虚拟机逆向

T206 逆向源码分析

Reverser：Maximus

摘要：虚拟机是目前讨论最多加密方式之一，我将通过分析整个 T206 挑战（价值 1500 刀）的逆向源代码，尝试着揭开虚拟机构建的原理。本文解释了如何编制一个虚拟机，同时给出了它的源代码和结构，以帮助那些想分析这个挑战的人。完整的逆向源代码在附录中。

关键词：虚拟机；VM；逆向；编码；分析

介绍

逆向一个虚拟机不一定需要对虚拟机全面了解。通常如果可能的话，我们可以用反汇编工具对虚拟机结构进行快速的分析，然后我们调试它以观察这些“活”的代码的移动，它们如何实现我们的初始目标，它们到底“做”了什么。在T206这个应用中，我完整的逆向了它，并给出了它的重建源代码。解决这个问题，并不需要给出它的完整源代码。我之所以这样做，是因为它与我之前的一篇虚拟机教程有关。我收到了许多更复杂或更简单的需求，有很多出色的逆向者由于缺乏理论支持，还无法面对这个新技术。因此，这篇小文，深入探讨了虚拟机的结构，虚拟机逆向编码和逆向构建。

如果只是想对VM(虚拟机英文首字母所写)有个简单了解，我强烈建议你读一下我之前写的虚拟机基础教程。你可以在《编码破解杂志》（CBM）上找到。请注意，建议当你读本文的时候，打开IDA或WDASM。为了简短，我忽略了一些“完整性检查”。同时，请注意，我没有调试T206挑战：我使用IDA4.3-没有调试器的那一款，也没有运行OLLY。（注1，老实说，我在挑战刚开始的时候使用了它，以检查它是否有壳）

好了，让我们打开MP3，开工吧。

通用方法和结构

当我打开T206几秒后，我马上意识到我面前的是一个虚拟机：你会问，这是如何知道的呢？如果你仔细看它的主函数_main()，你会注意到一个普通的类似一个调度器的循环，它循环调用从一张函数表选择出来的函数。这看起来非常像一个虚拟机的基本内核。

ww
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
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)

Opcode = *RealEIP;
MachineCheck = (*OpcodeProc[(char)Opcode])(&InstBuff,&VMContext);
if (!MachineCheck) continue; // check for opposite behavoir...
```

一旦我们找到了这个调度分派中心，下一步就可以看虚操作代码了（virtual opcodes），我们可以尝试着定位指令指针和部分指令，并试图去了解虚拟机环境（寄存器、虚拟内存、虚拟堆栈、虚拟堆等）。

如果你快速浏览指令系列，你很快就会发现下面的指令：

```
; int __cdecl VM_NOP(int Instruction_Ptr,int VM_Context_Ptr)
.text:00401F80 VM_NOP proc near ; CODE XREF: _main+162#p
.text:00401F80 ; DATA XREF: .data:0040746C#o ...
.text:00401F80
.text:00401F80     Instruction_Ptr = dword ptr 4
.text:00401F80     VM_Context_Ptr = dword ptr 8
.text:00401F80
.text:00401F80 000     mov ecx, [esp+Instruction_Ptr]
.text:00401F84 000     mov eax, [esp+VM_Context_Ptr]
.text:00401F88 000     mov edx, [ecx]
.text:00401F8A 000     mov ecx, [eax+VM_Context.VM_EIP]
.text:00401F8D 000     add ecx, edx ; Add
.text:00401F8F 000     mov [eax+VM_Context.VM_EIP], 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

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

不难看出，这是个NOP指令。它每次只是简单的取了一个地址(VM_EIP)并给这个地址加上一个数值。看上去像这样“EIP+=InstructionLength”，不是吗？如果你注意到这个格式会重复出现在许多其它指令的后面，你可以赌它就是NOP了。它也可以让我们对一个包含通用虚拟机寄存器和参数的结构有所了解-那就是VMContext(虚拟机运行环境)结构。如果你比我聪明，你会马上注意到这个包含VM_EIP的结构被放置在调用函数堆栈空间中，这意味着VM结构被当作_main()函数的局部变量来处理。不知有没有注意到，我不得不在IDA中重新命名_main()模块的相关域，以与恢复的VM结构名字相匹配。这可以让_main()函数看起来更容易理解。

然而，这些甚至还够不上冰山的一角。复杂的工作还没开始，特别是如果你的灵光没有及时出现（正如我一样，花费了数小时的逆向时间）。另一个有趣的指令是JCC（条件跳转系列指令），它位于.text:00401C80。这条指令可以很容易被识别出，因为他执行了一堆条件

PDF created with pdfFactory Pro trial version www.pdffactory.com

测试，而且用到了我们的VM_EIP，它会根据条件测试的结果来增加或改变VM_EIP值。继续查看代码，我们可以看到很多指令调用一些内部函数来完成一些未知的工作。在开始的时候最好先不理会它们（免得陷进去），关注那些容易一些的指令，或至少试着找出复杂的代码中一些熟悉的可理解的片段。例如，我们可以发现一个进行很多数学运算的指令，根据字节/字/双字来区分。你可以通过检查数值的格式（1-2-4）和检查这个数值相关的操作来区分它们…

```
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

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;; int __cdecl VM_NOP(int Instruction_Ptr,int VM_Context_Ptr)
}
```

然而，这些发现还远远不够。我们使用无法调试的IDA（V4.3），我们不知道在函数被调用前和调用后的情况。因此我们在一个困难的条件下来做这些工作-有趣。

我们可以识别的另一对有趣的指令是，VM_DEC和VM_INC。它们可以很容易的被识别，因为他们使用了inc(x)和dec(x)。同样，VM_NOT指令也可以通过这个方式找到。但是，之前或之后，我们必须开始“搬动真正的VM石头”了。好吧，我们选一个像VM_NOT的指令，并尝试研究它调用的过程。通过四处浏览，和查找保存“操作数大小”的域的所在，你可以认识到另一个传到指令的缓存，是一个“指令保存”缓存。如果你看一下_main()函数，你

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

会注意到在VMopcode(虚拟机指令)在执行之前，有一个函数调用来详细解释指令…你可以确认它是VM指令解码器所在。

```
.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...
```

可以通过检查用于跳转到指令执行的数值，可以识别出RealEIP的名字。这个索引是以字节表示的，它只能是指令操作码本身。

在我们这些随机分析之后，我们可以回头看一下_main()函数中，在我们的解码器之前的一个小的函数调用。如果我们看主函数和这个函数的执行流程，我们可以看出我们的VM_EIP地址在这个函数中处理，在这个函数中，VM_EIP被限定在两个区块内检测。我赌其中一个是VM的内存空间，另一个是VM的堆栈空间。当然，我是对的。

我们来恢复一下这个函数。

```
:004011D0
.text:004011D        ; int __cdecl VMAddress2Real(int vm_context_ptr,int VM_Address,int Write_RealAddr_To)
.text:004011D0       VMAddress2Real proc near ; CODE XREF: Write_VMMemory_From+16#p
.text:004011D0     ...

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;
        }
```

当你看上面的代码的时候，你会注意到我们恢复了一个“机器控制”寄存器，和它的一个状态。这个寄存器在很多不同的地方（那些错误条件出现时,这说明我在做这个假定前，使用了交叉索引）被设置，因此很自然的，我把它与机器控制寄存器联系起来。同时我们也注意到，它不仅仅是一个“错误状态”寄存器，它可以用来表示与错误不同的及其状态，例如：当VM与其它代码进行交互时，使用了机器控制寄存器的一个数值来标记。事实上，下面的虚拟机指令就是这么做的。

```
: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]
```


PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.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: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;
```

你可以从我给出的注释中看出，我不喜欢这个处理方法。我发现它有点笨拙。

```
.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;
```

从上面的代码中，你可以看出它是如何使用的，原始的应用并不像我这样使用“MachineCheck”（机器检查）。在_main()中的代码使我想了很多关于原始代码的事：

我希望没有goto的语义出现。那将真是让人头痛的事。

在这个分析过程中，我们可以尝试定位那些堆栈相关指令，正如我们开始猜想的那样，那些改变VM_ESP和管理堆栈的指令可以被找出。即使我们没有分析这些被这些指令调用的过程，至少可以理解他们是如何工作的，然后给它们添加相应的标签（注释）。在我们开始研究这些内部函数之前，定位CALL/RET指令对不是容易的事情：看一下下面的VM_RET：

```
.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
```


PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.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

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;
}
```

你会注意到，从堆栈恢复的数值由一个特定的处理虚拟内存的过程来完成。然而，四处查看，一旦你捕捉到进程调用的返回值，这个数值被保存在EIP中…同时这个函数操作我们可能的VM_ESP，你能够开始有一个大致轮廓了（你会注意到堆栈使用了一个“不同”的方向）

事实上，我用了另一种方法：我开始直接进攻内部指令函数，以找出他们到底做了什么。在分析过程中，当整个轮廓快被描绘出来的时候，我遇到了一个在text:00401E10的指令DEC EAX，我不能理解它为什么在这里-几分钟后，我放弃了它，我肯定会之后面解决它。你或许想看一下它-最后我几乎没有处理它，但它不不是真的很难。然而没有一个机器图来证明，对我来说确实有点迷茫。

另一个要注意的有趣的事情，那些在开始时分配到堆栈和虚拟内存的应用区域的初始值让我有点困惑。这个应用程序在0x8000000开始，这看起来是个“奇怪”的数值。我的灵光在这里没有出现，于是我只能用其它方法来寻找答案。

到处逆向一番后，我来到了下面代码段：

```
.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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.text:0040128B 000      mov eax, 1
.text:00401290 000      mov [edx], ecx
.text:00401292 000      retn ; Return Near from Procedure
```

如果你仔细看，你会发现，这段代码完成小端字节（Little Endian）和大端字节(Big Endian)的互换工作(译者加：Big-Endian一个Word中的高位的Byte放在内存中这个Word区域的低地址处；Little-Endian一个Word中的低位的Byte放在内存中这个Word区域的低地址处)。一旦我们找到并理解了这里的代码，我马上就想通了0x80000000的由来。它只是个1，使用的是大端字节表示法。我非常郁闷竟然没能在一开始就看出它来-毕竟人无完人嘛（译者加：呵呵）。

一个用来来理解指令中数值使用的好方法，是使用跳转指令做参考：你会在那里找到寻址模式：如果你检查VM_JMP指令，你会注意到有一个参数被检测，如果成功，会被加到当前VM_EIP上。这听起来象一个偏移跳转，不是吗？我们来看一下：

```
.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
```

通过那个比较检测，我们会发现地址模式代码-和它在参数类型中的相对位置。

嗯…在今天调整预编译头需要很多时间(“/£%#$!!)，因此我们可以更进一步了。

到这里，为了能确定更多的域，看一下VM解码器会很有用，我们可以更进一步分析这个虚拟机的两个重要函数，及从虚拟指令参数中读写数据的过程。

虚拟机主体

一个虚拟机通常在构建在一个虚拟环境里，这个虚拟环境（CONTEXT）是一个放置机器寄存器和参数的空间。T206也不例外：下面是这个虚拟机使用的内存空间：

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
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
}
```

这些段中某些是虚拟机器特有的，我们也可以在其中“看”到常见的段：一套专用和通用寄存器，执行指针和堆栈指针（是的，就是我们的EIP和ESP），机器标志位，和其它标志位-堆栈空间和内存空间的地址也在其中。最后一个段比较有趣-ResumeExec，这个域被用作“异常句柄”，它也可以在调试时使用（在T206中几乎所有的调试代码被移除，但你可以通过一些留下来的东东来恢复它。例如“明显”的陷阱标志检查）。

在前面的那几个寄存器，之所以那样命名，是因为它们是用作IO目的的。当然，那并不是他们的唯一用法-是他们的特殊用法。一旦IO被VM_ALLOW_IO指令允许使用（已经列出了），他们就会得到地址并被使用。

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


PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        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;
}
```

我如何能假定这些寄存器用另一种方式使用？当然，你可以想一下，有些东西必须“赋”值，不是吗？这很好，理由仍需要反着思考一下。TVMContext.Registers[]指针域在TVMContext结构中被初始化。这意味着TVMContext.Registers[]通用指令参考寄存器可以随意通过通用寄存器获得。

我们现在看一下这个虚拟机的主函数的细节，这样我们可以理解它是如何开始工作的：

```
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;
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
                        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....
```

让我们一点一点来解释他们（请注意我重新调整了一点代码的结构，因为我不喜欢在T206代码中的杂乱结构）

1) **初始化虚拟机**：这部分代码只是简单的分配内存，拷贝虚拟机程序和初始化开始数值，没有更多的内容了。
2) **开始虚拟机循环**：这是虚拟机的核心：在这里我们计数指令，我们测试特别条件（例如陷阱标志、IO请求，代码流异常）。在虚拟机循环中我们转换虚拟EIP到一个x86地址，同时我们在这个地址读取并解码这个虚拟指令。
3) **处理虚拟指令并执行他们**：如果VM_EIP转换和指令解码进行顺利，我们处理这个虚拟机指令，提供机器环境和保存解码指令数据的缓存给它。
4) **如果我们有一个机器检查在工作，确保捕捉“I/O”**：机器检查不会总是错误：因此检查它的特别条件如I/O请求。
5) **执行异常检查**：如果由于代码流或我们到达“异常检查”的错误，检测Resume_exec寄存器，如果为空，应用程序结束。

在我们研究虚拟机指令管理的细节以前，我们最好花些时间在虚拟机内存管理上。如前面所说，虚拟机内存是大端字节的方式。因此，在一个小端字节机器里，我们需要来回转换所有的数值。这个虚拟机使用两个函数来完成这些，Read_VMMemory_To()和Write_VMMemory_From()。让我们看一下它们的执行，来弄清楚他们是如何工作的：

```
.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]
```


PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.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;
}
```

正如我们所看到的，这个过程的代码仅仅是完成内存虚拟地址和它的x86地址的简单转换，然后写入一个数值，并调整其字节装载模式(大端/小端)。这个过程很明显是被用在那些执行如虚拟堆栈类的过程中。读的过程非常相似，不同的是写入的是x86内存，而不是虚拟内存。

虚拟机指令核心

虚拟机指令核心是指通过一个缓存，来解释VM的程序操作代码和将他们传递到虚拟机指令调度表中。从这个角度来看，你可以通过交换虚拟机解码器和保持一些相关域来写出你自己的虚拟机语言。这是可能的，因为事实上虚拟机解释被作为缓存中一层，它可以被看作是一个中间层。我们来仔细看一些这个缓存：

```
.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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.text:00401230 ; VM_PUSH+3D#p ...
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
};
```

这些结构经常在虚拟机中使用，我们再看一下读写参数的函数：

```
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) {
casevmaRegister:
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;
}
```

如我们所看到的，这个读取参数数值的过程要区分不同的地址类型，从虚拟寄存器设置中获得数据，或从虚拟地址空间的一个虚拟地址获得数据，或从一个立即数获得。任何时候它需要进入虚拟内存时，它使用了一个相应的内存读取函数，并返回一个小端数值。写参数

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

数值的函数处理过程非常相似。

到这里，只剩两个的主要函数需要看一下，一个用来处理虚拟标志，另一个是虚拟解码器自身。

在可能改变标志状态的函数中，调用了Evaluate_Flags()函数。例如VM_CMP和VM_SET指令调用这个函数来设定相应的VM标志位。这个标志处理函数也会在一些数学运算的函数之后调用，以保证相应的内部标志状态。这个函数中有趣的地方是，它接收了一个特殊参数用来表示是标志位应该如何检测：它被用在数学操作中，用来改变OF/CF标志。

```
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;
}
```

一个边注：你可以应用你自己的虚拟机指令来处理位域（BTS,BTC等），然后使用一个不同于常规大小（1-2-4）的操作数来调用这个函数。这将导致这个函数需要一个额外的参数用作“位掩码”，以对这些位进行Btx检测。

下面是解码函数：

```
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!
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
            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;
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
}
```

在开始部分，它简单的填充了指令的常规参数，之后它通过参数的解码结构的数值开始循环。这个结构包含指令的参数用法，及特定指令的子类型-看上去似乎和在JCC的标志使用相关。开始我把这些域称为“JCC类型”，之后我把它改为更通用的“Instr类型”，因为理论上有一个更通用的用法，就像在代码中展现出来的那样。

参数的主循环通过一个“ TParamDecoding.Params[]”数组中运行。零表示没有更多的参数。根据参数的地址类型，它取出正确的数据并填充TinstructionBuffer结构。TparamDecoding数组开始于“.data:00407030 ParamTable”，并在虚拟机指令表之前结束。

虚拟机指令

虚拟机指令序列包含最常用的x86指令，和很少的几个新的指令。下面就是应用在及其中的虚拟机操作数里列表：

```
.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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.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
```

除了看一些指令的应用外，没什么好说的了。

其中一个有趣的指令如下：

```
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;
}
```

正象你看到的，这个指令找回一个参数数值-完美的与ECX等效-和它一样递减，当它减为0时，它中止并跳到下一条指令，就像LOOPCXZ一样，或它把这个递减数值写入源操作数/寄存器和执行一个跳转到制定地址，依照地址类型不同这个地址可以是相对地址也可以是绝对地址。

另一条有趣的指令如下：

```
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);
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
return 0;
}
```

正如你所见到的，这个指令读子函数的地址，设置VM_EIP以在执行下一个指令时从这里开始，然后保存下一个指令的VM_EIP数值（在VM_CALL之后的那条）到堆栈中，写在VM_ESP处。正象之前讲述的，堆栈地址模式被交换过。

还有很多其它指令应该看一看，你可以自己去附录中的源代码，如果你希望找到一些特别的东西。

总结

这个虚拟机中还缺少一些东西，最明显的就是缺少位操作指令和调试层支持。

好了，估计我已经让你很痛苦了，我也写的很痛苦。希望从中你能学到一些关于虚拟机的知识，你会发现这篇小文会对你所需要的任何东西都有所帮助。

附录 A

这里你可以找到T206机器的几乎全部逆向源代码。有的部分丢失了（完整性检查确认了这一点），它不会像原来那样编译。然而，再花少许努力，它就可以被用来编码你自己的虚拟机，也能在T206学到更多的些节。

希望对你有所帮助。

```
/*#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,
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
.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) {
                casevmaRegister:
                switch(InstrBuff->OperandSize){
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
                        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) {
                casevmaRegister:
                        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;
                casevmaDirectAddress:
                        vmaddr = vmAddress;
                        casevmaRegisterAddres:
                        if (Param->AddressType!=vmaDirectAddress) {
                                vmaddr =VMContext->Registers[Param->RegisterIdx];
                        }
                        res =Write_VMMemory_From(vmaddr,&ValueToWriteTo,InstrBuff->OperandSize,VMContext)
                        if (!res) return 0;
                        break;
                casevmaVMValue:
                        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;
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
                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)
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
{
        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,VMContext);
        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;
        }
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        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) {
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
                        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 3: // IDIVREST - copy&paste above.
                break;
                default: // opcode 12 and follows
        }
        Evaluate_Flags(VMValueSrc,VMValueEval,ModeSwitch,InstructionPtr,VmContext);
        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);
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        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)
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
                                        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
```

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        // 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....
};
```

风暴译 2007-10-4 QQ:719110750

WW
PDF created with pdfFactory Pro trial version www.pdffactory.com