FinSpy VM Unpacking Tutorial Part 3: Devirtualization. Phase #1: Deobfuscating FinSpy VM Bytecode Programs

msreverseengineering.com/blog/2018/2/21/wsbjxrs1jjw7qi4trk9t3qy6hr7dye

February 21, 2018

February 21, 2018 Rolf Rolles

[Note: if you've been linked here without context, the introduction to Part #3 describing its four phases can be found here.]

1. Introduction

In part one of this series, we analyzed the obfuscation on the x86 implementation of the FinSpy VM, and wrote a tool to deobfuscate it to allow easier analysis. In the second part of this series, we analyzed the VM instruction set, wrote a disassembler that works for the running example, and obtained the VM bytecode listing. Now we are left with devirtualization: we want to regenerate a facsimile of the original x86 program prior to virtualization.

This task winds up being fairly lengthy, so we have divided the approach into four phases, which corresponds to the work I did in devirtualizing FinSpy in the same order in which I did it. (That being said, don't mistake "lengthy" for "difficult".) We begin Phase #1 by inspecting the FinSpy VM bytecode program, discovering obfuscation involving the "Group #2" instructions, and removing it via pattern-substitution.

2. Devirtualizing FinSpy: Initial Observations and Strategy

At this point in the process, I had my first listing of the VM bytecode program in disassembled form. I began by inspecting the VM bytecode program and identifying ways to simplify the bytecode. This ultimately resulted in the wholesale elimination of the Group #2 instructions.

You may wish to inspect the initial VM bytecode disassembly, which we shall progressively refine throughout this Phase #1. Along the way, after discovering and applying simplifications, we will gradually obtain new VM bytecode disassembly listings, which will be linked throughout.

2.1 Recap of FinSpy VM Instruction Set and Bytecode Programs

To summarize part two, the FinSpy VM uses a fixed-length instruction encoding, with each VM instruction represented by a structure that is 0x18 bytes in length.

Each instruction within a FinSpy VM program has two uniquely identifying characteristics associated with it. For one, we can refer to the raw position of the VM instruction within the array of VM bytecode instructions. So for example, the first instruction is located at position 0x0. Since each instruction is 0x18 bytes in length, the second instruction is located at position 0x18, and generally instruction #N is located at position 0x18*N within the bytecode array.

The second identifying characteristic for an instruction is its "key", a 32-bit value (the first DWORD in an encoded FinSpy VM instruction) that can be used to locate a particular VM instruction. Specifically, before entering the VM, the x86 code that executes before transferring control to the FinSpy VM interpreter first pushes a key DWORD onto the stack. Upon entering the VM, the FinSpy VM initalization code loads the key DWORD pushed by the x86 portion, searches through the VM bytecode array to locate the VM instruction with that key, and then begins interpreting VM instructions starting from that location. (It turns out that FinSpy VM instruction keys cause some complications in devirtualization, as we shall see later.)

Most FinSpy VM instructions are assumed to execute sequentially. That is, after one VM instruction completes, the x86 code for the instruction handler adds 0x18 to the current VM EIP, to advance it to the next VM instruction. VM control flow instructions may behave differently, e.g., the conditional branch instructions are relatively-addressed like their x86 counterparts (if the branch is taken, the VM instruction adds a displacement, a multiple of 0x18, to VM EIP; if the branch is not taken, execution continues at the VM instruction 0x18 bytes subsequent to the current instruction). The FinSpy VM also has an instruction for direct calls, and another one for indirect calls. These call instructions behave like you'd expect; they push a return address onto the stack to re-enter the VM after the called function returns.

The FinSpy VM's instruction set consists of three instruction groups:

- Group #1: Conditional and unconditional jumps, (namely, JMP and all 16 of x86's conditional branches such as JZ, JNS, and JP).
- Group #2: VM instructions that access the FinSpy VM's single dedicated register.
- Group #3: VM instructions that contain raw x86 instructions inside of them.

2.2 Preliminary Thoughts on Devirtualization

After analyzing the VM's instruction set and perusing the VM bytecode program, it seemed like the group #3 VM instructions -- those with embedded x86 machine code blobs -- would be easy to convert back into x86. There are three VM instructions in this group: "Raw X86", "Call Direct", and "Call Indirect". (In practice, the latter two VM instructions wound up presenting more complications than anticipated, with direct calls being the most difficult aspect of the VM to devirtualize.)

Group #1, the conditional and unconditional branch instructions, seemed similarly easy to translate back to x86, since their implementations are virtually identical to how x86 conditional branches operate internally. Since conditional branches use relative displacements to determine the address of the "taken" branch, the only challenge is to determine the relative locations of each devirtualized instruction. Thus, once we know exactly how far the branch instruction is from its destination, we can simply compute the displacement. (Indeed, in practice, this was easy.)

Group #2, the set of VM instructions that access the FinSpy VM's single dedicated register, was the wild-card. Although from analyzing the FinSpy VM harness I knew the raw functionality of these instructions, I did not know before analyzing the VM program how these instructions would be used, and nor how to devirtualize them. For ease of reference, those instructions are summarized below from their more detailed expositions in part two:

```
mov scratch, 0                     [Operand: none]
mov scratch, imm32                 [Operand: imm32]
shl scratch, imm32                 [Operand: imm32]
add scratch, imm32                 [Operand: imm32]
mov scratch, savedReg32            [Operand: index of savedReg32]
add scratch, savedReg32            [Operand: index of savedReg32]
mov savedReg32, scratch            [Operand: index of savedReg32]
mov dword ptr [scratch], savedReg32 [Operand: index of savedReg32]
mov scratch, dword ptr [scratch]   [Operand: none]
mov dword ptr [scratch], imm32     [Operand: imm32]
mov dword ptr [imm32], scratch     [Operand: imm32]
push scratch                       [Operand: none]
```

Given that the other two groups seem straightforward, and the heretofore-unknown nature of Group #2, this seems like a good first task to devirtualize our FinSpy VM program: analyze the usage patterns of group #2 instructions, and formulate a strategy for converting them back to x86 machine code. (This turned out to be very easy and intuitive in practice, with no real complications to speak of.)

3. Step #1: Tweak the Output to Print the x86 Disassembly

Upon perusing the output of my FinSpy VM disassembler, my first observation was that my task would benefit from knowing the textual disassembly for the "Raw x86" instructions, which were being rendered as raw machine code bytes. For example, the first few instructions in the VM bytecode disassembly were:

```
0x000000: MOV SCRATCH, 0
0x000018: ADD SCRATCH, EBP
0x000030: ADD SCRATCH, 0x000008
0x000048: MOV SCRATCH, DWORD PTR [SCRATCH]
0x000060: MOV EAX, SCRATCH
0x000078: X86 [199, 0, 215, 1, 0, 0] ; <- x86 machine code
0x000090: X86 [184, 0, 240, 65, 0]   ; <- would prefer x86 assembly
0x0000a8: X86 [201]
0x0000c0: X86 [194, 4, 0]
```

My proximate goal was to figure out how the Group #2 instructions (those involving the SCRATCH register, the first five in the listing above) were used within the VM program. For context, it would be nice to see what the X86 instructions on the four subsequent lines were doing. I.e., it would be convenient to have a textual representation for the x86 machine code, instead of the above display as lists of bytes.

For this task, I was fortunate to have already written an x86 disassembler and assembler library in Python. (I have previously released this code to the public as part of the course sample for my SMT-Based Program Analysis training class.) For the two group #3 VM instructions with embedded x86 machine code -- "Raw X86" and "Call Indirect" -- I modified my FinSpy VM disassembler to invoke the x86 disassembler functionality when printing them. This was trivial to accomplish in the __str__() method for these Python classes, for example:

```
        d = X86Decoder(self.Remainder[0:self.DataLen])
        i2container = d.Decode(0)
```

After doing this, the VM bytecode disassembly listing was easier to follow. An example is shown below; the full disassembly listing can be found here.

```
0x000000: MOV SCRATCH, 0
0x000018: ADD SCRATCH, EBP
0x000030: ADD SCRATCH, 0x000008
0x000048: MOV SCRATCH, DWORD PTR [SCRATCH]
0x000060: MOV EAX, SCRATCH
0x000078: X86 mov dword ptr [eax], 1D7h ; <- new: x86 assembly, not machine code
0x000090: X86 mov eax, 41F000h
0x0000a8: X86 leave
0x0000c0: X86 ret 4h
```

After some quick perusal, the x86 assembly language mostly looks like ordinary C code compiled with MSVC, though some functions use instructions like PUSHA and POPA that are not ordinarily generated by normal compilers, and hence were probably written in assembly language prior to virtualization.

4. Step #2: Pattern-Based Simplification for Scratch Register Access

Having a clearer FinSpy VM bytecode disassembly listing, I returned to trying to figure out what the Group #2 "dedicated register" instructions were accomplishing. It's easy enough to locate these instructions; all of them use the scratch register, which is denoted "SCRATCH" in the VM bytecode disassembly, so you can simply search for "SCRATCH" to see where these instructions are used. (Or just look at the disassembly listing -- group #2 instructions comprise a significant fraction of the VM instructions for our sample.)

Here I went a bit too fast and would have benefited from being a bit more deliberate. Inspecting the beginning of the VM bytecode program, I noticed a few different patterns involving loading x86 register values into the scratch register. For example, repeating the first two lines from the example in the last section, we see:

```
0x000000: MOV SCRATCH, 0
0x000018: ADD SCRATCH, EBP
```

This code first sets the scratch register to zero, and then adds the value of EBP. Effectively, this two-instruction sequence sets the scratch register to EBP. However, elsewhere there were different patterns of VM instructions used to accomplish the same goal. Here was the second pattern:

```
0x03c270: MOV SCRATCH, 0
0x03c288: MOV SCRATCH, EDX
```

And still elsewhere in the FinSpy VM bytecode disassembly, we see more natural one-instruction sequences to load register values into the stack register, for example:

```
0x001b48: MOV SCRATCH, ECX
```

Faced with these observations, I mistakenly identified this as a form of obfuscation: it seemed as though there were several ways to load the value of an x86 register into the scratch register, and that FinSpy was choosing between them arbitrarily in order to introduce some randomness into the VM bytecode program. Such randomness could make pattern-recognition more laborious. It might end up that I would need to write multiple patterns to match these sequences, thereby multiplying the number of patterns I needed to write involving x86 register loads by the number of patterns that the FinSpy VM employed for loading an x86 register into the SCRATCH register.

I figured that I could save work for myself down the road if I coalesced the two-instruction sequences down into a canonical representation of loading a register into the stack, namely, one instruction sequences as in the third example. (I later realized that this isn't really a form of obfuscation -- the FinSpy VM developers were just being lazy.)

To be specific, I wanted to write a search-and-replace rule that would take VM instruction sequences of the form:

```
MOV SCRATCH, 0      | MOV SCRATCH, 0
ADD SCRATCH, REG32  | MOV SCRATCH, REG32
```

And replace them with one-instruction sequences of the form:

```
MOV SCRATCH, REG32
```

It was easy enough to accomplish the pattern recognition and replacement using Python's type introspection functionality. Since my Python FinSpy VM disassembler from part two used Python class types for these individual VM bytecode instruction types, I could simply invoke the "isinstance" function to find instances of these patterns in the VM bytecode disassembly listing. The code for this can be found in the function "FirstSimplify" (search the code for that name to find the exact point of definition).

For instance, here is the code to identify the "MOV SCRATCH, 0" VM instruction. It encompasses two different possibilities: the dedicated VM instruction that moves 0 into SCRATCH, and the VM instruction that moves an arbitrary DWORD value into SCRATCH, when the DWORD is zero.

```
def IsMovScratch0(insn):
    if isinstance(insn,MovScratch0):
        return True
    if isinstance(insn,MovScratchImm32):
        return insn.Imm32 == 0
    return False
```

The complete code for recognition and replacement is shown in the following sequence. Please take the time to examine it, as all of the pattern-matching and replacement in the rest of this section have similar implementations, and won't be discussed in much detail for the sake of brevity.

```
# If the first VM instruction is "MOV SCRATCH, 0"...
if IsMovScratch0(i1):

    # Check to see if the second instruction is "MOV SCRATCH, REG32". If
    # this function returns None, it wasn't. If it returns something other
    # than None, then it returns the register number.
    mr = IsMovScratchReg(i2)

    # Otherwise, check to see if the instruction was "ADD SCRACH, REG32",
    # and get the register number if it was (or None if it wasn't).
    if mr is None:
        mr = IsAddScratchReg(i2)

    # Did one of the two patterns match?
    if mr is not None:

        # Yes: make a new VM instruction, namely "MOV SCRATCH, REG32" to
        # replace the two instruction sequence we just matched. Use the same
        # file offset position from the first instruction in the sequence.
        newInsn = MovScratchDisp8([0]*INSN_DESC_SIZE, i1.Pos)

        # Save the register number into the new instruction.
        newInsn.Disp8 = mr

        # Use the same VM instruction key as for the first instruction.
        newInsn.Key = i1.Key

        # Add the new VM instruction to the output list.
        newInsnArr.append(newInsn)
```

If one of the two patterns match, the code above generates a new VM instruction to replace the existing two. Since each VM instruction is uniquely associated with a position and key, we decided to copy those attributes from the first VM instruction within the matched pattern, and to delete the second VM instruction outright. (I worried that I might encounter situations where other VM instructions would reference the second and now-deleted VM instruction by its key or EIP, and in fact I had to account for this in Part #3, Phase #4.)

The new VM bytecode disassembly listing can be found here.

(Note that I originally wrote these pattern replacements in OCaml, since ML-family languages are intentionally designed with robust facilities for these sorts of tasks. My first implementation had the Python program print out an OCaml representation of the FinSpy VM program, which an OCaml program then simplified via pattern-substitutions. However, given that only a few such pattern-replacements were necessary to deobfuscate FinSpy VM bytecode programs, I decided that the cognitive and programmatic overhead of serializing between the OCaml and Python programs was unnecessarily complicated, and so I re-wrote the pattern replacement code in pure Python, despite the tedium.)

4.1 The "MOV SCRATCH, REG32" / "PUSH SCRATCH" Pattern

Next, I continued to inspect the VM bytecode disassembly listing and look for further patterns involving the SCRATCH register. I found my next candidate shortly after the previous one. In particular:

```
0x000120: MOV SCRATCH, ECX
0x000138: PUSH SCRATCH
```

This VM instruction sequence obviously replaces the x86 instruction "push ecx". As with the previous step, I decided to codify this into a pattern simplification. Any time we see the VM instruction sequence:

```
MOV SCRATCH, REG32
PUSH SCRATCH
```

We want to replace that with a single VM instruction representing "push reg32". Mechanically, this works nearly identically to the previous step; we use Python's isinstance() function to find occurrences of this VM instruction pattern. The complete code can be found in the "SecondSimplify" function in the Python source code (search for the function name). For ease of reference, here is how we identify "MOV SCRATCH, REG32" instructions:

```
def IsMovScratchReg(insn):
    if isinstance(insn,MovScratchDisp32):
        return insn.Disp32
    if isinstance(insn,MovScratchDisp8):
        return insn.Disp8
    return None
```

To replace these two-instruction sequences, we generate a FinSpy VM "Raw x86" instruction which contains the x86 machine code for the x86 PUSH operation. I use my x86 library to create the Python object for the replacement x86 instruction "push reg32": in this example, the Python object for "push ecx" can be constructed as X86.Instruction([], XM.Push, X86.Gd(mr,True)).

After generating the replacement x86 instruction object, I wrote a function to generate a FinSpy VM instruction containing the raw machine code. The function "MakeRawX86" from the simplification source code (search for the function name) is reproduced here for ease of presentation:

```
def MakeRawX86(Pos, Key, x86Insn):
    # Create a FinSpy VM "Raw X86" instruction with dummy
    # content at the correct position (specified by Pos)
    newInsn = RawX86StraightLine([0]*INSN_DESC_SIZE, Pos)

    # Set the key to be the key from the first of the two
    # instructions of the two-instruction matched sequence
    newInsn.Key = Key

    # Encode the x86 instruction into machine code, store
    # the bytes in the FinSpy VM instruction
    newInsn.Remainder = EncodeInstruction(x86Insn)

    # Cache the length of the x86 instruction's machine code
    newInsn.DataLen = len(newInsn.Remainder)

    # Cache the textual disassembly for that instruction
    newInsn.X86 = str(x86Insn)

    # Return the FinSpy VM instruction just constructed
    return newInsn
```

The FinSpy VM "Raw X86" instruction object returned by the function above is used as the replacement for the two-VM instruction sequence "MOV SCRATCH, REG32 / PUSH SCRATCH". As the code shows, its two uniquely-identifying characteristics (VM instruction position and VM instruction key) are duplicated from the first of the two VM instructions in the pattern.

After applying this substitution, the new VM bytecode disassembly listing can be found here.

4.2 The "MOV SCRATCH, REG32" / "MOV REG32, SCRATCH" Pattern

Shortly thereafter was another, similar VM instruction pattern:

```
0x000228: MOV SCRATCH, ESP
0x000240: MOV ESI, SCRATCH
```

Clearly, this sequence virtualizes the x86 instruction "mov esi, esp". More generally, when we see two adjacent VM instructions of the form:

```
MOV SCRATCH, REG32_1
MOV REG32_2, SCRATCH
```

We can replace this with an x86 instruction "mov reg32_2, reg32_1". As in the previous case, after using Python isinstance() checks to locate instances of this pattern, we generate a Python object representing the x86 instruction "mov esi, esp", and call the MakeRawX86 function (detailed in the previous section) to generate a new "Raw x86" VM instruction to replace the two VM instructions comprising the pattern instance. This process can be seen completely in the function "ThirdSimplify" from the simplification code (search for the function name).

After applying this substitution, the new VM bytecode disassembly listing can be found here.

4.3 The "MOV SCRATCH, 0" / "ADD SCRATCH, IMM32" Pattern

Continuing to look for patterns in the VM bytecode program, I saw the following:

```
0x0037b0: MOV SCRATCH, 0
0x0037c8: ADD SCRATCH, 0x420344
```

This sequence can obviously be replaced with the single-instruction sequence "MOV SCRATCH, 0x420344". The function "FourthSimplify" in the simplification code (search for the function name) implements the detection and substitution. Again, it's trivial to write this pattern-recognition and replacement code, though unlike the previously-described pattern-substitutions, the replacement code is a Group #2 "SCRATCH register" FinSpy VM instruction rather than a Group #3 "Raw x86" instruction, since the output involves the scratch register.

After applying this substitution, the new VM bytecode disassembly listing can be found here.

4.4 The "MOV SCRATCH, IMM32" / "PUSH SCRATCH" Pattern

More perusal of the VM bytecode disassembly turned up the following sequence:

```
0x006900: MOV SCRATCH, 0x000003
0x006918: PUSH SCRATCH
```

As with previous examples, we can replace this two-instruction sequence with the x86 instruction "push 3", by using a "Raw x86" VM instruction. The function "FifthSimplify" implements this -- it identifies "MOV SCRATCH, IMM32 / PUSH SCRATCH" sequences, and replaces them with a "Raw x86" FinSpy VM instruction object containing the machine code for an x86 "push imm32" instruction, again using identical techniques to those discussed previously.

After applying this substitution, the new VM bytecode disassembly listing can be found here.

4.5 Memory Address Patterns

After performing the aforementioned replacements, we start to see a lot less variety in the remaining Group #2 instructions, and their further purpose quickly became obvious. Namely, each remaining cluster of VM bytecode instructions using the SCRATCH register consisted of two consecutive components: 1) a sequence of FinSpy VM instructions to load a memory address into the SCRATCH register, using what I've called a "memory address pattern"; followed by 2) a sequence of FinSpy VM instructions utilizing using the memory address just generated, for a purpose such as reading from the address or writing to it, using what I've termed a "memory access pattern". We will discuss both such patterns in this section and the subsequent one.

For an example of both a memory address pattern and a memory access pattern, here is the very beginning of the VM bytecode program after all the prior replacements have executed:

```
0x000000: MOV SCRATCH, EBP                  ; Memory !address!
0x000030: ADD SCRATCH, 0x000008             ; Memory !address!
0x000048: MOV SCRATCH, DWORD PTR [SCRATCH] ; Memory !access!
0x000060: MOV EAX, SCRATCH                  ; Memory !access!
```

The first two VM instructions set the SCRATCH register to EBP+0x8. These two instructions comprise the "memory address pattern".

The last two VM instructions read a DWORD from the memory address contained in the SCRATCH register, and store the result into EAX. These two instructions comprise the "memory access pattern".

Clearly, these four VM instructions can be replaced by the single x86 instruction "mov eax, [ebp+8]" -- which looks very much like something we'd expect to see near the beginning of a function (suitably, given that the four VM instructions just shown are the first VM instructions in the bytecode program, and are situated toward the beginning of an x86 function).

For a more complicated example:

```
0x03c270: MOV SCRATCH, EDX      ; Memory !address!
0x03c2a0: SHL SCRATCH, 0x000003 ; Memory !address!
0x03c2b8: ADD SCRATCH, EAX      ; Memory !address!
0x03c2d0: ADD SCRATCH, 0x000010 ; Memory !address!
0x03c2e8: MOV EAX, SCRATCH      ; Memory !access!
```

Let's trace the value that is written to EAX at the end. The first two VM instructions shift EDX left by 3; the third VM instruction adds EAX, and the fourth VM instruction then adds 0x10. Thus, the expression written to EAX is "EDX<<3 + EAX + 0x10". EDX<<3 is the same thing as EDX*8, so our expression is "EDX*8 + EAX + 0x10". The format of the memory expression should look familiar -- it can be encoded as the legal x86 ModRM/32 memory expression "[EAX+EDX*8+0x10]". These first four VM instructions comprise a "memory address pattern". The fifth and final VM instruction is the "memory access pattern" instance

-- rather than reading from or writing to this memory location, we simply store the memory address itself into EAX. Thus this VM instruction sequence performs the equivalent of the x86 instruction "lea eax, [eax+edx*8+10h]".

Indeed, all of the memory address patterns appear to be creating memory addresses that would be legal to encode using X86 ModRM memory expressions. X86 ModRM memory expressions contain one or more of the following elements added together:

- A 32-bit base register
- A 32-bit scale register, optionally multiplied by 2, 4, or 8
- A 32-bit displacement (an immediate DWORD)

By inspecting the VM bytecode program, all memory address sequences have an identical layout.

- If an index register is being used, the first VM instruction of the sequence sets the SCRATCH register to the value of the base register.
- If the index register is multiplied by a scale factor, the next VM instruction is SHL SCRATCH, IMM32, where IMM32 is 1, 2, or 3 (to multiply the index register by *2, *4, or *8, respectively).
- If a base register is used, the next instruction in the sequence adds the base register.
- If a 32-bit displacement is used, the next VM instruction is an ADD SCRATCH, IMM32 instruction.

All of these elements are optional, though at least one must be present. So for example, if the memory address is a raw 32-bit value (e.g. dword ptr [401234h]), only the ADD SCRATCH, IMM32 VM instruction will be present. If the memory expression consists of only a register (e.g., dword ptr [eax]), then only the MOV SCRATCH, REG32 VM instruction will be present. Memory expressions that use more than one element will be virtualized by combining the VM instructions for the elements present.

4.6 Memory Access Patterns

After a memory address pattern just described creates a memory expression in the SCRATCH register, the next one or two VM instructions make up the memory access pattern, and dictate how the memory address is used through how they access the SCRATCH register. By examining the VM bytecode program, I found that there were four distinct VM instruction sequences used for memory access patterns.

4.6.1 Memory Access Case #1: Memory Read, Store Into Register

The first case reads from the memory address, and stores the result in a register:

```
; ... memory address pattern before this ...
0x000048: MOV SCRATCH, DWORD PTR [SCRATCH]
0x000060: MOV EAX, SCRATCH
```

This corresponds to "mov eax, dword ptr [memExpression]", with the details of "memExpression" being dictated by the memory address pattern.

4.6.2 Memory Access Case #2: Memory Read, Push Result

The second case reads from the memory address, and then pushes the result on the stack:

```
; ... memory address pattern before this ...
0x004950: MOV SCRATCH, DWORD PTR [SCRATCH]
0x004968: PUSH SCRATCH
```

This corresponds to "push dword ptr [memExpression]", with the details of "memExpression" being dictated by the memory address pattern.

4.6.3 Memory Access Case #3: Memory Write, Value Taken from Register

The third case stores a register into the memory location calculated by the memory address sequence:

```
; ... memory address pattern before this ...
0x000ae0: MOV DWORD PTR [SCRATCH], EAX
```

This corresponds to "mov dword ptr [memExpression], eax", with the details of "memExpression" being dictated by the memory address pattern.

4.6.4 Memory Access Case #4: Store Address into Register

The fourth case does not dereference the memory location, but rather saves the result into a register:

```
; ... memory address pattern before this ...
0x000e28: MOV EAX, SCRATCH
```

This corresponds to "lea eax, dword ptr [memExpression]", with the details of "memExpression" being dictated by the memory address pattern.

4.7 All Together: De-Virtualizing Memory References

After having fully analyzed the remaining references to the SCRATCH register as described above, I wrote code to detect the memory address patterns and the following memory access patterns, analyze them, and convert them back into x86 machine code. The function "SixthSimplify" in the simplification code recognizes memory address patterns, inspects the following instructions to determine the memory access patterns, and combines these two pieces of information to determine which x86 instruction the two memory address/access sequences have replaced. Next, the code reconstructs an x86 instruction -- as a FinSpy VM "Raw X86" instruction -- with which it finally replaces the VM instructions comprising the memory address and memory access sequences. As before, the replacement "Raw x86" FinSpy VM instruction uses the key and position attributes from the first VM instruction as the unique attributes for the replacement instruction.

First, "SixthSimplify" invokes the function "DecodeAddressSequence". That function determines if a given position within a list of FinSpy VM instructions begins a memory address sequence. If not, the function returns None. If so, DecodeAddressSequence extracts details from the memory address sequence -- the base register, the index register and optional scale factor, and the 32-bit displacement -- and uses it to create a Mem32 object (the class used by my Python x86 library to represent 32-bit memory expressions), which it then returns along with the number of VM instructions comprising the memory address pattern. That code is shown below.

```
# Given:
# * A list of FinSpy VM instructions in insnArr
# * An index within that list in idx
#
# Determine if insnArr[idx] contains a memory address sequence.
# If not, return None. If so, create an x86 memory expression
# operand using my x86 library, and return it along with the
# number of instructions in the address sequence.

def DecodeAddressSequence(insnArr, idx):
    # Save position of index within insnArr
    oldIdx = idx

    # The first VM instruction in the sequence is usually "MOV SCRATCH, REG32".
    r1 = IsMovScratchReg(insnArr[idx])

    # Was it?
    if r1 is not None:

        # Yes, it was, so increment the current index
        idx += 1

        # Is the next VM instruction "SHL REG, [1/2/3]"?
        if isinstance(insnArr[idx],ShlScratchImm32):
            # Yes, copy the scale factor
            scaleFac = insnArr[idx].Imm32
            assert(scaleFac == 1 or scaleFac == 2 or scaleFac == 3)
            # Increment the current index
            idx += 1

        # Otherwise, there is no scale factor
        else:
            scaleFac = None

        # Is the next VM instruction "ADD SCRATCH, REG32"?
        r2 = IsAddScratchReg(insnArr[idx])
        if r2 is not None:
            # Yes, increment the current index
            idx += 1

        # Is the next VM instruction "ADD SCRATCH, IMM32"?
        disp32 = IsAddScratchImm32(insnArr[idx])
        if disp32 is not None:
            # Yes, increment the current index
            idx += 1

        # Make a memory expression from the parts, and return the length
        # of the memory address sequence
        return (idx-oldIdx, MakeMemExpr(r2, r1, scaleFac, disp32))

    # The second possibility is that the memory expression is a raw address.
    imm = IsMovScratchImm32(insnArr[idx])
    # Was it?
    if imm is not None:
        # Yes: make a memory expression from the address, and return the
        # length of the memory address sequence (namely, 1).
        return (1, MakeMemExpr(None,None,None,imm))

    # If we are here, neither of the memory address patterns matched, so
    # signal match failure.
    return None
```

Next, a similar function called "DecodeAccessSequence" uses pattern-matching to determine which of the four memory access patterns -- mov reg32, [memExpr] / push [memExpr] / mov [memExpr], reg32 / lea reg32, [memExpr] -- follows the access sequence. The code is similar to the code just shown for decoding memory address sequences, and so is not duplicated here. See the source code for the complete details.

Finally, after recognizing a memory address sequence followed by a memory access sequence, we use the machinery we previously developed for regenerating x86 machine code -- namely, the function "MakeRawX86" -- to create a FinSpy VM instruction containing the machine code for the raw x86 instruction. We use this to replace the memory address and access sequences just recognized.

After running this simplification pass, and all prior ones described for Group #2 instructions, over the VM bytecode program, all Group #2 VM instructions disappear, and consequently the SCRATCH register disappears from the VM bytecode disassembly listing. The final disassembly listing of the simplified FinSpy VM program can be seen in its entirety here.

5. Step #3: Correcting a Small Error

Having dispensed with the Group #2 VM instructions, Groups #1 and #3 remained. Before addressing them, however, there was one VM instruction that didn't fit into either one of those groups. Namely, in producing my initial VM bytecode disassembly listing, I had incorrectly identified the FinSpy VM unconditional jump instruction as being an instruction to cause a deliberate crash. I disassembled this as "CRASH". My FinSpy VM bytecode disassembly listing was confusing with how frequently the CRASH instruction occurred:

```
0x005bb0: JNZ VM[0x005c88] (fallthrough VM[0x005bc8])
0x005bc8: X86 xor eax, eax
0x005be0: CRASH
0x005bf8: X86 mov eax, dword ptr [esi+4h]
0x005c70: CRASH
0x005c88: X86 push ebx
```

I discussed this issue at length in part two -- the implementation for unconditional jumps was considerably different from conditional jumps, and involved dynamic code generation, which was only used otherwise for Group #3 instructions. Closer analysis revealed that the true purpose of this instruction was as an unconditional jump and not a deliberate crash. Upon learning this, I updated my disassembler component to rename the "CRASH" instruction to "JMP", and to print the branch targets of these newly-renamed JMP instructions. The resulting VM bytecode disassembly listing was clearer:

```
0x005bb0: JNZ VM[0x005c88] (fallthrough VM[0x005bc8])
0x005bc8: X86 xor eax, eax
0x005be0: JMP VM[0x008df0]
0x005bf8: X86 mov eax, dword ptr [esi+4h]
0x005c70: JMP VM[0x008df0]
0x005c88: X86 push ebx
```

The entire VM bytecode disassembly listing after this modification can be found here.

6. Conclusion

In this Phase #1, we inspected our sample's initial FinSpy VM bytecode disassembly listing. We improved the listing by including x86 disassembly for the x86 machine code embedded in the Group #3 instructions. We inspected how the VM bytecode program used Group #2 instructions. A half-dozen simple pattern-replacement schema were sufficient to entirely remove the Group #2 instructions. Afterwards, we corrected the output by observing that the "CRASH" instructions were really unconditional jumps.

In the next Part #3, Phase #2, we are ready to attempt our first devirtualization for the FinSpy VM bytecode program. Our initial attempts wind up being insufficient. We study the deficiencies in Part #3, Phase #3, and remedy them in our second, final, devirtualization attempt in Part #3, Phase #4.