79 008                 cmp
[esi+SubInstr.AddressType], vmaVMValue_orC4__or_displacement ;
Compare Two Operands
.text:00401D7C 008                 jnz     short make_jmp  ; Jump
if Not Zero (ZF=0)
.text:00401D7C
.text:00401D7E 008                 mov     edx,
[esp+8+InstrBuf_Then_Addr_WriteTo] ; relative jump!
.text:00401D82 008                 mov     eax,
[edi+VM_Context.VM_EIP]
.text:00401D85 008                 add     eax, edx        ; Add
.text:00401D87 008                 mov
[edi+VM_Context.VM_EIP], eax
.text:00401D8A 008                 pop     edi
.text:00401D8B 004                 xor     eax, eax        ;
Logical Exclusive OR
.text:00401D8D 004                 pop     esi
.text:00401D8E 000                 retn                    ;
Return Near from Procedure
.text:00401D8E
.text:00401D8F     ;
---------------------------------------------------------------------
----------
.text:00401D8F
.text:00401D8F     make_jmp:                               ; CODE
XREF: VM_JMP+2C#j
.text:00401D8F 008                 mov     eax,
[esp+8+InstrBuf_Then_Addr_WriteTo]
.text:00401D93 008                 mov
[edi+VM_Context.VM_EIP], eax
.text:00401D96 008                 pop     edi
.text:00401D97 004                 xor     eax, eax        ;
Logical Exclusive OR
.text:00401D99 004                 pop     esi
.text:00401D9A 000                 retn                    ;
Return Near from Procedure
.text:00401D9A
.text:00401D9A     VM_JMP          endp
```

So, performing a final, quick recap of the way this VM machine was attacked:

1. We have located the loop's body, and found the dispatcher. It gave us the Opcode Table.
2. We examined randomly the VM Opcode functions looking for easy functions and recurrent code patterns.
3. We located the NOP, discovered the VMContext structure, the InstructionDecoder structure, some of the fields of the Context and the decoder.
4. We examined an instruction that operated on different data size, decoding the operand_size field.
5. We discovered possible stack usages.
6. We discovered the addressing way of the virtual memory
7. We discovered the opcodes addressing modes.

Surely much more has yet to be done for a full analysis of this VM, but these steps are enough to give a decent knowledge for reversing. Next article's sections will describe with more details various parts of this Virtual Machine, mainly using its reversed C code. Usually, address references of such functions are given, so it would be useful to keep your debugger/disassembler opened to see how the assembly code really looks like.

3 The Virtual Machine Body

A Virtual Machine is usually built around a Context, which is the space where the machine registers and parameters are allocated[4]. The T2'06 does not make exception: this is the memory space used by this Virtual Machine:

```
struct TVMContext {
        int Register_IOType,
        int Register_IO,
        int Register_IOAddress,
        int Register_IOCount,
        int GenericRegisters[12],
        int *Registers,
        int VM_ESP ,
        int VM_EIP ,
        int VMEIP_Saved_Prior_InstrExec,
        TVMFLAGS VM_EFLAGS,
        int InstructionCounter,
        int InitCode,
        int MemorySize,
        void* ProgramMemoryAddr,
        int Original_ESP,
        int StackMemorySize,
        void* StackMemoryAddr,
        int MachineControl,
        int VM_ResumeExec
}

struct TVMFLAGS {
         // ~Compiler Dependent~ -please check the order!!
         ZF:1, // compiler-supposed Bit 0
         CF:1,// compiler-supposed Bit 1
         OF:1,// compiler-supposed Bit 2
         SF:1,// compiler-supposed Bit 3
         Unused:3,// compiler-supposed Bit 4-6
         TF:1,// compiler-supposed Bit 7
}
```

Some of these fields are specific of this VM, however we can 'see' the generic fields: a set of specific and general purpose registers, the execution and stack pointer (yes, our EIP and ESP), the machine's flags, and other flags -the address of memory space and stack space among the others. An interesting field is the last one, the ResumeExec. This field is used as a sort of 'Exception Handler' and it is used also for Debugging purposes (almost all debugging code was removed by T206, but you can recover it by the things that were left i.e. the 'obvious' Trap Flag check).

The initial registers are named, as they are used for IO purposes. It is not their only usage -it is their *special* usage. They got addressed and used this way once IO is allowed by the VM_ALLOW_IO instruction (already shown).

```
int __cdecl CheckForIO(VM_Context) { // .text:00402040
```

4 Different Virtual Machines clearly use different allocations schemes.

```
        switch(VMContext.Register_IOType) {
        case 2: return Do_Write_Output(VM_Context);
        break;
        case 3: return Do_Read_Input(VM_Context);
        break;
        default: return 1;
        }

}

int __cdecl Do_Write_Output(TVMContext* VMContext) { //
.text:00401FF0

        int NumberBytesToWriteOut;
        void *BufferToWrite;

        if (VMContext->Register_IO!=0) return 0;

        VMAddress2Real(VMContext,VMContext-
>Register_IOAddress,&BufferToWrite);
        NumberBytesToWriteOut = VM_Context->Register_IOCount; //
        VM_Context->Register_IOCount =
write(stdout,BufferToWrite,NumberBytesToWriteOut);
        return 1;
}

int __cdecl Do_Read_Input(TVMContext* VMContext)  { //
.text:00401FA0

        int NumberBytesToReadIn;
        void *BufferToRead;

        if (VMContext->Register_IO!=0) return 0;

        VMAddress2Real(VMContext,VMContext-
>Register_IOAddress,&BufferToRead);
        NumberBytesToReadIn = VM_Context->Register_IOCount;
        VM_Context->Register_IOCount =
read(stdin,BufferToRead,NumberBytesToReadIn);
        return 1;
}
```

How can I assert there registers are other way used? Of course, you might think, something must *fill* them with values, no? It is a good point, but the reason is more reversing-side. The TVMContext.Registers[] pointer field is initialized with the head of the TVMContext structure. This means that generic instructions referring registers by TVMContext.Registers[] can freely access such general purpose registers.

Let us now examine in detail the main function of this VM, so to understand how it works from the scratch:

```
MemorySize = 4096;
initStack = SWAP(1);
initCode = SWAP(0x6EEFF);
byte * program;
int * OpcodeProc[];

int main() {
    dword RealEIP;
    TVMContext VMContext;
    TInstructionBuffer InstBuff;
    int res,MachineCheck;
    int c1, c2;
    char Opcode;

    /* 1. initialize VM */
    memset(VMContext,0,30*4);
    VMContext.Registers = &VMContext;
    if (*program!=0x102030) exit(1);
    //.text:004020A1
    VMContext.ProgramMemoryAddr = malloc(MemorySize+16);
    if (VMContext.ProgramMemoryAddr==0) exit(1);
    VMContext.InitCode = initCode;
    VMContext.MemorySize = MemorySize;
    memcpy(VMContext.ProgramMemoryAddr,program,2580+1);
    VMContext.StackMemoryAddr = malloc(MemorySize);
    VMContext.StackMemoryInit = initStack;
    if(VMContext.StackMemoryAddr==0) exit(1);
    //.text:00402111
    VM_EIP = initApp+28;
    VMContext.StackMemoryAddr= MemorySize;
    VMContext.VM_ESP = VMContext.StackMemoryInit;
    c1 = mcGenericError_or_CannotWriteTo;
    c2 = mcInputOutput;
    /* 2. start main VM Loop */
    while (true) { // .text:00402138
            // VM_Loop_Head_Default: .text:0040215B
            VMContext.InstructionCounter++;
            if (VMContext.VM_EFLAGS==TF) // Step-flag for
debugging purposes (code removed)
                    VMContext.MachineControl=mcStepBreakPoint;
            else {
                //--->body<--- .text:00402177
                VMContext.VMEIP_Saved_Prior_InstrExec=VM_EIP;
                /* 3. process a VM Instruction and execute it
*/
                MachineCheck =
VMAddress2Real(&VMContext,VM_EIP,&RealEIP);
                if (!MachineCheck) {
                        MachineCheck =
VMInstructionDecoder(&InstBuff,RealEIP);
                        if (!MachineCheck) // check for
opposite behavoir...
                                VMContext.MachineControl = c1;
                        else {
                                Opcode = *RealEIP;
                                MachineCheck =
(*OpcodeProc[(char)Opcode])(&InstBuff,&VMContext);
                                if (!MachineCheck) continue; //
check for opposite behavoir...
                        }
                }
                /* 4. if we have a Machine-Check to do, ensure
to catch the 'I/O' one  */
                if (MachineCheck &&
VMContext.maybe_MachineControl==c2) { // VM loop end.
c2==mcInputOutput
                        CheckForInputOutput(&VMContext);
                        continue;
                }
            }
            /* 5. perform the exception check  */
            // VM_MachineErrCheck_OrEndOfVM:
            if (VMContext.VM_ResumeExec==0)
                return 0;
            VM_EIP = VMContext.VM_ResumeExec;
            VMContext.VM_ResumeExec = 0;
   }
};
// end....
```

Let's comment it point to point (please note that I have rearranged and restructured a bit the code, as I did not like the messy structure it appears to have in the T2'06 code):

1. **Initialize VM**: this part of the code simply allocates memory, copy the VM program and initialize the start values, nothing more.
2. **Start main VM Loop**: this is the core of the virtual machine: here we count instructions, we test for special conditions (i.e. Trap-flag, I/O requests, Code Flow exceptions). In the VM Loop we convert the virtual EIP to an x86 address and we read and decode the virtual instruction at such address.
3. **Process a VM Instruction and execute it**: If the VM_EIP conversion and the instruction decoding goes well, we process the VM Instruction, feeding it with the machine's context and a buffer which helds the 'decoded' instruction data.
4. **If we have a Machine-Check to do, ensure to catch the 'I/O' one**: machine checks are not always errors: so, check its special conditions, as I/O requests.

5. **Perform the Exception check**: if due to the code flow or the errors we reach the 'Exception Check', the resume_exec register is tested: if null, application ends.

Before examining in detail the VM Instruction's management, better spend few lines on the VM memory management. As said, the VM memory is Bing Endian ordered. So, in a Little Endian machine, we need to convert back and forth all the values that moves there. This VM implements two functions for this step, Read_VMMemory_To() and Write_VMMemory_From(). Let's Examine their implementation, for getting an idea of the work:

```
.text:00401230     ; int __cdecl Write_VMMemory_From(int
VMAddress,int Ptr_ValueToWriteAndSwap,int VMValue_OperandSize,int
vm_context_ptr)
.text:00401230     Write_VMMemory_From proc near           ; CODE
XREF: Write_VMValue_To_Param+CB#p
.text:00401230                                             ;
VM_PUSH+3D#p ...
.text:00401230
.text:00401230     VMAddress       = dword ptr  4
.text:00401230     Ptr_ValueToWriteAndSwap= dword ptr  8
.text:00401230     VMValue_OperandSize= dword ptr  0Ch
.text:00401230     vm_context_ptr  = dword ptr  10h
.text:00401230
.text:00401230     ebx_param_vmvalue= ebx
.text:00401230
.text:00401230 000                 mov     eax,
[esp+Ptr_ValueToWriteAndSwap]
.text:00401234 000                 mov     edx, [esp+VMAddress]
.text:00401238 000                 push    ebx
.text:00401239 004                 lea     ecx,
[esp+4+Ptr_ValueToWriteAndSwap] ; Load Effective Address
.text:0040123D 004                 mov     ebx_param_vmvalue,
[eax]
.text:0040123F 004                 mov     eax,
[esp+4+vm_context_ptr]
.text:00401243 004                 push    ecx             ;
Write_RealAddr_To
.text:00401244 008                 push    edx             ;
VM_Address
.text:00401245 00C                 push    eax             ;
vm_context_ptr
.text:00401246 010                 call    VMAddress2Real  ; Call
Procedure
.text:00401246
.text:0040124B 010                 add     esp, 0Ch        ; Add
.text:0040124E 004                 test    eax, eax        ;
Logical Compare
.text:00401250 004                 jnz     short loc_401254 ;
Jump if Not Zero (ZF=0)
.text:00401250
.text:00401252 004                 pop     ebx_param_vmvalue
.text:00401253 000                 retn                    ;
Return Near from Procedure
.text:00401253
.text:00401254     ;
---------------------------------------------------------------
----------
.text:00401254
.text:00401254     loc_401254:                             ; CODE
XREF: Write_VMMemory_From+20#j
.text:00401254 004                 mov     eax,
[esp+4+VMValue_OperandSize]
.text:00401258 004                 dec     eax             ;
Decrement by 1
.text:00401259 004                 jz      short
op_byte_no_be_swap ; Jump if Zero (ZF=1)
.text:00401259
.text:0040125B 004                 dec     eax             ;
Decrement by 1
.text:0040125C 004                 jz      short
op_word_do_le2be_swap ; Jump if Zero (ZF=1)
.text:0040125C
.text:0040125E 004                 sub     eax, 2          ;
Integer Subtraction
.text:00401261 004                 jnz     short loc_4012AD ;
here below, operandsize is 4
... (the code here was already shown: it is the prior 'swap
endian' code)
.text:004012AD 004                 mov     eax, 1
.text:004012B2 004                 pop     ebx_param_vmvalue
.text:004012B3 000                 retn                    ;
Return Near from Procedure
```

```
.text:004012B3
.text:004012B3     Write_VMMemory_From endp

int Write_VMMemory_From(int VMAddress,int *LEValueSource, //
Ptr_ValueToWriteAndSwap
    int OperandSize, TVMContext* VMContext) // .text:00401230
{
    int *DestAddr;
    res = VMAddress2Real(VMContext,VMAddress,&DestAddr);
    switch(OperandSize) {
        case 1: *(byte*)DestAddr =
SWAP((byte)LEValueSource);break;
        case 2: *(word*)DestAddr =
SWAP((word)LEValueSource);break;
        case 4: *(dword*)DestAddr =
SWAP((dword)LEValueSource);break;
    }
    return 1;
}
```

As you can see, this procedure's code simply translates a memory virtual address to its x86 address, then write a value on it, swapping its Endianness. This procedure is obviously used on those procedure that manipulates i.e. the virtual stack. The read procedure is very similar, but it differs because the x86 memory is written, not the virtual one.

4 VM Instruction Core

The instruction core of this VM is represented by the buffer it uses for interpretating the VM program Opcodes and transfers them to the VM instruction scheduler. From this point of view, you could write your own VM Language by swapping the VM Decoder and keeping few fields attuned. This could be possible due to the fact the VM interpretation is layered within a buffer, which acts as an indirection layer. But let's examine such buffers in detail:

```
struct TparamDecoding{ // used by the decoding array to retrieve
i.e. the parameters usage of instructions
       int ID,
       int Params[3];
}

struct TSubInstr { // represents a parameter's field of the VM
Instruction
       int AddressType,
       int RegisterIdx,
       int Decoder_ParamsValue,
       int VMValue
}

struct TInstructionBuffer{
       int Length,
       int InstructionData,
       char InstrType,
       //char Fillers1[3], // if structure alignment is 1
       int Operand_Size,
       char InstructionParamsCount,
       //char Fillers2[3], // if structure alignment is 1
       SubInstr ParamDest,
       SubInstr paramSrc,
       SubInstr ParamThird,
       SubInstr *WorkSubField
};
```

These structures are very used by the VM. Let's examine now the functions that read and write the parameters:

```
enum TVMAddressType {
```

```
     vmaRegister      = 0,
     vmaRegisterAddress  = 1,
     vmaDirectAddress  = 2,
     vmaVMValue_orC4__or_displacement  = 3
}

int Retrieve_Param_Value(TInstructionBuffer *InstrBuff, SubInstr
*Param, // 14h/24h/34h
    int *WriteValueTo, TVMContext *VMContext) // .text:00401340
{
   int myLEvalue;

   switch(Param->AddressType) {
     case        vmaRegister:
         switch(InstrBuff->OperandSize){
             case 4: myLEvalue = (dword)VMContext-
>Registers[Param->RegisterIdx];break;
             case 2: myLEvalue = (word)VMContext->Registers[Param-
>RegisterIdx];break;
             case 1: myLEvalue = (char)VMContext->Registers[Param-
>RegisterIdx];break;
         }
     break;
     case vmaDirectAddress:
              vmaddr = vmAddress;
     case vmaRegisterAddres:
              if (Param->AddressType!=vmaDirectAddress) {
                    vmaddr =VMContext->Registers[Param-
>RegisterIdx];
              }
              res =Read_VMMemory_To(vmaddr,myLEvalue,InstrBuff-
>OperandSize)
              if (!res) return 0;
     break;
     case vmaVMValue:
              myLEvalue = ParamField->VMAddress;
     break:
     default:
   }
   *WriteValueTo=myLEvalue;
}
```

As we can notice, the procedure that read the value of the parameters distinguishes between the addressing type requested, retrieving the value from the VM register's set, from a virtual address in the virtual address memory, from a direct value. Whenever it needs to access the VM memory, it uses the appropriate function that reads memory and return a swapped (Little Endian) value. The function for writing the parameters is very similar.

At this point, only two another major functions needs to be examined, the one that takes care of Virtual Flags, and the VM Decoder itself.

The Evaluate_Flags() function is called within functions that might alter the flags status. For example the VM_CMP and VM_TEST instructions call this procedure for setting the appropriate VM flags. This function is also called after math operations, to keep coherent the internal status. An interesting part of this routine is that it receives a special parameters that indicates how the flag tests should be performed: it is used for math operations that alter the OF/CF flag.

```
int __cdecl Evaluate_Flags(int ParamAdditional, int
ParamEvaluate,
int TestType, TInstructionBuffer* Instruction_ptr, TVMContext*
VMContext) // .text:00401340
{
        TVMFLAGS *Flags = &VMContext->VMFlags;
        int OpSize = Instruction_ptr->OperandSize;
        int NegMark;
        switch(OpSize) {
        case 4: NegMark =0x80000000;Flags->ZF= ParamEvaluate==0;
break;
        case 2: NegMark =0x8000;Flags->ZF=
(word)ParamEvaluate==0; break;
        case 1: NegMark =0x80;Flags->ZF= (char)ParamEvaluate==0;
break;
        default: NegMark = ParamAdditional; // room for BTx
instructions expansion.
                 // custom evaluation of flags based on bit-
testing.
        }
        Flags->SF= (NegMark&ParamEvaluate)!=0;
        switch (TestType) {
        case 2: Flags->OF =
(NegMark&ParamEvaluate)==0&&(NegMark&ParamAdditional)!=0;
                 Flags->CF =
(NegMark&ParamEvaluate)!=0&&(NegMark&ParamAdditional)==0;
                 break;
        case 1: Flags->OF = !
((NegMark&ParamEvaluate)==0&&(NegMark&ParamAdditional)!=0);
                 Flags->CF = !
((NegMark&ParamEvaluate)!=0&&(NegMark&ParamAdditional)==0);
                 break;
        case 0:
        default:
        }
        return;
}
```

A side note: you can implement your own VM instruction that works on bit fields (BTS, BTC etc.) and then calls this function with an OperandSize different from the natural size (1-2-4). This causes the function to take the additional parameter as a bit mask for performing the BTX tests on such bit.

This is the Decoder function:

```
bool VMInstructionDecoder(TInstructionBuffer* InstructionPtr,
byte *VMEIP_RealAddr) { // .text:00401000
        TInstrTag InstrType;
        byte LowNib,HiNib;
        AddrSize;
        JccIndex;
        dword *ExaminedDwords;
        TempInstrSize;
        dd* TempPtr;
        ParamsCount;
        Temp;
        wTemp;
        bTemp;


        memset(InstrBuf,0,0x13*4);
        InstructionPtr->WorkSubField = &InstructionPtr-
>ParamDest; // set which is the first decoded param

    /* 1. set types */
         InstrType = VMEIP_RealAddr[0];//*(byte *)VMEIP_RealAddr
        InstructionPtr->InstrType = InstrType.InstrType;//b&0x3F;
// ==00111111b
        swith(InstrType.AddSize) { //swith(InstrType>>6) { //
the sub is needed for setting flags!
        case 0:  AddrSize = 1;    break;
        case 1:          AddrSize = 2;    break;
        case 2:          AddrSize = 4;    break;
        default: return 0;
        };
        InstructionPtr->OperandSize = AddrSize;
        ParamIdx= InstructionPtr->InstrType; // InstructionPtr-
>InstrType<<4; // *structure size
        if (ParamTable[ParamIdx].ID==0x33) return 0; // 0x33
entry has no associated instruction
        if ( (char)ParamTable[ParamIdx].ParamDest==4 &&
AddrSize!=4) return 0;
        InstructionPtr->InstrID = ParamTable[ParamIdx].ID; //
Jump Address!
        /* 2. cycle thru instruction parameters as from
Instruction Decoder's Table, and fill buffer */
        ExaminedParams = 0;
        TempInstrSize = 1; //0 was already used for getting
here!!
        ParamsCount = 0; // decode the first param, so!
        while (ParamTable[ParamIdx].Params[ParamsCount]!=0) { //
.text:004010B1  param decoding loop
                 //ParamsValue =
ParamTable[ParamIdx].Params[ParamsCount].ID;
                 LowNib_RegIdx =
VMEIP_RealAddr[TempInstrSize]&0x0F;
                 HiNib_AddrMode =
VMEIP_RealAddr[TempInstrSize]>>4;
                 InstructionPtr-
```

```
>WorkSubField[ParamsCount].AddressType = HiNib;
                /* 3. set up instruction sub-type (Jcc Type in
this VM) */
                InstructionPtr-
>WorkSubField[ParamsCount].Decoder_ParamsValue =
ParamTable[ParamIdx].Params[ParamsCount];
                switch (HiNib_AddrMode) { // NOTE: switch on
decoded address type!!
                case vmaRegister: // 0
                case vmaRegisterAddress: // 1
                        InstructionPtr-
>WorkSubField[ParaCount].RegisterIdx = LowNib_RegIdx;
                        TempInstrSize++;
                break;
                case vmaVMValue_orC4__or_displacement: // 3
.text:00401134
                        if
( (char)ParamTable[ParamIdx].Params[ParamsCount]==2) return 0;
                        TempInstrSize++;
                        switch(InstructionPtr->OperandSize) {
                        case 1:
                                bTemp = ((byte
*)VMEIP_RealAddr)[TempInstrSize];
                                InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)bTemp;
                                TempInstrSize++;
                                break;
                        case 2:
                                // this might be an instrinsic
inline function, due to code shape (compiler didnt recon param 2
was 0)
                                wTemp = ((word
*)VMEIP_RealAddr)[TempInstrSize];
                                wTemp = SWAP(wTemp);
                                InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)wTemp;
                                TempInstrSize+=2;
                        case 4:
                                break;
                        default: return 0;
                        }
                case vmaDirectAddress: // 2
                        if (HiNib_AddrMode==vmaDirectAddress) {
//added by me to keep flow
                                TempInstrSize++;
                                if (InstructionPtr-
>OperandSize!=4) return 0;
                        }
                        // .text:00401101 common code to case 2
and 3 here...
                        Temp = ((dword
*)VMEIP_RealAddr)[TempInstrSize];
                        Temp = SWAP(Temp);
                        InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)Temp;
                        TempInstrSize+=4;
                break;
                default:
                        return 0;
                }
                ParamsCount++; // next param data!
                ExaminedParams++;
                if (ParaCount>=3) break; // max 32 bytes fetched
this way

        };
        InstructionPtr->InstructionParamsCount = ExaminedParams;
        InstructionPtr->InstrSize = TempInstrSize;
        return TempInstrSize;
}
```

It simply fills the common parameters to instruction in the initial part, then it start cycling through the values of the Parameter's decoder structure. This structure holds the parameters usage for the instruction, as well as certain instruction sub-types, related -it seems- to the flag's usage in the Jcc. I had initially called this field “JccType”, then I changed it to a more general “InstrType”, as it could theoretically have a more general usage that the one exposed in the code.

The main loop on parameters is performed on the array of “ TParamDecoding.Params[]”. A Zero means that no further parameter is used by the instruction. Depending on the addressing type of the parameter, it fetch the right data and fills the TInstructionBuffer structure. As a note, the array of TParamDecoding starts at “.data:00407030 ParamTable” and ends up right before the VM Instruction table.

5 VM Instructions

The Instruction set encompasses the most common x86 instructions, with very little news. Here is the list of VM Operands implemented in the machine:

```
.data:00407430     VM_Opcode_Table dd offset VM_MOV        ; DATA
XREF: _main+162#r
.data:00407434                     dd offset VM_Multiple_op2
.data:00407438                     dd offset VM_Multiple_op2
.data:0040743C                     dd offset VM_Multiple_op2
.data:00407440                     dd offset VM_Multiple_op2
.data:00407444                     dd offset VM_Multiple_op2
.data:00407448                     dd offset VM_PUSH
.data:0040744C                     dd offset VM_POP
.data:00407450                     dd offset VM_JMP
.data:00407454                     dd offset VM_CALL
.data:00407458                     dd offset VM_LOOP
.data:0040745C                     dd offset VM_RET
.data:00407460                     dd offset VM_Multiple_op2
.data:00407464                     dd offset VM_INC
.data:00407468                     dd offset VM_DEC
.data:0040746C                     dd offset VM_NOP
.data:00407470                     dd offset VM_Jcc
.data:00407474                     dd offset VM_Jcc
.data:00407478                     dd offset VM_Jcc
.data:0040747C                     dd offset VM_Jcc
.data:00407480                     dd offset VM_Jcc
.data:00407484                     dd offset VM_Jcc
.data:00407488                     dd offset VM_Jcc
.data:0040748C                     dd offset VM_Jcc
.data:00407490                     dd offset VM_NOP
.data:00407494                     dd offset VM_NOP
.data:00407498                     dd offset VM_NOP
.data:0040749C                     dd offset VM_NOP
.data:004074A0                     dd offset VM_NOP
.data:004074A4                     dd offset VM_NOP
.data:004074A8                     dd offset VM_Multiple_op2
.data:004074AC                     dd offset VM_Multiple_op2
.data:004074B0                     dd offset VM_Multiple_op2
.data:004074B4                     dd offset VM_Multiple_op2
.data:004074B8                     dd offset VM_CMP
.data:004074BC                     dd offset VM_TEST
.data:004074C0                     dd offset VM_NOT
.data:004074C4                     dd offset VM_Multiple_op2
.data:004074C8                     dd offset VM_Multiple_op2
.data:004074CC                     dd offset VM_MOV_MEMADDR_TO
.data:004074D0                     dd offset VM_MOV_EIP_TO
.data:004074D4                     dd offset VM_SWAP
.data:004074D8                     dd offset VM_ADD_TO_ESP
.data:004074DC                     dd offset VM_SUB_FROM_ESP
.data:004074E0                     dd offset VM_MOV_FROM_ESP
.data:004074E4                     dd offset VM_MOV_TO_ESP
.data:004074E8                     dd offset VM_NOP
.data:004074EC                     dd offset VM_NOP
.data:004074F0                     dd offset VM_NOP
.data:004074F4                     dd offset VM_NOP
.data:004074F8                     dd offset VM_NOP
.data:004074FC                     dd offset VM_NOP
.data:00407500                     dd offset VM_NOP
.data:00407504                     dd offset VM_NOP
.data:00407508                     dd offset VM_NOP
.data:0040750C                     dd offset VM_NOP
.data:00407510                     dd offset VM_NOP
.data:00407514                     dd offset VM_NOP
.data:00407518                     dd offset VM_NOP
.data:0040751C                     dd offset VM_NOP
.data:00407520                     dd offset VM_Status_8
.data:00407524                     dd offset VM_SET_RESUME_EIP
.data:00407528                     dd offset VM_NOP
.data:0040752C                     dd offset VM_ALLOW_IO
```

There is not very much to say, except perhaps examining some of the instruction's implementation.

One that might be interesting is the following:

```
int __cdecl VM_LOOP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMCycleValue, VMValue;
```

```
        Retrieve_Param_Value(DecodedInstr->ParamSrc,
&VMCycleValue);
        VMCycleValue--;
        if (VMCycleValue==0) {
                NextInstr(VMContext,DecodedInstr);
                return 0;
        }
        Write_VMValue_To_Param(&DecodedInstr->ParamSrc,
&VMCycleValue,4,VMContext);
        Retrieve_Param_Value(DecodedInstr->ParamDest, &VMValue);
        if (DecodedInstr->AddressType==vmaVMValueOrDisplacement)
                VMValue+=VMContext->VM_EIP;
        VMContext->VM_EIP = VMValue;
        return 0;
}
```

As you may notice, this instruction retrieve a param's value -ideally the ECX equivalent- and decrements it. When it reaches Zero it ends up and skip to the next instruction, like LOOPCXZ, else it writes the decremented value to the source operand/register and performs a jump to the requested address, either relative or absolute depending on the addressing type.

Another instruction that might be of interest is the following:

```
int __cdecl VM_CALL(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue, VMRetValue;int res;

        VMRetValue = VMContext->VM_EIP+DecodedInstr->Length;
        res = Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMValue);
        if (!res) return 1;
        if (DecodedInstr->AddressType==vmaVMValueOrDisplacement)
                VMValue+=VMContext->VM_EIP;
        VMContext->VM_EIP = VMValue;
        VMContext->VM_ESP+=4;
        Write_VMMemory_From(&VMContext->VM_ESP,
&VMRetValue,4,VMContext);
        return 0;
}
```

As you can see, this instruction read the subroutine address, set up the VM_EIP for starting there at the next execution, then save the VM_EIP value of the next instruction (the one after the VM_CALL) in the stack, writing at VM_ESP. As noted before, stack addressing mode is swapped.

There are many other instructions to examine, but you can see the source code in appendix if you are looking for something special.

6 Conclusion

There are several things that are missing from this VM, most notably the instructions that operates on single bits, and the debugging layer support. Both are present *in nuce*, as we evicts from the possibility to have tests in Evaluate_Flags() for special bits other than the Sign one and the Trap Flag test on machine main cycle.

Well, I bored you enough, and bored me enough on writing this. Hope you learned something on Virtual Machines, and that you found this essay useful enough for whatever you might need it.

I wish to thank the whole RCE community for the great know-how freely shared. I just hope to give back what I learned from you all.

7 Appendix

Here you can find the (almost) whole reversed source code of T2'06 machine. Some parts are missing (all sanity checks for sure), and it will not compile as-is. It can however be used to encode your OWN Virtual Machine with little effort, as well as studying with more detail the T2'06.

Hope you appreciate it.

```
/*#define SWITCHSIZE(OpSize,Code4,Code2,Code1) switch( (OpSize) )
\
                                                case 4:
{Code4;} break; \
                                                case 2:
{Code2;} break; \
                                                case 1:
{Code1;} break; \
                                                };
*/

enum TVMAddressType {
     vmaRegister      = 0,
     vmaRegisterAddress  = 1,
     vmaDirectAddress  = 2,
     vmaVMValue_orC4__or_displacement  = 3
}

enum TMachineControlStatus {
     mcStepBreakPoint  = 2,
     mcWrongAddress   = 3,
     mcGenericError_or_CannotWriteTo  = 4,
     mcDivideByZero   = 5,
     mcInputOutput    = 9
}

enum TFLAGS {
     ZF               = 1,
     CF               = 2,
     OF               = 4,
     SF               = 8,
     TF               = 0x80
}

enum TIOFlags {
     DoOutput         = 1,
     DoInput          = 2
};
enum TJccType {
     jccJZ            = 0x10,
     jccJNZ           = 0x11,
     jccJS            = 0x12,
     jccJNS           = 0x13,
     jccJO            = 0x14,
     jccJNO           = 0x15,
     jccJB            = 0x16,
     jccJNB           = 0x17
}

struct TVMFLAGS {
// ~Compiler Dependent~ -please check the order!!
ZF:1, // supposed Bit 0
CF:1,// supposed Bit 1
OF:1,// supposed Bit 2
SF:1,// supposed Bit 3
Unused:3,// supposed Bit 4-6
TF:1,// supposed Bit 7
}

struct TInstrTag {
       AddrSize:2,
       InstrType:6
}

struct TVMContext {
       int Register_IOType,
       int Register_IO,
       int Register_IOAddress,
       int Register_IOCount,
       int GenericRegisters[12],
       int *Registers,
       int VM_ESP ,
       int VM_EIP ,
       int VMEIP_Saved_Prior_InstrExec,
       TVMFLAGS VM_EFLAGS,
       int InstructionCounter,
       int InitCode,
       int MemorySize,
       void* ProgramMemoryAddr,
       int Original_ESP,
       int StackMemorySize,
       void* StackMemoryAddr,
       int MachineControl,
       int VM_ResumeExec
}

struct TSubInstr {
       int AddressType,
       int RegisterIdx,
       int Decoder_ParamsValue,
       int VMValue
}

struct TParamDecoding{
       int ID,
       int Params[3];
}

struct TInstructionBuffer{
       int Length,
       int InstructionData,
       char InstrType,
       char Fillers1[3], // if structure alignment is 1, nothing
otherwise
       int Operand_Size,
       char InstructionParamsCount,
       char Fillers1[3], // if structure alignment is 1, nothing
otherwise
       SubInstr ParamDest,
       SubInstr paramSrc,
       SubInstr ParamThird,
       SubInstr *WorkSubField
};



/* this is the original Opcode Table Array, from IDA
.data:00407430     VM_Opcode_Table dd offset VM_MOV        ; DATA
XREF: _main+162#r
.data:00407434                     dd offset VM_Multiple_op2
.data:00407438                     dd offset VM_Multiple_op2
.data:0040743C                     dd offset VM_Multiple_op2
.data:00407440                     dd offset VM_Multiple_op2
.data:00407444                     dd offset VM_Multiple_op2
.data:00407448                     dd offset VM_PUSH
.data:0040744C                     dd offset VM_POP
.data:00407450                     dd offset VM_JMP
.data:00407454                     dd offset VM_CALL
.data:00407458                     dd offset VM_LOOP
.data:0040745C                     dd offset VM_RET
.data:00407460                     dd offset VM_Multiple_op2
.data:00407464                     dd offset VM_INC
.data:00407468                     dd offset VM_DEC
.data:0040746C                     dd offset VM_NOP
.data:00407470                     dd offset VM_Jcc
.data:00407474                     dd offset VM_Jcc
.data:00407478                     dd offset VM_Jcc
.data:0040747C                     dd offset VM_Jcc
.data:00407480                     dd offset VM_Jcc
.data:00407484                     dd offset VM_Jcc
.data:00407488                     dd offset VM_Jcc
.data:0040748C                     dd offset VM_Jcc
.data:00407490                     dd offset VM_NOP
.data:00407494                     dd offset VM_NOP
.data:00407498                     dd offset VM_NOP
.data:0040749C                     dd offset VM_NOP
.data:004074A0                     dd offset VM_NOP
.data:004074A4                     dd offset VM_NOP
.data:004074A8                     dd offset VM_Multiple_op2
.data:004074AC                     dd offset VM_Multiple_op2
.data:004074B0                     dd offset VM_Multiple_op2
.data:004074B4                     dd offset VM_Multiple_op2
.data:004074B8                     dd offset VM_CMP
.data:004074BC                     dd offset VM_TEST
.data:004074C0                     dd offset VM_NOT
.data:004074C4                     dd offset VM_Multiple_op2
.data:004074C8                     dd offset VM_Multiple_op2
.data:004074CC                     dd offset VM_MOV_MEMADDR_TO
.data:004074D0                     dd offset VM_MOV_EIP_TO
.data:004074D4                     dd offset VM_SWAP
.data:004074D8                     dd offset VM_ADD_TO_ESP
.data:004074DC                     dd offset VM_SUB_FROM_ESP
.data:004074E0                     dd offset VM_MOV_FROM_ESP
.data:004074E4                     dd offset VM_MOV_TO_ESP
.data:004074E8                     dd offset VM_NOP
.data:004074EC                     dd offset VM_NOP
.data:004074F0                     dd offset VM_NOP
.data:004074F4                     dd offset VM_NOP
.data:004074F8                     dd offset VM_NOP
.data:004074FC                     dd offset VM_NOP
.data:00407500                     dd offset VM_NOP
.data:00407504                     dd offset VM_NOP
.data:00407508                     dd offset VM_NOP
.data:0040750C                     dd offset VM_NOP
.data:00407510                     dd offset VM_NOP
.data:00407514                     dd offset VM_NOP
```

```
.data:00407518                          dd offset VM_NOP
.data:0040751C                          dd offset VM_NOP
.data:00407520                          dd offset VM_Status_8
.data:00407524                          dd offset VM_SET_RESUME_EIP
.data:00407528                          dd offset VM_NOP
.data:0040752C                          dd offset VM_ALLOW_IO
*/

void NextInstr(TVMContext &VMContext,TInstructionBuffer
&DecodedInstr) {VMContext.VM_EIP+=DecodedInstr.InstructionLength;
}
short int SWAP(short int x); // swap endian like above, x=
SWAP(x1)
Smallint SWAP(Smallint x);
int SWAP(int x);

#DEFINE BETWEEN(x,loBound,hiBound)
((x>loBound)&&(x<hiBound)?true:false)
#DEFINE RANGE(x,lowLimit,RangeFromLimit)
((x>lowLimit)&&(x<(lowLimit+RangeFromLimit))?true:false)

bool VMAddress2Real(TVMContext *VMContext,int VMAddress,int
*RealAddr) { // .text:004011D0
        if( RANGE(VMAddress,VMContext->InitCode,VMContext-
>MemorySize) ) {
                *RealAddr = (VM_Address-VMContext-
>InitCode)+VMContext->ProgramMemoryAddr;
                return 1;
        }
        if( RANGE(VMAddress,VMContext->Original_ESP,VMContext-
>StackMemorySize) ) {
                *RealAddr = (VM_Address-VMContext-
>Original_ESP)+VMContext->StackMemoryAddr;
                return 1;
        }
        VMContext->MachineControl = mcWrongAddress;
        return 0;

}

int Write_VMMemory_From(
    int VMAddress,
    int *LEValueSource,// Ptr_ValueToWriteAndSwap
    int OperandSize,
    TVMContext* VMContext)
{
    int *DestAddr;
    res = VMAddress2Real(VMContext,VMAddress,&DestAddr);
    switch(OperandSize) {
        case 1: *(byte*)DestAddr =
SWAP((byte)LEValueSource);break;
        case 2: *(word*)DestAddr =
SWAP((word)LEValueSource);break;
        case 4: *(dword*)DestAddr =
SWAP((dword)LEValueSource);break;
    }
    return 1;
}

//
// used for memory values
int Read_VMMemory_To(
    int VMAddress,
    int *Write_Address_To,
    int OperandSize,
    TVMContext* VMContext)
{
   // VMAddress will contain the real addr of memory location
   res = VMAddress2Real(VMContext,VMAddress,&VMAddress);
   switch(OperandSize) {
        case 1: *(byte*)Write_Address_To =
SWAP((byte)VMAddress);break;
        case 2: *(word*)Write_Address_To =
SWAP((word)VMAddress);break;
        case 4: *(dword*)Write_Address_To =
SWAP((dword)VMAddress);break;
    }
    //SwapEndianMem(AddressDataSize,VMAddress,Write_Address_To);
    return 1;
}

//
//

int Retrieve_Param_Value(
    TInstructionBuffer *InstrBuff,
    SubInstr *Param, // 14h/24h/34h
    int *WriteValueTo,
    TVMContext *VMContext)
{
   int myLEvalue;

   switch(Param->AddressType) {
     case        vmaRegister:
        switch(InstrBuff->OperandSize){
            case 4: myLEvalue = (dword)VMContext-
>Registers[Param->RegisterIdx];break;
            case 2: myLEvalue = (word)VMContext->Registers[Param-
>RegisterIdx];break;
            case 1: myLEvalue = (char)VMContext->Registers[Param-
>RegisterIdx];break;
        }
     break;
     case vmaDirectAddress:
              vmaddr = vmAddress;
     case vmaRegisterAddres:
              if (Param->AddressType!=vmaDirectAddress) {
                    vmaddr =VMContext->Registers[Param-
>RegisterIdx];
              }
              res =Read_VMMemory_To(vmaddr,myLEvalue,InstrBuff-
>OperandSize)
              if (!res) return 0;
     break;
     case vmaVMValue:
              myLEvalue = ParamField->VMAddress;
     break:
     default:
   }
   *WriteValueTo=myLEvalue;
}

//
//

int __cdecl Write_VMValue_To_Param(
    TInstructionBuffer *InstrBuff,
    TSubInstr *Param, // 14h/24h/34h
    int *WriteValueTo,
    TVMContext *VMContext)
{
switch(Param->AddressType) {
case     vmaRegister:
         switch(InstrBuff->OperandSize){
         case 4: VMContext->Registers[Param->RegisterIdx] =
ValueToWriteTo;break;
         case 2: VMContext->Registers[Param->RegisterIdx] =
(word)ValueToWriteTo|(value&!0xFFFF);break;
         case 1: VMContext->Registers[Param->RegisterIdx] =
(char)ValueToWriteTo|(value&!0xFF);break;
         default: return 1;
         }
break;
case     vmaDirectAddress:
          vmaddr = vmAddress;
case     vmaRegisterAddres:
         if (Param->AddressType!=vmaDirectAddress) {
                  vmaddr =VMContext->Registers[Param-
>RegisterIdx];
         }
         res
=Write_VMMemory_From(vmaddr,&ValueToWriteTo,InstrBuff-
>OperandSize,VMContext)
         if (!res) return 0;
break;
case     vmaVMValue:
         VMContext->MachineControl =
mcGenericError_or_CannotWriteTo;
break:
default:
}
return 1;
}


int __cdecl Evaluate_Flags(
int ParamAdditional,
int ParamEvaluate,
int TestType,
TInstructionBuffer* Instruction_ptr,
TVMContext* VMContext)
{
         TVMFLAGS *Flags = &VMContext->VMFlags;
         int OpSize = Instruction_ptr->OperandSize;
         int NegMark;
         switch(OpSize) {
         case 4: NegMark =0x80000000;Flags->ZF= ParamEvaluate==0;
break;
         case 2: NegMark =0x8000;Flags->ZF=
(word)ParamEvaluate==0; break;
         case 1: NegMark =0x80;Flags->ZF= (char)ParamEvaluate==0;
break;
         default: NegMark = ParamAdditional; // room for BTx
instructions expansion.
                  // custom evaluation of flags based on bit-
testing.
         }
         Flags->SF= (NegMark&ParamEvaluate)!=0;
         switch (TestType) {
         case 2: Flags->OF =
(NegMark&ParamEvaluate)==0&&(NegMark&ParamAdditional)!=0;
                  Flags->CF =
(NegMark&ParamEvaluate)!=0&&(NegMark&ParamAdditional)==0;
                  break;
         case 1: Flags->OF = !
((NegMark&ParamEvaluate)==0&&(NegMark&ParamAdditional)!=0);
                  Flags->CF = !
((NegMark&ParamEvaluate)!=0&&(NegMark&ParamAdditional)==0);
```

```
                break;
        case 0:
        default:
        }
        return;
}

//
//

int __cdecl VM_MOV_EIP_TO(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int res = Write_VMValue_To_Param(&DecodedInstr-
>ParamDest,VMContext->VMEIP,VMContext);
        if (res) return 1;
        NextInstr(VMContext,DecodedInstr);
        return 0;
}

int __cdecl VM_MOV_MEMADDR_TO(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int VMAddress;

        switch(DecodedInstr->ParamSrc) {
        case vmaRegisterAddress:
                VMAddress = VMContext->Registers[DecodedInstr-
>ParamSrc.RegisterIdx];
        break;
        case vmaDirectAddress:
                VMAddress = DecodedInstr->ParamSrc.VMAddress;
        break;
        default:
                VMContext->MachineControl = mcCannotWriteTo;
                return 0;
        }
        Write_VMValue_To_Param(&DecodedInstr.ParamDest,
VMAddress,VMContext);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_ADD_TO_ESP(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamDest,&VMValue);
        VMContext->VM_ESP+= VMValue;
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_SUB_FROM_ESP(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamDest,&VMValue);
        VMContext->VM_ESP-= VMValue;
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_MOV_FROM_ESP(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int VMValue;

        Write_VMValue_To_Param(DecodedInstr->ParamDest,
&VMContext.VM_ESP);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_MOV_TO_ESP(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr)
{
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMContext.VM_ESP);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_MOV(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr)
{
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamSrc, &VMValue);
        Write_VMValue_To_Param(&DecodedInstr->ParamDest,
&VMValue,VMContext);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_NOT(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr)
{
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamSrc, &VMValue);
        VMValue=!VMValue;
        Evaluate_Flags(VMValue,0,DecodedInstr,VMContext);
        Write_VMValue_To_Param(&DecodedInstr->ParamSrc,
&VMValue,VMContext);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_CMP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr)
{
        int VMValue, VMValueSrc,VMValueDst;

        Retrieve_Param_Value(DecodedInstr->ParamSrc,
&VMValueSrc);
        Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMValueDst);
        switch(DecodedInstr->OperandSize){
        case 4: VMValue = (dword)VMValueDst-
(dword)VMValueSrc;break;
        case 2: VMValue = (word)VMValueDst-
(word)VMValueSrc;break;
        case 1: VMValue = (char)VMValueDst-
(char)VMValueSrc;break;
        }

Evaluate_Flags(VMContext,VMValue,VMValueSrc,2,DecodedInstr,VMCont
ext);
        NextInstr(VMContext,DecodedInstr);
}
int __cdecl VM_TEST(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {/* as above, make and */};
int __cdecl VM_XCHG(VMContext,DecodedInstr) {
        int VMValueSrc,VMValueDst;

        Retrieve_Param_Value(DecodedInstr->ParamSrc,
&VMValueSrc);
        Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMValueDst);
        Write_VMValue_To_Param(&DecodedInstr->ParamSrc,
&VMValueDst,VMContext);
        Write_VMValue_To_Param(&DecodedInstr->ParamDest,
&VMValueSrc,VMContext);
        NextInstr(VMContext,DecodedInstr);

}
int __cdecl VM_INC(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {...};
int __cdecl VM_DEC(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {...};

int __cdecl VM_PUSH(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue; int res;

        Retrieve_Param_Value(DecodedInstr->ParamSrc, &VMValue);
        VMContext.VM_ESP+=4;
        Write_VMValue_To_Param(&VMContext->VM_ESP,
&VMValue,4,VMContext);
        NextInstr(VMContext,DecodedInstr);
}
int __cdecl VM_POP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue;

        Read_VMMemory_To(VMContext->VM_ESP, &VMValue);
        Write_VMValue_To_Param(&DecodedInstr->ParamDest,
&VMValue,4,VMContext);
        VMContext.VM_ESP-=4;
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_Jcc(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue;
        bool DoJump=false;

        // Original code is a bit different here:
        // BUT it is more professional this way.
        switch(DecodedInstr->InstrType) {
        case jccJZ: DoJump = VMContext->VM_EFlags&ZF; break;
        case jccJNZ: DoJump = !VMContext->VM_EFlags&ZF; break;
        case jccJS: DoJump = VMContext->VM_EFlags&SF; break;
        case jccJNS: DoJump = !VMContext->VM_EFlags&SF; break;
        case jccJO: DoJump = VMContext->VM_EFlags&OF; break;
        case jccJNO: DoJump = !VMContext->VM_EFlags&OF; break;
        case jccJB: DoJump = VMContext->VM_EFlags&CF; break;
        case jccJNB: DoJump = !VMContext->VM_EFlags&CF; break;
        default: break;
        }
        if(!DoJump) {
                NextInstr(VMContext,DecodedInstr);
                return 0;
        }
        res = Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMValue);
        if (!res) return 1;
        if (DecodedInstr->AddressType==vmaVMValueOrDisplacement)
                VMValue+=VMContext->VM_EIP;
        VMContext->VM_EIP = VMValue;
```

```
        return 0;
}

int __cdecl VM_JMP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {...};

int __cdecl VM_CALL(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue, VMRetValue;int res;

        VMRetValue = VMContext->VM_EIP+DecodedInstr->Length;
        res = Retrieve_Param_Value(DecodedInstr->ParamDest,
&VMValue);
        if (!res) return 1;
        if (DecodedInstr->AddressType==vmaVMValueOrDisplacement)
                VMValue+=VMContext->VM_EIP;
        VMContext->VM_EIP = VMValue;
        VMContext->VM_ESP+=4;
        Write_VMValue_To_Param(&VMContext->VM_ESP,
&VMRetValue,4,VMContext);
        return 0;
}

int __cdecl VM_LOOP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMCycleValue, VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamSrc,
&VMCycleValue);
        VMCycleValue--;
        if (VMCycleValue==0) {
                NextInstr(VMContext,DecodedInstr);
                return 0;
        }
        Write_VMValue_To_Param(&DecodedInstr->ParamSrc,
&VMCycleValue,4,VMContext);
        Retrieve_Param_Value(DecodedInstr->ParamDest, &VMValue);
        if (DecodedInstr->AddressType==vmaVMValueOrDisplacement)
                VMValue+=VMContext->VM_EIP;
        VMContext->VM_EIP = VMValue;
        return 0;
}

int __cdecl VM_RET(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {
        int VMValue;

        Read_VMMemory_To(VMContext->VM_ESP, &VMValue);
        VMContext->VM_ESP-=4;
        VMContext->VM_EIP = VMValue;
}

int __cdecl VM_RESUME_EIP(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr) {
        // set the resume address on error/the end condition
        int VMValue;

        Retrieve_Param_Value(DecodedInstr->ParamDest, &VMValue);
        VMContext->VM_ResumeExec = VMValue;
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl VM_ALLOW_IO(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr) {

        VMContext->MachineControl = mcInputOutput;
        NextInstr(VMContext,DecodedInstr);
        // very ugly: for having IO you must force a
MachineControl check, as if error were in.
        return 1;
}

int __cdecl VM_NOP(TVMContext* VMContext, TInstructionBuffer*
DecodedInstr) {NextInstr(VMContext,DecodedInstr);}

// unsigned -> signed
int __cdecl VM_MULTIPLE_OP2(TVMContext* VMContext,
TInstructionBuffer* DecodedInstr) {
        int VMValueDst,VMValueThird,VMValueSrc,VMValueEval;
        int ModeSwitch = 0;

        Retrieve_Param_Value(DecodedInstr->ParamThird,
&VMValueThird);
        Retrieve_Param_Value(DecodedInstr->ParamSource,
&VMValueSrc);
        switch(DecodedInstr->InstrID-1) {
        case 0: // ADD
                ModeSwitch = 2;
                switch(DecodedInstr->OperandSize) {
          case 1: VMValueEval = (char)VMValueSrc +
(char)VMValueThird; break;
                  case 2: VMValueEval = (word)VMValueSrc +
(word)VMValueThird; break;
                  case 4:       VMValueEval = VMValueSrc +
VMValueThird;
        }
        break;
        case 1: // SUB
                ModeSwitch = 1;
                switch(DecodedInstr->OperandSize) {
                   case 1:       VMValueEval = (char)VMValueSrc
- (char)VMValueThird; break;
                  case 2:       VMValueEval = (word)VMValueSrc
- (word)VMValueThird; break;
                  case 4:       VMValueEval = VMValueSrc -
VMValueThird;
        }
        break;
        case 2: // XOR
                switch(DecodedInstr->OperandSize) {
                   case 1:      VMValueEval = (char)VMValueSrc
^ (char)VMValueThird; break;
                   case 2:      VMValueEval = (word)VMValueSrc
^ (word)VMValueThird;break;
                   case 4:      VMValueEval = VMValueSrc ^
VMValueThird;
        }
        break;
        case 10: // AND - copy&paste above.
        break;
        case 11: // OR - copy&paste above.
        break;
        case 6: // SHL - copy&paste above.
        break;
        case 7: // SHR - copy&paste above.
        break;
        case 8: // ROL - copy&paste above.
        break;
        case 9: // ROR - copy&paste above.
        break;
        case 4: // IMUL - copy&paste above.
        break;
        case 5: // IDIV
                if (VMValueThird==0) {
                        VMContext->MachineControl =
mcDivideByZero;
                        return 0;
                }
                switch(DecodedInstr->OperandSize) {
                   case 1:      VMValueEval = (byte)VMValueSrc
/ (byte)VMValueThird; break;
                   case 2:      VMValueEval = (word)VMValueSrc
/ (word)VMValueThird; break;
                   case 4:      VMValueEval = VMValueSrc /
VMValueThird;
               }
        break;
        case 5: // IDIVREST - copy&paste above.
        break;
        default: // opcode 12 and follows
        }

Evaluate_Flags(VMValueSrc,VMValueEval,ModeSwitch,InstructionPtr,V
mContext);
        Write_VMValue_To_Param(InstructionPtr, DecodedInstr-
>ParamDest, &VMValueEval,VMContext);
        NextInstr(VMContext,DecodedInstr);
}

int __cdecl Do_Write_Output(TVMContext* VMContext) {

        int NumberBytesToWriteOut;
        void *BufferToWrite;

        if (VMContext->Register_IO!=0) return 0;

        VMAddress2Real(VMContext,VMContext-
>Register_IOAddress,&BufferToWrite);
        NumberBytesToWriteOut = VM_Context->Register_IOCount; //
        VM_Context->Register_IOCount =
write(stdout,BufferToWrite,NumberBytesToWriteOut);
        return 1;
}
int __cdecl Do_Read_Input(TVMContext* VMContext)  {

        int NumberBytesToReadIn;
        void *BufferToRead;

        if (VMContext->Register_IO!=0) return 0;

        VMAddress2Real(VMContext,VMContext-
>Register_IOAddress,&BufferToRead);
        NumberBytesToReadIn = VM_Context->Register_IOCount;
        VM_Context->Register_IOCount =
read(stdin,BufferToRead,NumberBytesToReadIn);
        return 1;
}


int __cdecl CheckForIO(VM_Context) {

        switch(VMContext.Register_IOType) {
        case 2: return Do_Write_Output(VM_Context);
        break;
        case 3: return Do_Read_Input(VM_Context);
        break;
        default: return 1;
        }
```

```
}

bool VMInstructionDecoder(TInstructionBuffer* InstructionPtr,
byte *VMEIP_RealAddr) { // .text:00401000
        TInstrTag InstrType;
        byte LowNib,HiNib;
        AddrSize;
        JccIndex;
        dword *ExaminedDwords;
        TempInstrSize;
        dd* TempPtr;
        ParamsCount;
        Temp;
        wTemp;
        bTemp;


        memset(InstrBuf,0,0x13*4);
        InstructionPtr->WorkSubField = &InstructionPtr-
>ParamDest; // set which is the first decoded param


        InstrType = VMEIP_RealAddr[0];//*(byte *)VMEIP_RealAddr
    InstructionPtr->InstrType = InstrType.InstrType;//b&0x3F; //
==00111111b
        //swith(InstrType>>6) { // the sub is needed for setting
flags!
        swith(InstrType.AddSize)
        case 0:  AddrSize = 1;    break;
        case 1:          AddrSize = 2;    break;
        case 2:          AddrSize = 4;    break;
        default: return 0;
        };
        InstructionPtr->OperandSize = AddrSize;
        //ParamIdx= InstructionPtr->InstrType<<4; // *structure
size
        ParamIdx= InstructionPtr->InstrType;
        if (ParamTable[ParamIdx].ID==0x33) return 0; // 0x33
entries has no associated instruction
        if ( (char)ParamTable[ParamIdx].ParamDest==4 &&
AddrSize!=4) return 0;
        InstructionPtr->InstrID = ParamTable[ParamIdx].ID; //
Jump Address!
        ExaminedParams = 0;
        TempInstrSize = 1; //0 was already used for getting
here!!
        ParamsCount = 0; // decode the first param, so!
        // ParamTable[ParamIdx].Params[ParamsCount]; // 401099
        while (ParamTable[ParamIdx].Params[ParamsCount]!=0) { //
.text:004010B1  param decoding loop
                ParamsValue =
ParamTable[ParamIdx].Params[ParamsCount];
                LowNib_RegIdx =
VMEIP_RealAddr[TempInstrSize]&0x0F;
                HiNib_AddrMode =
VMEIP_RealAddr[TempInstrSize]>>4;
                InstructionPtr-
>WorkSubField[ParamsCount].AddressType = HiNib;
                InstructionPtr->WorkSubField[ParamsCount].field8
= ParamTable[ParamIdx].Params[ParamsCount];
                switch (HiNib_AddrMode) { // NOTE: switch on
decoded address type!!
                case vmaRegister: // 0
                case vmaRegisterAddress: // 1
                        InstructionPtr-
>WorkSubField[ParaCount].RegisterIdx = LowNib_RegIdx;
                        TempInstrSize++;
                break;
                case vmaVMValue_orC4__or_displacement: // 3
.text:00401134
                        if
( (char)ParamTable[ParamIdx].Params[ParamsCount]==2) return 0;
                        TempInstrSize++;
                        switch(InstructionPtr->OperandSize) {
                        case 1:
                                bTemp = ((byte
*)VMEIP_RealAddr)[TempInstrSize];
                                InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)bTemp;
                                TempInstrSize++;
                                break;
                        case 2:
                                // this might be an instrinsic
inline function, due to code shape (compiler didnt recon param 2
was 0)
                                wTemp = ((word
*)VMEIP_RealAddr)[TempInstrSize];
                                wTemp = SWAP(wTemp);
                                InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)wTemp;
                                TempInstrSize+=2;
                        case 4:
                                break;
                        default: return 0;
                        }
                case vmaDirectAddress: // 2
                        if (HiNib_AddrMode==vmaDirectAddress) {
//added by me to keep flow
                                TempInstrSize++;
                                if (InstructionPtr-
>OperandSize!=4) return 0;
                        }
                        // .text:00401101 common code to case 2
and 3 here...
                        Temp = ((dword
*)VMEIP_RealAddr)[TempInstrSize];
                        Temp = SWAP(Temp);
                        InstructionPtr-
>WorkSubField[ParamsCount].VMValue = (dword)Temp;
                        TempInstrSize+=4;
                break;
                default:
                        return 0;
                }
                ParamsCount++; // next param data!
                ExaminedParams++;
                if (ParaCount>=3) break; // max 32 bytes fetched
this way

        };
        InstructionPtr->InstructionParamsCount = ExaminedParams;
        InstructionPtr->InstrSize = TempInstrSize;
        return TempInstrSize;
}

#DEFINE BETWEEN(x,loBound,hiBound)
((x>loBound)&&(x<hiBound)?true:false)
#DEFINE RANGE(x,lowLimit,RangeFromLimit)
((x>lowLimit)&&(x<(lowLimit+RangeFromLimit))?true:false)


MemorySize = 4096;
initStack = SWAP(1);
initCode = SWAP(0x6EEFF);
byte * program;
int * OpcodeProc[];

int main() {
    dword RealEIP;
    TVMContext VMContext;
    TInstructionBuffer InstBuff;
    int res,MachineCheck;
    int c1, c2;
    char Opcode;

    /* 1. initialize VM */
    memset(VMContext,0,30*4);
    VMContext.Registers = &VMContext;
    if (*program!=0x102030) exit(1);
    //.text:004020A1
    VMContext.ProgramMemoryAddr = malloc(MemorySize+16);
    if (VMContext.ProgramMemoryAddr==0) exit(1);
    VMContext.InitCode = initCode;
    VMContext.MemorySize = MemorySize;
    memcpy(VMContext.ProgramMemoryAddr,program,2580+1);
    VMContext.StackMemoryAddr = malloc(MemorySize);
    VMContext.StackMemoryInit = initStack;
    if(VMContext.StackMemoryAddr==0) exit(1);
    //.text:00402111
    VM_EIP = initApp+28;
    VMContext.StackMemoryAddr= MemorySize;
    VMContext.VM_ESP = VMContext.StackMemoryInit;
    c1 = mcGenericError_or_CannotWriteTo;
    c2 = mcInputOutput;
    /* 2. start main VM Loop */
    while (true) { // .text:00402138
                // VM_Loop_Head_Default: .text:0040215B
                VMContext.InstructionCounter++;
                if (VMContext.VM_EFLAGS==TF) // Step-flag for
debugging purposes (code removed)
                        VMContext.MachineControl=mcStepBreakPoint;
                else {
              //--->body<--- .text:00402177
              VMContext.VMEIP_Saved_Prior_InstrExec=VM_EIP;
              /* 3. process a VM Instruction and execute it  */
              MachineCheck =
VMAddress2Real(&VMContext,VM_EIP,&RealEIP);
              if (!MachineCheck) {
                  MachineCheck =
VMInstructionDecoder(&InstBuff,RealEIP);
                  if (!MachineCheck) // check for opposite
behavoir...
                      VMContext.MachineControl = c1;
                  else {
                      Opcode = *RealEIP;
                      MachineCheck =
(*OpcodeProc[(char)Opcode])(&InstBuff,&VMContext);

                                if (!MachineCheck) continue; //
check for opposite behavoir...
                  }
              }
              /* 4. if we have a MachineCheck to do, ensure to
catch the 'IO' one  */
              if (MachineCheck &&
```

```
VMContext.maybe_MachineControl==c2) { // VM loop end.
c2==mcInputOutput
                          CheckForInputOutput(&VMContext);
                          continue;
             }
          }
          /* 5. perform the exception check  */
          // VM_MachineErrCheck_OrEndOfVM:
          if (VMContext.VM_ResumeExec==0) return 0;
              VM_EIP = VMContext.VM_ResumeExec;
              VMContext.VM_ResumeExec = 0;
   }
   // end....
};
```