AUTOMATED STATIC ANALYSIS OF VIRTUAL-MACHINE PACKERS

BY

JOSEPH KWUN LEONG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Assistant Professor Matthew Caesar

Abstract

The ability to reverse the most advanced software protection schemes is a critical step in mitigating malicious code attacks. Unfortunately, the analyst side seems to be losing in the ongoing arms race between malware developers and reverse engineers. Obfuscation that takes advantage of a virtual-machine like architecture has proven to be one of the most difficult to deal with. Virtual-machine packers are able to hide the intentions of programs they are applied to and are resistant to formerly effective unpacking techniques. Others have proposed methods to deal with such complex protections, but they are often tedious, expensive, and/or inflexible. We propose a novel approach to automate the analysis process of virtualization protected executables. Our design avoids many pitfalls and performance issues of dynamic-analysis systems by only employing static program-analysis techniques and emphasizing work-reuse and generality in order to maintain efficiency, flexibility, and accessibility, for even novice analysts. The proof-of-concept system we have developed shows promise for the future of virtual-machine protected software analysis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software packers, obfuscators and protectors are most prevalent in malware: viruses, worms, trojans, etc., but can also be used in commercial software. In either application, their goal is to prevent, or at least hinder, analysis of the underlying code that they protect. In practice, they are almost exclusively used as a form of obfuscation for nefarious purposes. Estimates suggest that the percentage of malware that is packed has dramatically increased, over a short period of time, to upwards of 79% [8]. Furthermore, with new record-numbers of malware samples each year [6], the need for automation grows stronger. Being able to triage a large number of samples quickly is a great advantage. Monetary incentives have ensured advancements on both sides. However, as of late the analyst side seems to have stalled when it comes to certain more advanced incarnations of protection. Without means of reversing said types of protectors, it becomes extremely difficult to protect ourselves against malware, and any hope of tracking down the cyber-criminals whom are responsible is dramatically reduced.

In this paper we take the first step towards automatically unpacking VM-protected code through completely static methods. We take advantage of modern program analysis techniques such as constraint solving, along with common structures that are often present in existing VM implementations. The system automatically finds references to virtualized instructions and reverse engineers the dispatching code to find the opcodes of the virtualized instruction set and lift their handlers to an intermediate representation for further processing. These steps are crucial to completely reversing such a complex protection scheme and it eliminates much of the tedium of manually analyzing the same protected code. Furthermore, we are able to bypass many of the shortcomings of automated dynamic systems which are often incomplete and more easily evaded.

1.1 Thesis Overview

In Chapter 1 we present the overall problem and motivation of the paper, as well as some background knowledge needed for the sections that follow. Chapter 2 goes over related works. In Chapter 3 we discuss the design decisions and assumptions made in developing the system we propose as a solution, along with the advantages and trade-offs that come with them. Chapter 4 explains our overall approach to the problem. Next, Chapter 5 explains how the system was implemented. In Chapter 6 we evaluate the validity and performance of our system, and Chapter 7 discusses possible future work. In Chapter 8 we draw conclusions and end with any closing remarks. Appendix A is primarily a code listing for the VM-obfuscation we wrote for testing.

1.2 Packer Evolution

Packers are constantly evolving. At a high level, they simply take in an executable binary and output another executable binary that is now smaller and/or more difficult to analyze. The first generation of packers compressed or encrypted the code portion of the executable before re-adding it as a new section of the binary. The original code section was then replaced with a small unpacking stub, responsible for reversing the prior obfuscation step, at run-time. The actual algorithm used for obfuscation can vary greatly: anything from simple XOR schemes and complex custom ciphers, to off-the-shelf cryptographic or compression algorithms.

Malware analysts developed techniques that are fairly successful in dealing with the most basic forms of run-time protectors. The most widely accepted automated-solutions involve detecting when the original unpacked-code resides in memory and dumping it back to disk for further inspection [30, 32]. It is worth noting that this only works because the entire unobfuscated program is in memory at one time, so the trick is to detect when that occurs.

In response to such effective counter-measures, the opposition created several new types of advanced-packers. Such protectors might involve breaking the original program up into multiple processes, threads, and/or binaries, which can then communicate over various covert channels or even debug each other, to impede analysis. Others might inject malicious code into other process, or create hollow or shadow processes; some even have their own custom loading procedures (which the operating system is typically responsible for), destroy the import table, dynamically load external code sources, etc. However, this paper focuses on the virtualization packer, or *VM-Packer*.

1.3 Virtual-Machine Packers

Obfuscators taking advantage of virtualization technology are some of the most widely used, as well as, difficult to analyze, forms of protection available today. The underlying architecture shares themes common with virtual machines and interpreted programming languages. The initial code segment is first recompiled from the binary's native instruction set (ISA), most commonly x86, into another "target-ISA". This instruction set can vary greatly between implementations and can even change between different runs of the same protector. VM-packers can utilize practically any kind of ISA imaginable. This includes both RISC and CISC inspired schemes. It can be as complete as the author would like and/or is capable of producing, and may have components generated at random. Next, similar to the first generation of packer, the entry-point of the executable is replaced by an interpreter for the newly generated bytecode. This allows each VM-instruction to be processed individually, so the entirety of the unobfuscated code never resides in memory at the same time. Without the entirety of the original code in view at once, it cannot be dumped to disk for later analysis.

1.3.1 Common Terms and Architecture

The term *Virtual-Machine Packer* comes from the internal architecture's likeness to a virtual machine. While each packer can vary greatly, in practice many implementations share a common overall architecture and certain terms are often used when talking about them:

VM Context

The context is basically the VM's state which persists throughout the lifetime of the virtualized program. It is updated by executing virtualized instructions. Each implementation of of a Virtual-Machine Packer may be unique, but common types of information stored in a VM's context might include, a virtual instruction pointer or program counter (VIP/VPC), registers, flags, stack(s), etc. A VM's context usually is implemented as a simple structure, like a C struct, and will often mimic that of a real hardware-CPU.

Dispatcher

The dispatcher or dispatching code, has the job of parsing the raw virtualized bytecode and determining which virtual instruction is to be executed. It is responsible for interpreting the ISA encoding, translating VM-bytecode to native instructions, and typically is called at least once for every VM-instruction. This portion of code might also perform tasks such as advance the VIP/VPC.

Handlers

A VM-instruction handler refers to the native code which is called as a result of a specific virtual instruction being interpreted. It is primarily responsible for changing the state of the VM according to what virtual instruction is called. There is often, but not necessarily, a one-to-one correspondence between VM-instructions and handlers. Being able to determine the mapping of bytecode to handler is essential for reversing a virtualization protector.

Architecture

The packed bytecode-program is generally stored in a data segment of a portable executable (PE), the standard executable format of binaries. From there, the dispatcher code will read instruction-by-instruction, calling the appropriate instruction-handler and updating the VM-context accordingly throughout. Because the obfuscated-program is never unpacked in memory at once, this form of protection is immune to unpackers which try to dump memory images to disk when certain conditions are met. Furthermore, due to a single original-instruction being translated into several packed-instructions, it makes manual step-through analysis with a debugger very tedious. Please refer to Appendix A for an example implementation of a very basic VM, named *StackVM*, implemented in MASM assembler.

Chapter 2

Related Works

Many researchers have developed techniques to deal with malware packers. One of the first solutions was OllyBonE [10]. It is a plug-in for the popular debugger OllyDbg [11]. It uses a kernel driver to allow the user to select sections of memory to "Break ON Execute". The idea is to have a malware analyst mark the memory segment where the obfuscated code is to be unpacked. Then, when the control flow jumps to the original entry point (OEP), the debugger will break and allow the unobfuscated code to be dumped to disk.

PolyUnpack [32] compares a static model of the original executable to a dynamically generated model. The differences between the models are used to identify hidden code. Renovo [29] and Omniunpack [30] automatically unpack malware by executing the packed program, either on bare-metal hardware or in an emulator, while keeping track of memory-writes. When a memory location that was written to, is set to be executed, that code is then typically considered to be unpacked-code and is dumped to disk.

All of these techniques are effective against the first generation of packers, which unpack themselves into data segments and then execute the unobfuscated code from memory. However, they are relatively ineffective against the most advanced of protection technologies, such as VM-packers, where the unobfuscated code never resides in memory all at once.

The system we are proposing also leverages classic program analysis techniques, applying them specifically to the problem of VM-Packers. Automated theorem provers have been used for quite some time in the program analysis field [22]. Also, recently, Vanegue et al. have explored the use of SMT solvers for security purposes such as vulnerability checking and exploit generation [35]. Rolles has detailed input-crafting using symbolic execution and constraint-solvers to break cryptosystems statically [16]. Our system uses a constraint solver to glean information regarding VM-protections.

2.1 VM Obfuscators

Existing work on analyzing Virtual-Machine Packers can be split into two main categories based on the types of analyses used: static or dynamic.

Static reverse engineering

Static reverse engineering typically involves having a skilled specialist analyze the output of a disassembler, such as IDA Pro [5], and manually draw conclusions about how the particular VM-Packer works. This is an extremely time-consuming and expensive process. Since relatively few people

have the skills required to do such work, and the process itself is very tedious. This must also be repeated for each instance of packer. Rolles has published several works which detail some of the techniques and processes he uses to reverse industrial-grade protections, and how one might be able to use compiler theory and optimizations for deobfuscation. [31] provides a general method for reverse-engineering the whole family of protectors manually. [3] explains unpacking a custom packer using an IDA processor module. He also has a series of articles on reverse-engineering binaries packed with VMProtect [20], [12–14, 19]. These kinds of methods are the most prevalent in commercial applications and are the most reliable approach we have currently for dealing with the most sophisticated software protections.

Automated dynamic analysis

Automated dynamic analysis will often take advantage of traces, generally produced by using some kind of virtual environment or emulator to execute the binary in question under instrumentation. Sharif et al. took advantage of traces from a modified Qemu [15] build and used memory access patterns to automatically learn several features of malware emulators [33]. They were able to extract many structural details as well as information regarding the VM's instruction set through these methods. Coogan et al. also used traces. However, they took a semantics-based approach to deobfuscating instruction-traces, working backwards from system-calls [25]. Their goal was *observational-equivalence*, while also providing insight into the original unobfuscated-instructions.

Hybrid

Beyond academic research, in practice, most analysts take a hybrid approach. There have been works published on reverse-engineering Code Virtualizer [1] using both dynamic and static analysis techniques simultaneously [4, 7].

Chapter 3

Design

Our work comes out of the need for a way to handle VM-packed binaries more efficiently. There are usually only a handful of skilled-individuals, who do not always publish their methods, capable of dealing with the most advanced of software-protections. VM-protections are typically broken by manually reverse engineering each new VM-implementation. However, such methods are inaccessible to the average user and are often both costly and tedious. Some dynamic systems have been developed to address the challenge; however, they are often inefficient and incomplete. This is especially true in the case of virtualization packers. We have designed an automated system that leverages purely static program-analysis techniques to aid in reverse engineering of VM-based protection schemes. This system enables even novice analysts to understand the VM-interpreter in a time and cost-efficient manner.

3.1 Static Analysis

Unlike many of the current state-of-the-art virtual-machine analysis systems, our system was designed to use only static program analysis techniques. Thus, no instructions of the binaries that are analyzed are executed during the analysis process. This is especially important when dealing with malicious executables because there is much less risk of damage being done to friendly systems.

The nature of static analysis allows for a more complete view of the code to be analyzed. In dynamic analysis systems, only the code paths that are executed can be analyzed, and reproducibility can be a problem. It can be difficult to get certain code paths to execute deterministically and branch decisions are often unclear. In the static-case everything that is visible to the user is processed.

Additionally, unimportant portions of code that are executed many times over, which are extremely common in this kind of architecture, result in considerable time and resources being wasted. The set up code, dispatcher and decoding steps are repeated for each virtual instruction, resulting in a great deal of time wasted on code that is uninteresting. Static analysis allows us to avoid many of the inefficiencies that accompany dynamic-analysis. If we were required to execute code paths of the executable there might be considerable overhead and time spent on instructions unrelated to the portions we are interested in. This fact will often make static analysis techniques much faster to perform than dynamic-analysis techniques.

Moreover, we can avoid many execution and environment related pitfalls such as the myriad of anti-execution, anti-debugger [26], and anti-VM [27] tricks available. That is methods for avoiding: execution, analysis within a debugger, and analysis within virtual environments such as Qemu [15], which are necessary for many dynamic-analysis systems. A drawback however, is that there is

less information available when using pure static analysis, compared to the wealth of information available concerning code that is actually being executed, like in the dynamic case. To deal with this, many of the values in a static model must be symbolic or variable in nature. This also causes manual analysis to require more tedium or expertise in a similar scenario.

3.2 Assumptions

When analyzing binary code it is important to be able to properly disassemble the code sections of whatever native code you are dealing with. This is a particularly hard problem to solve, and has been compared to the classic halting-problem [28]. It becomes especially difficult when working against an active adversary that might be using anti-disassembly tricks [21]. In our work we have assumed that we are able to reliably disassemble the executables in question. Since our focus is on reversing virtualization obfuscators we naturally have only designed our system to handle the issues that arise with that specific family of obfuscation. We have no provisions for dealing with even the most basic of other-forms of obfuscation and leave that to other works. Our system is meant to be a proof-of-concept for automated static-analysis techniques applied to software protections using virtualization.

Chapter 4

Approach

The goal of our system is to provide automated analysis of virtualization protected binaries, and provide reports that can be useful for reverse engineering, even to a novice user. A schematic of our system can be seen in Figure 4. Our system takes a binary executable as input and outputs details of the virtual machine in question to aid in further analysis of the protected code.

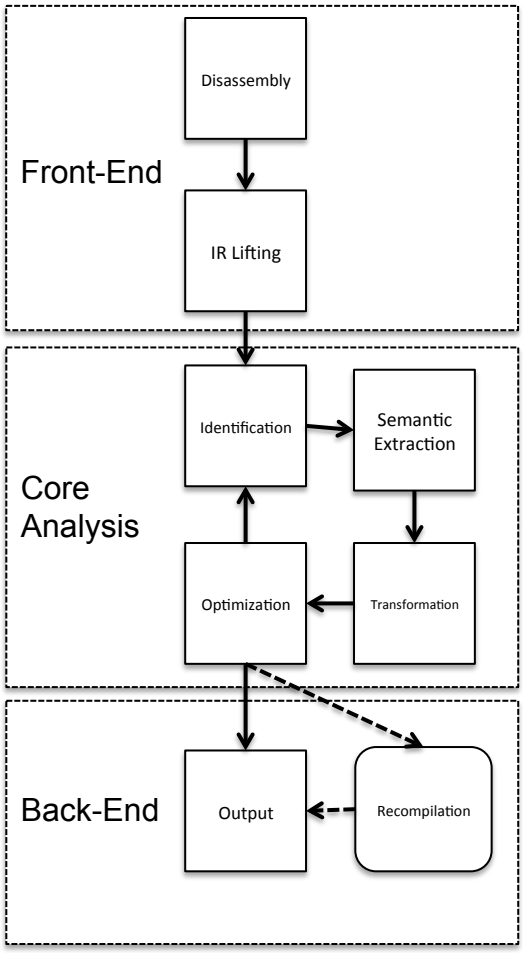

Figure 4.1: Overview

The front-end of our system reads the binary input file and then disassembles and lifts it to an intermediate representation, explained in Section 4.1. The core of our system analyzes and extracts

the relevant details of the VM implementation and then applies any number of transformations and optimizations to deobfuscate the translated code. This is outlined in the Sections 4.2, 4.3, and 4.4. Finally, the back-end of our system is responsible for providing an output that is useful to the user by displaying the relevant information explained in Section 4.4.

Implementation details of the system can be found in Section 5.

4.1 Disassembly and IR Lifting

First, we must convert the binary data into assembly instructions using a disassembler. The goal of disassembly is to translate raw binary data into human-readable mnemonics. Second, we lift the code segments to an intermediate representation (IR) to extract some semantics of the disassembly. Lifting the code in this manner makes the code easier to work with in the core analysis portion of our system. The IR also shows the system side-effects far more explicitly than native assembly. For example, changes to the flags register, type conversions, endianess, and temporary computations are made obvious.

```
addr 0x401013 @asm "cmp    $0x1,%al"
label pc_0x401013
T_t:u8 = low:u8(R_EAX:u32) - 1:u8
R_CF:bool = low:u8(R_EAX:u32) < 1:u8
R_OF:bool =
  high:bool((low:u8(R_EAX:u32) ^ 1:u8) & (low:u8(R_EAX:u32) ^ T_t:u8))
R_AF:bool = 0x10:u8 == (0x10:u8 & (T_t:u8 ^ low:u8(R_EAX:u32) ^ 1:u8))
R_PF:bool =
  ~low:bool(T_t:u8 >> 7:u8 ^ T_t:u8 >> 6:u8 ^ T_t:u8 >> 5:u8 ^ T_t:u8 >> 4:u8 ^
            T_t:u8 >> 3:u8 ^ T_t:u8 >> 2:u8 ^ T_t:u8 >> 1:u8 ^ T_t:u8)
R_SF:bool = high:bool(T_t:u8)
R_ZF:bool = 0:u8 == T_t:u8
```

Figure 4.2: IR representation of *cmp* instruction

As an example, the bytes **0x3c01**, would be disassembled as **cmp al, 0x1**, and lifted as the IR as seen in Figure 4.2. Here, R_CF, R_OF, R_AF, R_PF, R_SF, and R_ZF make up the flag register, and T_t is a temporary variable, since the result of the *cmp* instruction's subtraction is not stored.

4.2 Identification of Common Constructs

Once the code is in a standardized format we are able to search for common constructs in the VM-interpreter code. At this point we are just simply iterating and pattern matching. For example, a series of conditional jumps is often used in VM-packer implementations as a simple VM-instruction

decoder, mapping VM-encoded bytes to their appropriate handlers. If you were to disassemble a program packed by such a protector it might resemble Figure 4.3. In this trivial example, the current VM-instruction is being compared to numerical opcodes; when there is a match the corresponding instruction handler is invoked.

```
.text:00401013                cmp     curr_vm_inst, 1
.text:00401015                jnz     short loc_401021
.text:00401017                call    1_handler
.text:0040101C                jmp     next_vm_instruction
.text:00401021 ; ---------------------------------------------------
.text:00401021
.text:00401021 loc_401021:
.text:00401021                cmp     curr_vm_inst, 2
.text:00401023                jnz     short loc_40102F
.text:00401025                call    2_handler
.text:0040102A                jmp     next_vm_instruction
.text:0040102F ; ---------------------------------------------------
.text:0040102F
.text:0040102F loc_40102F:
.text:0040102F                cmp     curr_vm_inst, 3
.text:00401031                jnz     short loc_40103D
.text:00401033                call    3_handler
.text:00401038                jmp     next_vm_instruction
.text:0040103D                ...
```

Figure 4.3: Simple vm-instruction decoder

4.2.1 Semantic Constraint Solving

Our goal is to search for common constructs which yield valuable information about the vm-interpreter. However, if we were to do pattern matching alone, there would be far too much noise in the tool's output, due to the possible generality in certain code constructs, exemplified in Figure 4.3. Furthermore, it would be very difficult to create a system that uses pattern matching alone to identify many constructs without becoming extremely bloated and inefficient.

Instead, we leverage the semantics of the instructions along with a constraint solver, to increase our ability to identify certain structures with a higher level of granularity and efficiency than pattern matching alone. The sets of constraints are generated at run-time, and are dependent on what kind of structure is in question.

Resolving Virtualized Code References

One application of using a constraint solver in this manner is in resolving references to the obfuscated code. However, this can be quite difficult to do statically. Consider the case of opcode

handling. Typically the current vm-instruction, either residing in memory or in a register, is compared to potential values so that the appropriate handler may recognized and invoked, as shown in Figure 4.3. If the vm-instruction is in a general-purpose register or scratch memory location it is very difficult to know when references to the same register or memory location are referring to vm-specific values or arbitrary ones. However, if one utilizes the context of such references it can be narrowed down significantly. For instance, Figure 4.4 illustrates a common possible-idiom for loading a vm-instruction into the general-purpose register *eax*:

```
xor eax, eax
xor ecx, ecx
lea esi, code
mov cx, [vpc]
add esi, ecx
mov al, [esi]
```

Figure 4.4: Loading vm-instruction into general-purpose register *eax*

At the end of this snippet, *eax* contains the next vm-instruction. However, this observation is only evident immediately after the last *mov* instruction. If we were to choose another arbitrary instruction following this code this may no longer be true. We can check this condition at any point by using a constraint solver if we know the address of the beginning of the virtualized code segment, and optionally, the address or value of the virtual program counter/instruction pointer. In our implementation we make this a user-input, but others have shown the process of determining this address to be automatable, and is often fairly simple to do manually, even for a novice [33]. Depending on the specific construct, our system outputs a set of constraints which can be solved by a constraint solver such as STP [17]. In the case above, we might solve for goal where: $goal : bool := eax == code + [vpc]$. For a simple case of what an STP input might look like, refer to Figure 4.5.

```
% free variables:
R_EAX_5 : BITVECTOR(32);
goal_51292 : BITVECTOR(1);
mem_array_51395 : ARRAY BITVECTOR(32) OF BITVECTOR(8);
% end free variables.

ASSERT(
0bin1 =
(LET R_ECX_52095_0 =
    (0hex0000@((0bin00000000 @ mem_array_51395[0hex00413052])|
        (((0bin00000000 @ mem_array_51395[0hex00413053]) << 8)[15:0])))
IN
(LET R_ESI_52096_1 = BVPLUS(32, 0hex00403025,R_ECX_52095_0) IN
(LET R_EAX_52099_2 = ((R_EAX_5)[31:8]@mem_array_51395[R_ESI_52096_1]) IN
(LET T_t_52101_3 = BVSUB(8, (R_EAX_52099_2[7:0]),0hexff) IN
(LET R_ZF_52102_4 = IF (0hex00=T_t_52101_3) THEN 0bin1 ELSE 0bin0 ENDIF IN
((~(~(R_ZF_52102_4))|(0bin0&goal_51292))&
(~(~(~(R_ZF_52102_4)))|(0bin0&goal_51292)))))))
);
QUERY(FALSE);
COUNTEREXAMPLE;
```

Figure 4.5: Example constraint solver input

In the case of the commercial packer VMProtect [20], we can check if *eax* is referencing the virtualized code by solving for: $goal : bool := eax == [esi - 1]$.

4.3 Extracting Semantics

Once we have identified the structures we are interested in, we extract the relevant semantics. In our running example, this translates to the virtual instruction set's opcodes and respective handlers. This is done fairly easily given the identification steps along with the the expressiveness of the intermediate representation. Determining the mapping of vm-bytecode to x86 vm-instruction handlers is essential for unpacking a protected binary. Other possible uses for the process would include extracting vm-context specifics. The raw disassembly of our simple decoder from section 4.2 is shown in Figure 4.6

```
.text:00401013                cmp     al, 1
.text:00401015                jnz     short loc_401021
.text:00401017                call    sub_401125
.text:0040101C                jmp     loc_401118
.text:00401021 ; ---------------------------------------------------
.text:00401021
.text:00401021 loc_401021:
.text:00401021                cmp     al, 2
.text:00401023                jnz     short loc_40102F
.text:00401025                call    sub_401148
.text:0040102A                jmp     loc_401118
.text:0040102F ; ---------------------------------------------------
.text:0040102F
.text:0040102F loc_40102F:
.text:0040102F                cmp     al, 3
.text:00401031                jnz     short loc_40103D
.text:00401033                call    sub_40116B
.text:00401038                jmp     loc_401118
.text:0040103D                ...
```

Figure 4.6: The raw disassembly of a simple decoding idiom

The opcode-handler mapping is fairly obvious. If vm-instruction 1 is encountered then the decoder will call the subroutine at memory address 0x401125, 2 would result in the subroutine at 0x401148, and so on.

In the case of VMProtect [20], the fetch-decode portion is a essentially a load and *jmp* table offset, which when viewed in a disassembler would look like Figure 4.7. We are able to extract the bytecode-handler mapping by knowing that *esi* effectively serves as the virtual instruction pointer from which a byte is loaded, and by looking at the *jmp* target located at the memory address 0x004C2080. In this case, there is a table of function pointers to the handling functions beginning at memory address 0x4C2171.

```
004C207A  mov     al, [esi]
004C207C  movzx   eax, al
004C207F  inc     esi
004C2080  jmp     dword [eax*4+0x4c2171]
```

Figure 4.7: VMProtect's fetch-decode snippet

4.4 Post-Processing: Deobfuscation, Optimization, and Output

At this stage of the analysis, we have obtained semantic information about the vm-interpreter and virtual instruction set. Now, we can begin the deobfuscation, optimizations, and output a useful result to the user. Given the mapping from VM-bytecode to the handler-code, we begin deobfuscating each handler's subroutine. The exact techniques that must be applied are fairly subjective and is mostly outside the scope of this paper. We will show some example methods that are particularly effective for these types of protectors. For instance, Figure 4.8 shows a toy bytecode-program obfuscated with StackVM, from Appendix A.

```
0x05 0xCD 0xAB            ; vmov r0, 0xABCD
0x01                      ; vpush r0
0x05, 0x011, 0x011        ; mov r0, 0x1111
0x0B                      ; vxchg r0, r1
0x02                      ; vpop r0
```

Figure 4.8: Example StackVM bytecode

Using the bytecode-handler mapping extracted in previous steps, we can replace the bytecode portion of virtual instructions with their corresponding handler subroutine-bodies. After direct replacement the user is left with something similar to Figure 4.9.

```
xor edx, edx
mov dx, ABCDh
mov word ptr [vr0], dx
sub vsp, 2
mov ax, [vr0]
lea ebx, stack
xor edi, edi
mov di, vsp
add edi, ebx
mov word ptr[edi], ax
xor edx, edx
mov dx, 1111h
mov word ptr [vr0], dx
mov ax, [vr0]
mov bx, [vr1]
mov word ptr [vr1], ax
mov word ptr [vr0], bx
lea ebx, stack
xor edi, edi
mov di, vsp
add edi, ebx
mov ax, [edi]
mov word ptr[vr0], ax
add vsp, 2
```

Figure 4.9: Example StackVM bytecode after direct replacement

Figure 4.9 is similar to the output of a dynamic-analysis tool if you traced the execution of the obfuscated program. However, in our static system we are able to skip having to analyze and process the control-flow code that is quite sizeable in this type of obfuscator. While we now have fairly straight-line code, which is a significant improvement over the distributed control-flow of the original program, the code's actual meaning is difficult to discern and is quite inefficient and lengthy.

There are numerous options from this point to begin the deobfuscation process. We begin by using a number of compiler optimizations. Then, a few heuristic replacements are applied to the resulting code in order to get very close to the ideal unobfuscated code. Others first proposed the use of compiler optimizations for this purpose in 2008 [4, 31] in dealing with VMProtect and Code Virtualizer [1]. Constant folding and constant propagation allow the substitution of variables for known-constants when the variable in question is constant at the time the expression is evaluated.

In the case of the running example, after constant folding and constant propagation the user is left with Figure 4.10.

```
xor edx, edx
mov dx, ABCDh
mov word ptr [vr0], ABCDh
sub vsp, 2
mov ax, ABCDh
lea ebx, stack
xor edi, edi
mov di, vsp
add edi, ebx
mov word ptr[edi], ABCDh
xor edx, edx
mov dx, 1111h
mov word ptr [vr0], 1111h
mov ax, 1111h
mov bx, [vr1]
mov word ptr [vr1], 1111h
mov word ptr [vr0], bx
lea ebx, stack
xor edi, edi
mov di, vsp
add edi, ebx
mov ax, [edi]
mov word ptr[vr0], ax
add vsp, 2
```

Figure 4.10: Example StackVM bytecode after constant folding and constant propagation

At this point we are able to apply dead code elimination to get rid of the statements that are no longer necessary. Dead code is simply the set of instructions that are executed but do not affect the end result or whose results are no longer referenced after they are executed. Peephole optimization is an optimization technique that replaces short sequences of instructions within a small window or "peephole", with more efficient, or in this case less-obfuscated, instructions. This step allows us to do some heuristic-based deobfuscation dependent on the specific implementation of VM we are dealing with. In the example, we removed several instructions that became dead code and are able to eliminate the virtual stack completely through peephole optimizations. After dead code elimination and peephole-optimization we are left with Figure 4.11.

```
mov bx, [vr1]
mov word ptr [vr1], 1111h
mov word ptr [vr0], bx
mov word ptr [vr0], ABCDh
```

Figure 4.11: Example StackVM bytecode after dead code elimination and peephole-optimization

Now that we have reduced the size of the code considerably, from twenty-four to four instructions in our running example, and effectively deobfuscated the code, we must repeat the process for the entire virtualized bytecode program. This will essentially leave the user with an unobfuscated program.

At this point, we provide the user with a useful output. A classical step would be to "recompile" the new intermediate representation into an executable, but this is somewhat outside of our goals. In our system, we simply print the results to standard output, which can also easily be written to a file. However, if one were so inclined it would be fairly straightforward to generate a new executable, effectively unpacking the original-obfuscated executable.

By following this approach, we created a system that allows relatively novice users to perform analysis on a very sophisticated set of software protections in an efficient manner. Furthermore, as the set of common constructs and post-processing transformation grows, the number of specific implementations that can be dealt with in the future does as well. We believe that this generalization and work-reuse is a great asset for a system such as ours. In a field that can be quite tedious and extremely expensive to start from scratch on each new project, this system could be very valuable.

Chapter 5

Implementation

There are three main stages in our system: front-end processing, core analysis, and back-end output. Our system builds upon the Binary Analysis Platform (BAP) [24] framework, the successor of Vine [18], the static analysis component from the BitBlaze platform [34]. We chose this framework because it provides several nice book-keeping and general analysis features out-of-the-box, including explicit endianess. It is well-documented, and along with its predecessors, is fairly well-known and used throughout many program analysis, reverse-engineering and malware analysis-focused, academic research groups.

Our system was implemented for IA-32 (x86), the most popular architecture for this kind of work and malware in general, but the techniques themselves are platform-independent. Other architectures are outside the scope of this paper and were not tested. All development and experiments were done on Debian 6.0.7 Squeeze with an Intel® Core^{TM} i5-3210M CPU with a 2.50 GHz clock and 2 GB of RAM. All analysis code is written in OCaml, with many test binaries written in MASM and IA-32 (x86).

5.1 Front-End

In the first step, we use BAP's linear sweep disassembler on the input binaries. Disassembly is the process by which binary data is interpreted as native instructions; the output mnemonics also happen to be more descriptive to a human.

The next step is to convert the disassembly into a form that is more suitable for analysis and transformations, an intermediate representation. BAP's *asmir* module lifts the assembly into BAP's Intermediate Language (BIL). This provides us with a good starting point to perform our analyses and transformations because it is a fairly well-documented intermediate representation. Figure 5.1 shows the formalization of the IR, but for the full specification please refer to the BAP Handbook [23].

5.2 Core Analysis

From the IR we are able to construct a control-flow graph (CFG). Refer to Figure 5.1 for an example CFG of a simple "Hello World!" program. For reasons discussed in Section 5.4, before we continue the CFG must be cleaned up by removing certain components that BAP has trouble with, such as: indirect *jmp* instructions, graph meta data, comments, and special nodes. In order to analyze the code segments, we transform the CFG into an abstract syntax tree (AST) and begin the core analysis phase. The identification of the constructs common to virtual-machine protections, as well

program	::=	*stmt**
stmt	::=	*var* := *exp* | `jmp`(*exp*) | `cjmp`(*exp*,*exp*,*exp*)
		| `halt`(*exp*) | `assert`(*exp*) | `label` *label_kind* | `special`(string)
exp	::=	`load`(*exp*, *exp*, *exp*, τ_{reg}) | `store`(*exp*, *exp*, *exp*,*exp*,τ_{reg}) | *exp* $\Diamond_b$ *exp*
		| $\Diamond_u$ *exp* | *var* | `lab`(string) | *integer* | `cast`(*cast_kind*,τ_{reg},*exp*)
		| `let` *var* `=` *exp* `in` *exp* | `unknown`(string, τ) | `name`(*exp*)
label_kind	::=	*integer* | string
cast_kind	::=	`unsigned` | `signed` | `high` | `low`
var	::=	(string, id_v, τ)
$\Diamond_b$	::=	$+, -, *, /, /_s, \text{mod}, \text{mod}_s, \ll, \gg, \gg_a, \&, \mid, \oplus, ==, !=, <, \leq, <_s, \leq_s$
$\Diamond_u$	::=	$-$ (unary minus), $\sim$ (bit-wise not)
value	::=	*integer* | *memory* | string | $\perp$
integer	::=	n ($:\tau_{\text{reg}}$)
memory	::=	{ *integer* $\rightarrow$ *integer*, *integer* $\rightarrow$ *integer*, ... } ($:\tau_{\text{mem}}$)
τ	::=	τ_{reg} | τ_{mem}
τ_{mem}	::=	`mem_t`(τ_{reg}) | `array_t`($\tau_{\text{reg}}, \tau_{\text{reg}}$)
τ_{reg}	::=	`reg1_t` | `reg8_t` | `reg16_t` | `reg32_t` | `reg64_t`

Table 5.1: BIL Specification

as deobfuscation and optimization processing were all invoked as transformations on the BAP-lifted intermediate representations. Specifically, they were each written as OCaml modules and run by making modifications to the BAP *iltrans* tool.

Basic optimizations on the CFG/AST are performed: Cycles and unreachable nodes are removed from the graphs and basic blocks which can be combined are *coalesced*. Furthermore, the CFG is converted into a common compiler optimization-friendly format known as static single assignment (SSA). In this form, each variable in the IR is guaranteed to only be assigned once.

Each relevant construct is specified before the tool is run, but must only be done once per structure and can be used from then on, independent of specific protection-implementation details. General pattern matching is applied recursively by iterating over the AST through standard OCaml statements in accordance with the BAP AST specification.

The semantic extraction and identification is accomplished simultaneously considering the syntax of the "signatures" written for each common structure and how OCaml types are utilized. When a match is found it is generally checked for virtual code references using the constraint solver. Each relevant vm-specific structure is a set variables in an OCaml expression and once the values are obtained through pattern matching and semantic constraint solving they are passed to the next step. The virtual code references are checked as described in Section 4.2.1. The constraint formulas pertaining to the specific type of VM-identified are generated at run-time. Constraint solving is done by exporting the formulas to STP [17] version 1373M.

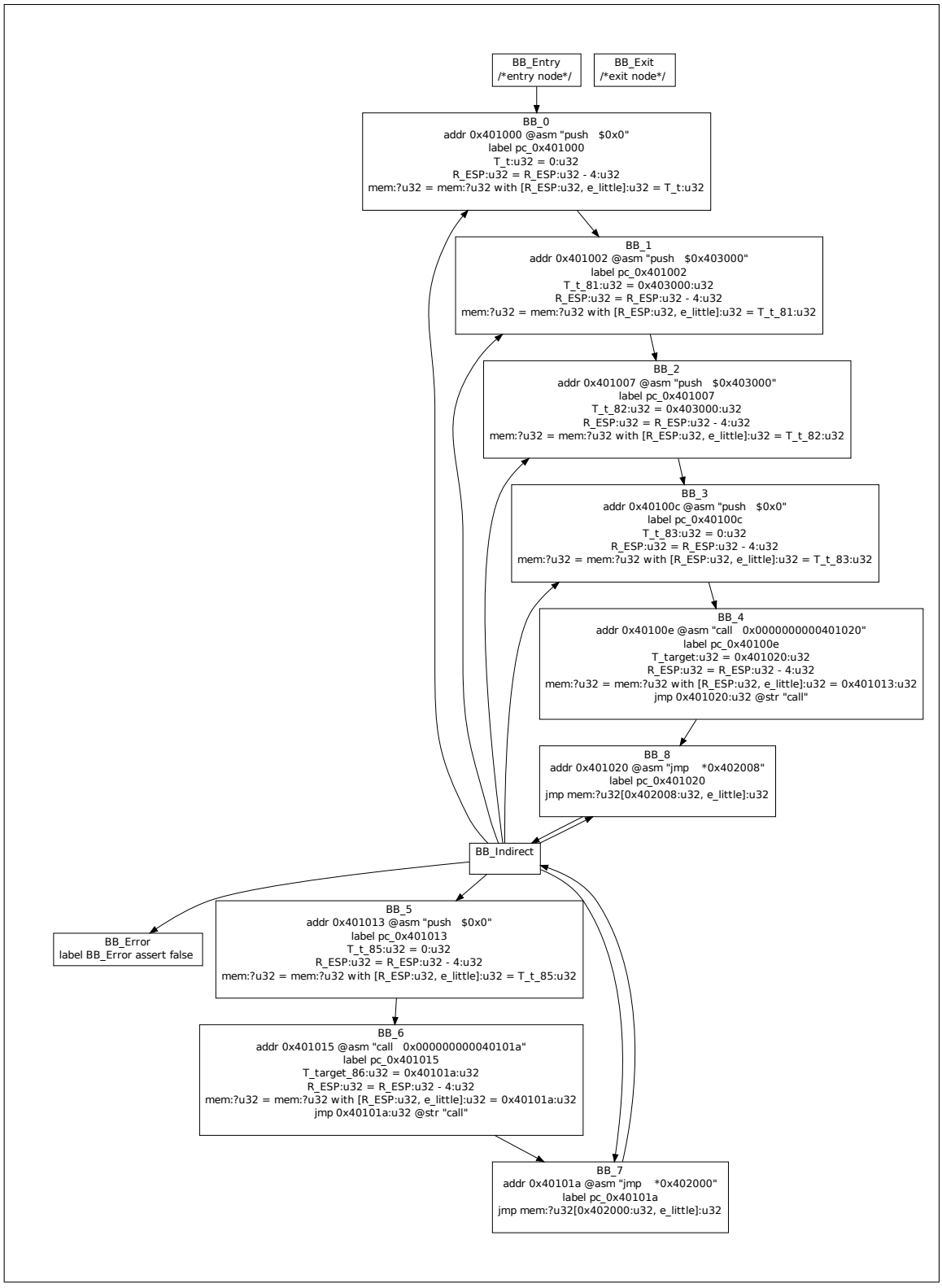

Figure 5.1: Sample CFG of “Hello World!” program

5.3 Back-End

As previously mentioned, at the end of the process, the user is presented with a report. The output includes all relevant details regarding the VM-specifics that were analyzed. It shows any of the code constructs found, VM type, suspected opcodes and accompanying handlers. There is also an optional verbose mode, which outputs all computations, constraint formulas, and their solutions.

5.4 Challenges

We found that binaries packed by VMProtect had meta data that was, most-likely purposely, incorrect. In particular, some of the fields in the PE header regarding the memory image size were incorrect. The binaries lied about how big they were, claiming to be smaller than their actual size. The PE/COFF specification [9] defines the format binaries on the Microsoft Windows platform must follow. Because our system relies on these values in the default case, we had to manually input the range of memory to analyze. This is not at all difficult to do, but one must notice the trick that VMProtected binaries employ.

Furthermore, it is worth noting that in spite of BAP's robustness there were several challenges encountered, mostly with transformations done on the intermediate representation returned by BAP. In particular BAP's public release has problems with indirect *jmp* instructions such as *jmp %eax* and the intermediate representation had some meta data which caused the representation to be slightly imprecise and cumbersome. Both of these issues were dealt with by hand, doing some processing and clean-up after the binaries were lifted, but before being processed by our modules.

There are also concerns in regards to BAP's linear sweep disassembler. Disassembly is a sensitive process due to the nature of binary code. It can be difficult to discern the separation of data and code. If the point where the code evaluation begins is off by one single bit, the code can be interpreted completely incorrectly relative to its intended meaning. Thus, we would have preferred to use a recursive algorithm for disassembly since they are generally more resilient to anti-disassembly tricks.

Chapter 6

Results and Evaluation

6.1 Methodology

In order to evaluate our system, it was tested against three types of virtual-machine based protections: First, was a simple stack-based machine called StackVM, detailed in Appendix A, which we wrote ourselves. Second, was a well-documented VM [2] written by a third party, popularized by a "crackme", which are essentially reverse engineering puzzles popular in the security, malware analysis, capture-the-flag, and reverse engineering communities. Third, we tested our system against a popular commercial protector, often found packing malware in the wild, VMProtect [20]. Unfortunately, because our system is more of an applied methodology, which results in a useful tool and even forms somewhat of a framework, it is difficult to quantify its effectiveness. However, we will discuss the varying levels of success on each set of binaries we tested it on. We also provide statistics relevant for performance comparisons to alternative analysis systems and techniques.

6.2 StackVM

In analyzing the simplest VM, the one we developed ourselves, the system performed well. We were able to extract all seventeen opcodes corresponding to the virtual instruction set and the accompanying handler subroutines flawlessly, without false-positives. Each analysis of binaries of this type only took on average 6.2 seconds. This is not surprising since the VM is so small. This by itself is a very useful result because reverse engineering, even a simple VM like this, would take orders of magnitude longer and be extremely tedious.

6.3 FuelVM

The analysis of the second VM that we tested was quite similar. We were again able to extract the primary details particular to the VM implementation without any special modifications. However, there was an imprecision in the way we determined potential opcodes. Due to the way the VM passes operands, the code used to decode them is typically recognized as part of the opcode decoding. This can be handled by specifying exceptions particular to this implemention. It is worth noting that while the VM-context is very similar to StackVM with respect to the virtual instruction bookkeeping, in the decode-step, there was another temporary memory address which held a copy of the VM-bytecode, but it was handled without issue. The analyses on this VM took on average 18.6 seconds.

Another aspect of this particular implementation worth noting is the presence of anti-debugging measures. At the start of the program a custom structured exception handler (SEH) frame is created and added to the SEH chain. An exception is then triggered by an *int 3* instruction, which triggers an interrupt meant to trap to a system's debugger. If a debugger is not attached, this causes the exception handler that was just added to the SEH chain to gain control. Disassembly of the first custom SEH is shown in Figure 6.1.

```
.text:004012D7     customSEH1:
.text:004012D7     pop     large dword ptr fs:0
.text:004012DE     add     esp, 4
.text:004012E1     push    offset customSEH2
.text:004012E6     push    large dword ptr fs:0
.text:004012ED     mov     large fs:0, esp
.text:004012F4     mov     eax, 1
.text:004012F9     xor     ecx, ecx
.text:004012FB     div     ecx
.text:004012FD     xor     eax, eax
.text:004012FF     mul     ecx
```

Figure 6.1: Custom SEH1

The SEH chain is a linked list that is used when an exception occurs. Each exception handler in the chain is called in succession until the exception is dealt with or the end of the chain is reached; *fs*:0 refers to the beginning of the SEH chain and the first member of the data structure known as the Thread Information Block (TIB), which is specific to Windows. In the first custom SEH an exception is triggered by zeroing out *ecx* and attempting a division operation. This causes a divide by zero exception. There are two more custom SEH's chained together by the VM as a means to thwart dynamic-analysis in the form of a debugger because if you attach a debugger the exceptions will not reach the custom handlers, and execution will be interrupted. Since our tool is purely static this was not an issue at all.

6.4 VMProtect

In the case of the binaries packed with VMProtect, we were able to perform our analysis, but with a few modifications. The main modifications needed to analyze VMProtected binaries comes from their use of an indirect *jmp* instruction in their dispatcher's decoding step. The current version of BAP (0.7) has issues dealing with indirect $jmp's$, so we had to implement a workaround specifically for this case. In a related note, because the opcodes are actually used in calculating the *jmp* target as seen in Figure 4.7, where *eax* contains the bytecode and is multiplied by four to get the offset into the handler functions' *jmp*-table, the opcode identification is actually relatively simple when compared to the other VM's we looked at in this paper. Due to this simplicity, coupled with the

modifications needed, it is not quite fair to compare analysis times for this VM, but it was very quick, on the order of seconds. We used the trial version of VMProtect Ultimate v2.12 on Windows XP for all VMProtect-related experiments and analysis.

6.5 Virtual Code Independence

Due to the design of our system. The analysis of the VM-obfuscated binaries is relatively independent of the virtual code that will be interpreted. This is because the bulk of the analysis focuses on reverse engineering the details of the VM itself and its interpreter. This means that the run-time performance of our system should not increase by very much as the size of the virtual program increases. It is also very contrary to what would happen in the case of a dynamic analysis system, which must execute and analyze the entirety of the virtualized code. To test this, we performed our analysis on programs with varying lengths of bytecode within StackVM executables. We used bytecode programs of lengths: 42, 142, 542, and 1042 bytecode instructions. We followed a similar procedure for VMProtect. However, because VMProtect is commercial software, it was more difficult to precisely control the size of the vm-bytecode. Instead, we varied the size of the executables that we packed with VMProtect. We tested a toy "Hello World!" program of size 2.5KB, and several Windows XP utilities from C:\WINDOWS\system32: notepad.exe (67.5KB), calc.exe (112KB), and cmd.exe (380KB). In all cases the affect on run-time was negligible.

Chapter 7

Future Work

This paper is meant to provide a methodology for developing an automated system for statically analyzing binaries protected by virtual-machine obfuscators with an emphasis on efficiency and accessibility. The system we developed is meant to be a proof of concept for this approach. That being said, there are a number of improvements left for future work.

As mentioned in Section 5.4, a difficulty we faced involved the framework we built upon having some issues with certain instructions, most notably indirect *jmp*'s. This caused quite a few problems, which we had to develop work-arounds for, but we hope future versions will solve those issues. Whether developers introduce solutions in later revisions, we write patches ourselves, or if we find other means, support for these kinds of instructions should greatly improve the efficiency and generality of our system.

There was a problem we encountered when analyzing VMProtected binaries involving the PE header fields lying about the binary. We could solve this in the future by calculating values such as the size of the image manually instead of relying on the values in the header fields.

We also would like to increase the catalog of common structures that we are able to identify and analyze. Right now our system is only able to deal with the most common cases: a standard fetch-decode-execute workflow and a few different decoding schemes. A wider breadth of capabilities would make our system much more robust. Similarly, more post-processing modules should be written to deal with different kinds of obfuscation schemes and to remove different VM-implementation specific features where possible.

Finally, we would like to eventually add support for full-code generation and recompilation after the post-processing transformations. This would allow a working executable to be output that could be loaded into other tools which take binary executables as input.

Chapter 8

Conclusions

The analysis of Virtual-Machine packed binaries is typically a very challenging task. There have been attempts to develop dynamic-analysis tools to aide in their reverse-engineering. However, these systems suffer from many performance and run-time issues. Typically, a very skilled individual must reverse-engineer each new implementation of a VM-Packer from scratch, by hand, through mostly static means such as looking through an interactive disassembler. This process is expensive, time-consuming, and tedious.

We have successfully developed a system as alternative to previous methods. It is meant to provide automated, purely-static-analysis of Virtual-Machine protected binaries in a manner that is more efficient than pure manual analysis. This is accomplished by reusing analysis of common structures, leveraging semantics of processed instructions, and reducing some of the tedium of repeating the same analysis, either on the same binary or on several different binaries, by invoking a programmatic solution. It is also faster and more complete then dynamic-analysis alone. By keeping the analysis confined to static techniques, we are able to avoid many pitfalls that dynamic-analysis systems fall into. These include anti-execution, anti-debugging, and anti-virtualization tricks. The system is also able to analyze entire executables as a whole without relying on the execution of singular code paths. Our system is able to extract several features of current VM protections on an extensible platform that is accessible to even novice analysts. It was evaluated by analyzing three separate VM-protections: a simple VM we developed, a third-party VM, and a commercial protection scheme. This work shows that an approach such as ours, is capable of efficiently analyzing even the most advanced protection schemes.

Appendix A

StackVM

StackVM is a small stack-based VM that we developed for testing. It has seventeen virtual instructions and maintains two virtual registers and a virtual stack. The instructions are stored in a data block to which a virtual offset is kept track of. Virtual instruction decoding is handled by a simple switch-like structure. It is implemented in MASM and the primary code is listed below:

```
.386
.model flat, stdcall
assume fs:flat
option casemap :none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

COMMENT /
Stack VM - A Simple stack-based VM
2KB stack and stack offset vsp
2 general purpose registers: r0, r1
program counter: vpc

ISA:
0x01 vpush r0
0x02 vpop r0
0x03 vmov r0, r1
0x04 vmov r1, r0
0x05 vmov r0, imm16
0x06 vadd r0, r1
0x07 vsub r0, r1 ; r0 = r0-r1
0x08 vand r0, r1
```

```
0x09 vor r0, r1
0x0A vxor r0, r1
0x0B vxchg r0, r1
0x0C vjeq offset
0x0D vja offset
0xFE vnop
0xFF exit
/

.data
 input db "StackVM Exiting.", 0
 correct db "Correct!", 0
 incorrect db "Incorrect!", 0
 code db 05h, 0CDh, 0ABh ; mov r0, 0xABCD
      db 01h ; push r0
    db 05h, 011h, 011h ; mov r0, 0x1111
    db 04h ; mov r1, r0
    db 05h, 088h, 088h ; mov r0, 0x8888
    db 07h ; sub r0, r1
    db 04h ; mov r1, r0
      db 02h ; pop r0
    db 0bh ; xchg r0, r1
    db 01h ; push r0
      db 0FEh ; nop
    db 05h, 08h, 0 ; mov r0, 8
    db 04h ; mov r1, r0
    db 05h, 2Fh, 13h ; mov r0, 0x133F
    db 03h ; add r0, r1
    db 0Ch, 1, 0; jeq 1
    db 0FDh ; incorrect
    db 0FCh ; correct
    db 04h ; mov r1, r0
    db 05h, 038h, 13h ; mov r0, 0x1338
    db 0Dh, 3, 0; ja 3
    db 0FDh ; incorrect
    db 0FEh ;  nop
    db 0FFh ; exit
    db 0FCh ; correct
    db 0FFh, 0 ; exit

 ; VM State
```

```
 stack db 65536 DUP(?) ; 2^16
 vsp dw 0FFFFh ; VM stack pointer
 vpc dw 0 ; VM instruction pointer
 vr0 dw 0 ; general purpose register 0
 vr1 dw 0 ; general purpose register 1

.code
start:
 lea esi, code ; fetch code[vpc]
 xor ecx, ecx
 mov cx, [vpc]
 add esi, ecx
 mov al, [esi]

 .IF al == 01h
  call vpush
 .ELSEIF al == 02h
  call vpop
 .ELSEIF al == 03h
  call vmov01
 .ELSEIF al == 04h
  call vmov10
 .ELSEIF al == 05h
  call vmovimm
 .ELSEIF al == 06h
  call vadd
 .ELSEIF al == 07h
  call vsub
 .ELSEIF al == 08h
  call vand
 .ELSEIF al == 09h
  call vor
 .ELSEIF al == 0Ah
  call vxor
 .ELSEIF al == 0Bh
  call vxchg
 .ELSEIF al == 0Ch
  call vjeq
 .ELSEIF al == 0Dh
  call vja
 .ELSEIF al == 0FCh
```

```
  invoke MessageBox, NULL, addr correct, addr correct, MB_OK
 .ELSEIF al == 0FDh
  invoke MessageBox, NULL, addr incorrect, addr incorrect, MB_OK
 .ELSEIF al == 0FEh
  call vnop
 .ELSEIF al == 0FFh
  invoke MessageBox, NULL, addr input, addr input, MB_OK
  invoke ExitProcess, 0
 .ENDIF

 add word ptr[vpc], 1
 jmp start

vpush:
 sub vsp, 2 ; decrement stack pointer
 ; mov ax, [esp+4]
 mov ax, [vr0]
 lea ebx, stack
 xor edi, edi
 mov di, vsp
 add edi, ebx
 mov word ptr[edi], ax
 ret

vpop:
 lea ebx, stack
 xor edi, edi
 mov di, vsp
 add edi, ebx
 mov ax, [edi]
 mov word ptr[vr0], ax
 add vsp, 2
 ret

vmov01:
 mov ax, [vr1]
 mov word ptr [vr0], ax
 ret

vxchg:
 mov ax, [vr0]
```

```
 mov bx, [vr1]
 mov word ptr [vr1], ax
 mov word ptr [vr0], bx
 ret

vmov10:
 mov ax, [vr0]
 mov word ptr [vr1], ax
 ret

vmovimm:
 add word ptr[vpc], 1
 lea esi, code ; fetch code[vpc]
 xor ecx, ecx
 mov cx, [vpc]
 add word ptr[vpc], 1
 add esi, ecx
 xor edx, edx
 mov dx, [esi]
 mov word ptr [vr0], dx
 ret

vadd:
 mov ax, [vr0]
 add ax, [vr1]
 mov word ptr [vr0], ax
 ret

vsub:
 mov ax, [vr0]
 sub ax, [vr1]
 mov word ptr [vr0], ax
 ret

 vand:
 mov ax, [vr0]
 and ax, [vr1]
 mov word ptr [vr0], ax
 ret

 vor:
```

```
 mov ax, [vr0]
 or ax, [vr1]
 mov word ptr [vr0], ax
 ret

vxor:
 mov ax, [vr0]
 xor ax, [vr1]
 mov word ptr [vr0], ax
 ret

vnop:
 call vmov01
 call vmov01
 ret

vja:
 add word ptr[vpc], 1
 mov ax, [vr0]
 mov bx, [vr1]
 cmp ax, bx
 jna end_ja
 lea esi, code ; fetch code[vpc]
 xor ecx, ecx
 mov cx, [vpc]
 add esi, ecx
 xor edx, edx
 mov dx, [esi]
 add word ptr [vpc], dx
 end_ja:
  add word ptr[vpc], 1
  ret

vjeq:
 add word ptr[vpc], 1
 mov ax, [vr0]
 mov bx, [vr1]
 cmp ax, bx
 jne end_jeq
 lea esi, code ; fetch code[vpc]
 xor ecx, ecx
```

```
 mov cx, [vpc]
 add esi, ecx
 xor edx, edx
 mov dx, [esi]
 add word ptr [vpc], dx
 end_jeq:
  add word ptr[vpc], 1
  ret
end start
```

References

[1] Code Virtualizer - Total Obfuscation againt Reverse Engineering. http://www.oreans.com/codevirtualizer.php.

[2] Crackme FuelVM v0.1 by daybreak. http://crackmes.us/read.py?id=361.

[3] Defeating HyperUnpackMe2 With and IDA Processor Module. http://www.openrce.org/articles/full_view/28/.

[4] Fighting Oreans' VM (code virtualizer flavour).

[5] IDA Pro. https://www.hex-rays.com/products/ida/index.shtml.

[6] Infographic: The state of malware 2013 | McAfee.

[7] Inside Code Virtualizer. http://repo.meh.or.id/Reverse%20Engineering/Inside%20Code%20Virtualizer.pdf.

[8] Mal(ware) formation statistics. http://research.pandasecurity.com/malwareformation-statistics/.

[9] Microsoft PE and COFF Specification. http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx.

[10] OllyBonE. http://www.joestewart.org/ollybone/.

[11] OllyDbg. http://www.ollydbg.de/.

[12] Part 1: Bytecode and IR. http://www.openrce.org/blog/view/1239/Part_1:__Bytecode_and_IR.

[13] Part 2: Introduction to Optimization. https://www.openrce.org/blog/view/1240/Part_2:__Introduction_to_Optimization.

[14] Part 3: Optimizing and Compiling. https://www.openrce.org/blog/view/1241/Part_3:__Optimizing_and_Compiling.

[15] QEMU - Open Source Processor Emulator. http://wiki.qemu.org/.

[16] Semi-Automated Input Crafting by Symbolic Execution, with an Application to Automatic Key Generator Generation. http://www.openrce.org/blog/view/2049/.

[17] STP Constraint Solver. https://sites.google.com/site/stpfastprover/.

[18] Vine. http://bitblaze.cs.berkeley.edu/release/vine-1.0/howto.html.

[19] VMProtect, Part 0: Basics. http://www.openrce.org/blog/view/1238/VMProtect,_Part_0:__Basics.

[20] VMProtect Software. http://vmpsoft.com/.

[21] John Aycock, Rennie deGraaf, and Michael Jacobson Jr. Anti-disassembly using cryptographic hash functions. *Journal in Computer Virology*, 2(1):79–85, August 2006.

[22] Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. *Zap: Automated Theorem Proving for Software Analysis*. 2005.

[23] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward Schwartz. The BAP Handbook. `http://bap.ece.cmu.edu/doc/bap.pdf`.

[24] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward Schwartz. Bap: a binary analysis platform. In *Computer Aided Verification*, page 463469, 2011.

[25] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, page 275284, 2011.

[26] Peter Ferrie. The "Ultimate" Anti-Debugging Reference. `http://pferrie.host22.com/papers/antidebug.pdf`.

[27] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[28] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, August 1980.

[29] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, page 4653, 2007.

[30] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: fast, generic, and safe unpacking of malware. *ACSAC*, pages 431–441, 2007.

[31] Rolf Rolles. Unpacking virtualization obfuscators. USENIX Association, 2009.

[32] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: automating the hidden-code extraction of unpack-executing malware. *ACSAC*, pages 289–300, 2006.

[33] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, page 94109, 2009.

[34] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: a new approach to computer security via binary analysis. In R. Sekar and Arun K. Pujari, editors, *Information Systems Security*, number 5352 in Lecture Notes in Computer Science, pages 1–25. Springer Berlin Heidelberg, January 2008.

[35] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT solvers for software security. In *WOOT'12: Proceedings of the 6th USENIX conference on Offensive technologies*. USENIX Association.