Rotalumè: A Tool for Automatic Reverse Engineering of Malware Emulators

Monirul Sharif Andrea Lanzi Jonathon Giffin Wenke Lee
School of Computer Science, Georgia Institute of Technology
{*msharif, andrea, giffin, wenke*}*@cc.gatech.edu*

Abstract

Malware authors have recently begun using emulation technology to obfuscate their code. They convert native malware binaries into bytecode programs written in a randomly generated instruction set and paired with a native binary emulator that interprets the bytecode. No existing malware analysis can reliably reverse this obfuscation technique. In this paper, we present the first work in automatic reverse engineering of malware emulators. Our algorithms are based on dynamic analysis. We execute the emulated malware in a protected environment and record the entire x86 instruction trace generated by the emulator. We then use dynamic data-flow and taint analysis over the trace to identify data buffers containing the bytecode program and extract the syntactic and semantic information about the bytecode instruction set. With these analysis outputs, we are able to generate data structures, such as control-flow graphs, that provide the foundation for subsequent malware analysis. We implemented a proof-of-concept system called Rotalumè and evaluated it using both legitimate programs and malware emulated by VMProtect and Code Virtualizer. The results show that Rotalumè accurately reveals the syntax and semantics of emulated instruction sets and reconstructs execution paths of original programs from their bytecode representations.

1. Introduction

Malware authors often attempt to defeat state-of-the-art malware analysis with obfuscation techniques that are becoming increasingly sophisticated. Anti-analysis techniques have moved from simple code encryption, polymorphism, and metamorphism to multilayered encryption and page-by-page unpacking. One new alarming trend is the incorporation of *emulation technology* as a means to obfuscate malware [23], [35]. With emulation techniques maturing, we believe that the widespread use of emulation for malware obfuscation is imminent.

Emulation is the general approach of running a program written for one underlying hardware interface on another. An obfuscator that utilizes emulation would convert a binary program for a real instruction set architecture (ISA), such as x86, to a bytecode program written for a randomly generated virtual ISA and paired with an emulator that emulates this ISA on the real machine. Figure 1 shows an example of this obfuscation process. The obfuscator has complete freedom to choose the semantics of the bytecode instructions, and entities such as virtual registers and memory addresses can be independent from the underlying real machine. For example, the Java virtual machine executes on commodity register machines but emulates a stack machine ISA. The obfuscated program is the generated emulator together with a data block containing bytecode. Code protection tools such as Code Virtualizer [23] and VMProtect [35] are real-world examples of this class of obfuscators.

Without knowledge of the source bytecode language, many existing malware analysis schemes are crippled in the face of malware obfuscated with emulation. At one end of the spectrum, emulators completely defeat any pure static analysis, including symbolic execution. The code analyzed by static analyzers is that of the emulator; the true malware logic is encoded as bytecode contained in some memory buffer that is treated as data by the analysis. At the other extreme, pure dynamic analysis based approaches that treat the emulated malware as a black box and simply observe external events are not affected. However, pure dynamic analysis schemes cannot perform fine-grained instruction level analysis and can discover only a single execution path of the malware. More advanced analysis techniques employing dynamic tainting, information flow analysis, or other instruction level analysis fall in the middle of the spectrum. In the context of malware emulators, these techniques analyze the instructions and behaviors of the generic emulator and not of the target malware. As an example, multi-path exploration [21] may explore all possible execution paths of the emulator. Unfortunately, these paths include all possible bytecode instruction semantics and all possible bytecode programs, rather than the paths encoded in the specific bytecode program of the malware instance. In short, we need new techniques to analyze emulated malware.

The key challenges in analyzing a malware emulator are the syntactic and semantic gaps between the observable (x86) instruction trace of the emulator and the non-observable (interpreted) bytecode trace. Theoretically, precisely and completely

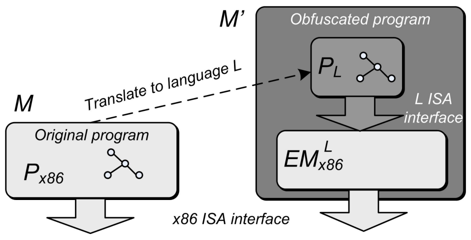


Figure 1. Using Emulation for Obfuscation

identifying an emulator's bytecode language is undecidable. Practically, the manner in which an emulator fetches, decodes, and executes a bytecode instruction may enable us to extract useful information about a bytecode program. By analyzing a malware emulator's (x86) trace, we can identify portions of the malware's bytecode program along with syntactic and semantic information of the bytecode language.

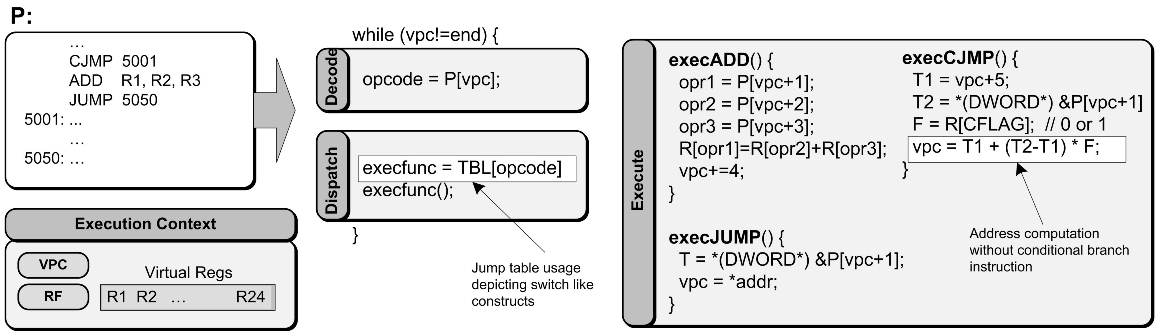
In this paper, we take the first leap toward automatic reverse engineering of unknown malware emulators. Our goal is to extract the bytecode malware trace (program) and the syntax and semantics of the bytecode instructions to enable further malware analysis, such as multi-path exploration, across the bytecode program. We have developed an approach based on dynamic analysis. We execute the malware emulator and record the entire x86 instruction trace generated by the malware. Applying dynamic data-flow and taint analysis techniques to these traces, we identify data regions containing the bytecode program and extract information about the bytecode instruction set. Our approach identifies the fundamental characteristics of *decode-dispatch emulation*: an iterative main loop containing bytecode fetches based upon the current value of a virtual program counter, decoding of opcodes from within the bytecode, and dispatch to bytecode handlers based upon opcodes. This analysis yields the data region containing the bytecode, syntactic information showing how bytecodes are parsed into opcodes and operands, and semantic information about control transfer instructions.

We have implemented a prototype called *Rotalumè* that uses a QEMU [6] based component to perform dynamic analysis. The analysis generates both an instruction trace in an intermediate representation (IR) and a dynamic control-flow graph (CFG) for offline analysis. Rotalumè reverse engineers the emulator using a collection of techniques: abstract variable binding analyzes memory access patterns; clustering finds associated memory reads, such as those fetching bytecode during the emulator's main loop; and dynamic tainting identifies the primary decode, dispatch, and execute operations of the emulator. The output of our system is the extracted syntax and semantics of the source bytecode language suitable for subsequent analysis using traditional malware analyses. We have evaluated Rotalumè on legitimate programs and on malware emulated by VMProtect and Code Virtualizer. Our results show that Rotalumè accurately identified the bytecode buffers in the emulated malware and reconstructed syntactic and semantic information for the bytecode instructions.

The main contributions of our paper are:

- We formulate the research problem of automatic reverse engineering of malware emulators. To the best of our knowledge, this is the first work in this area. Although our current work assumes a decode-dispatch emulation model, we believe that our ideas and techniques are applicable to other emulation models: by analyzing an emulator's execution trace using a given emulation model on how a bytecode instruction is fetched and executed, we can identify the bytecode region and discover the syntax and semantics of the bytecode instructions.
- We develop a framework and working prototype system that includes: a novel method to identify candidate memory regions containing bytecode, a method for identifying dispatch and instruction execution blocks, and a method for discovering bytecode instruction syntax and semantics. The output of our system can be used by existing analysis tools to analyze and extract malware behavior; for example, the identified bytecode can be converted to x86 instructions for static and/or dynamic analysis.
- Although our work is in the context of malware, we believe that this line of research will help spawn work in several other areas. For example, similar techniques may help reverse engineer script interpreters, providing novel ways to analyze scripting languages with binary analysis.

Section 2 provides a background of program emulation techniques. Section 3 provides the details of our algorithms that identify bytecode regions as well as bytecode syntax and semantics. Section 4 describes our prototype system, Rotalumè. Section 5 reports results on evaluating Rotalumè on VMProtect and CodeVirtualizer using both real-world and synthetic malware programs. Section 6 discusses open problems of reverse engineering malware emulators. Section 7 compares our work with other relevant research. Section 8 discusses future directions and concludes the paper.



```
P:
    ...
    CJMP  5001
    ADD   R1, R2, R3
    JUMP  5050
5001: ...
    ...
5050: ...
```

```
while (vpc!=end) {
    opcode = P[vpc];
    execfunc = TBL[opcode]
    execfunc();
}
```

```
execADD() {
  opr1 = P[vpc+1];
  opr2 = P[vpc+2];
  opr3 = P[vpc+3];
  R[opr1]=R[opr2]+R[opr3];
  vpc+=4;
}

execJUMP() {
  T = *(DWORD*) &P[vpc+1];
  vpc = *addr;
}

execCJMP() {
  T1 = vpc+5;
  T2 = *(DWORD*) &P[vpc+1]
  F = R[CFLAG];  // 0 or 1
  vpc = T1 + (T2-T1) * F;
}
```

Decode
Dispatch
Execute
Execution Context
VPC
RF
Virtual Regs
R1 R2 … R24
Jump table usage depicting switch like constructs
Address computation without conditional branch instruction

Figure 2. An example of a simple-interpreter (decode-dispatch) based emulator, executing program written in V

2. Background

The term emulation generally expresses the support of a binary instruction set architecture (ISA) that is different from that provided by the computer system's hardware. This section describes emulation's use in program obfuscation and the various emulation techniques possible.

2.1. Using Emulation for Obfuscation

Code authors, including malware authors, are now using emulation to obfuscate programs. In Figure 1, malicious software M consists of a program P_{x86} written in native x86 code and directly executable on x86 processors. Analyzers with knowledge of x86 can therefore perform various analyses on the malware. In order to impede analysis, an adversary can choose a new ISA L and translate P_{x86} to P_L that uses only instructions of L. In order to execute P_L on the real x86 machine, the adversary introduces an emulator EM^L_{x86} that emulates the ISA L on x86. The adversary can now spread a new malware instance M' that is a combination of P_L and EM^L_{x86}.

To further impede possible analysis, a malware author can choose a new, randomly-generated bytecode language for every instance of the malware and make tools to automatically generate a corresponding emulator. Therefore, results found about the bytecode language after reverse engineering one instance of the emulator will not be useful for another instance. Thus, automated reverse engineering of EM^L_{x86} is essential to malware analysis given that each malware instance has a completely unknown bytecode instruction set L and a previously unseen emulator instance.

2.2. Emulation Techniques

Various emulation techniques are widely used in software-based virtual machines, script interpretation, run-time abstract interfaces for high-level languages (e.g. Java virtual machine (JVM)), and other environments. Although these are very complex systems, they are variations of the simple-interpreter method, also known as the *decode-dispatch* approach [31]. Decode-dispatch is used in environments where performance overhead is not an issue. The simple interpreter utilizes a main loop that iterates through three phases for every bytecode instruction: *opcode decode*, *dispatch*, and *execute*. The decode phase fetches the part of an instruction (opcode) that represents the instruction type. The dispatch phase uses this information to invoke appropriate handling routines. The execute phase, which is performed by the dispatched handling routine, performs additional fetches of operands and executes the semantics of the instruction. We first provide an illustration of how a decode-dispatch emulator works and then discuss the other broad variations in emulation methods.

We show the design of the simple interpreter based emulators using an illustrative running example. Figure 2 shows a fraction of a simple decode-dispatch based emulator [11] written in a pseudo-C like language. The emulator executes a hypothetical bytecode language for a machine we named *V*. For conciseness, we describe only the aspects of this machine relevant to the example. This machine supports variable length instructions similar to x86. There are general purpose registers named `R1` to `R24`. A special register called `RF` maintains a flag that can be either 0 or 1 based on some previously executed instruction, and it is used for performing conditional jumps. We show three instructions supported by the machine: `ADD`, `JUMP` and `CJMP`. While the `ADD` instruction takes three operands, both jump instructions take an immediate target address. The conditional jump instruction `CJMP` jumps to the target if `RF` is 1, otherwise control flows to the next instruction.

In this example, the emulator fetches instructions from the emulated program stored in the buffer `P`. An emulator maintains a run-time context of the emulated program, which includes the necessary storage for virtual registers and scratch space. The emulator maintains execution context via a pointer to the next bytecode instruction to be executed, which we denote throughout the paper as the *virtual program counter* or *VPC*. For the example emulator, the VPC is an index into the buffer `P`. Here, *decoding* is performed by fetching the opcode from `P[VPC]`, i.e. the first byte of the instruction. *Dispatch* uses a jump table resulting from `switch-case` constructs in C. Three *execution* routines for the three instructions are shown in the example. The `execADD` routine updates the register store by adding relevant virtual register values. The `execJUMP` routine updates the VPC with an immediate address contained in the instruction. Finally, `execJMP` shows how the conditional branch updates the VPC depending on the flag `RF`. It is interesting to note that the branch is emulated without using any conditional jump, but rather with a direct address computation. This shows how an emulator provides a way to remove identifiable conditional branches, making it hard for analysis approaches such as multi-path exploration to even explore any branch related to the emulated program.

More sophisticated emulation approaches often improve efficiency. The *threaded approach* [16] improves performance by removing the central decode-dispatch loop. The decoding and dispatching logic is appended to the execution semantics by adding a copy of that code to the end of each execution routine. This removes several branches and improves execution performance on processors that have branch prediction. By using *pre-decoding* [19], the logic of decoding instructions to their opcodes and operands executes only once per unique instruction, and the program subsequently reuses the decoded results. Hence, the opcode decoding phase is not executed for each executed bytecode instruction. The *direct threading approach* [5] removes jump table lookups by storing the function address that executes the instruction semantics together with the predecoded results. Therefore, the dispatch of the next instruction's execution routine is an indirect control transfer at the end of the previous bytecode instruction's routine. Finally, *dynamic translation* [30], one of the most efficient methods of emulation, converts blocks of the emulated program into executable instructions for the target machine and caches them for subsequent execution. These categories of emulators maintain a VPC that is updated after blocks of the translated native instructions are executed.

Dynamic translation based emulators have very complex behavioral phases. They may seem attractive to malware authors because of their analysis and reverse engineering difficulty. However, like page-level unpacking used in some packers [29], dynamic translation reveals large blocks of translated code as a program's execution proceeds. This approach reduces the advantage of using emulation as an obfuscation because the heuristics used by automated unpackers can capture the fact that new code was generated and executed. In this paper, we focus our methods on automatically reverse engineering the decode-dispatch class of emulators.

Several challenges complicate automatic reverse engineering of interpreter-based emulators. First, no information of the bytecode program, such as its location, are known beforehand. Second, no information regarding the emulator's code corresponding to the decode, dispatch, and execution routines is known. Finally, we anticipate that emulator code varies in terms of how it fetches opcodes and operands, maintains context related to the emulated program, dispatches code, executes semantics, and so on. An adversary may even intentionally attempt to complicate the identification of bytecode by storing the bytecode program in non-contiguous memory or use multiple correlated variables to maintain the VPC.

Our current work, as a first step, advances the state-of-the-art and significantly challenges attackers. It also lays the foundation for reverse engineering of emulators that are based on other (more advanced and efficient) approaches.

3. Reverse Engineering of Emulation

In order to enable malware analysis of an emulated malware instance, it is necessary to understand the unknown bytecode language used by the instance. We have developed algorithms to systematically and automatically extract the syntax and semantics of unknown bytecode based upon the execution behavior of the decode-dispatch based emulator within a malware instance. Our approach identifies fundamental characteristics of decode-dispatch emulation: an iterative main loop, bytecode fetches based upon the current value of a virtual program counter, and dispatch to bytecode handlers based upon opcodes within the bytecode. This analysis yields the data region within the malware containing the bytecode, syntactic information showing how bytecode instructions are parsed into opcodes and operands, and semantic information consisting of native instructions that carry out the actions of the bytecode instructions. We identify the control-flow semantics from which structures such as a control-flow graph (CFG) can be generated. Together, these techniques provide the foundation for overcoming the emulation layer and performing subsequent malware analysis.

Our algorithms are based on dynamic analysis. We execute the emulated malware once in a protected environment and record the entire x86 instruction trace generated by the malware. From this trace, we extract syntactic and semantic information about the bytecode that the malware's emulator was interpreting during execution. The contributions made by our approach offer the opportunities to reconstruct behavioral information about unknown bytecode interpreted by an unknown decode-dispatch emulator and to subsequently apply traditional malware analysis to the sample.

The algorithms operate as follows:

1) Identify variables within the raw memory of the emulator based upon the access patterns of reads and writes in an execution trace. We developed *abstract variable binding*, a forward and backward dynamic data-flow analysis, for this identification.
2) Identify the subset of those variables that are candidates for the emulator's virtual program counter (VPC). We find possibilities by clustering the emulator's memory reads, some of which are bytecode fetches, based upon the abstract variable used to specify the accessed memory location.
3) Identify the boundaries of the bytecode data within the x86 application, the decode-dispatch loop and the emulator phases. For each cluster of reads through the same abstract variable, we determine if the reads occurred during execution of loop iteration with emulator-like operations.
4) Identify the syntax and the semantics of bytecode operations. We examine how bytecode is accessed by the emulation phases to identify the syntax. We analyze the bytecode handler or semantics for updates to the VPC. Non-sequential VPC updates indicate that the bytecode opcode corresponds to a control transfer operation. These control-transfer operations allow our system to construct a CFG for the bytecode.

The following sections describe these steps in detail.

3.1. Abstract Variable Binding

A decode-dispatch emulator fetches bytecode instructions from addresses specified by a virtual program counter (VPC). Like a program counter or instruction pointer register in hardware, a VPC acts as a pointer to the currently executing bytecode instruction. Knowing the memory location of the VPC allows an analyzer to observe how the emulator accesses bytecode instructions and executes them, which reveals information about the bytecode instruction syntax and semantics. We locate an emulator's VPC through a series of analyses, beginning with abstract variable binding.

Abstract variable binding identifies, for each memory read instruction of an execution trace, the program variable containing the address specifying the location from which the data should be read. Consider pseudo-code of an emulator that regularly fetches instructions pointed to by the VPC:

$$\texttt{instruction} = \texttt{bytecode}[\texttt{VPC}]$$

or

$$\texttt{instruction} = *\texttt{VPC} \tag{1}$$

In these examples, the VPC is an index into an array of bytecode or a direct pointer into a buffer of bytecode. During its execution, the emulator will execute these bytecode fetches many times. Although each fetch may access a different memory location within the bytecode buffer, all fetches used the same VPC variable as the specifier of the location. Abstract variable binding will attach a program variable, such as VPC, to every memory read instruction in the execution trace that uses that variable to specify its access location.

Successful abstract variable binding will help our analyzer identify the VPC and the bytecode buffer used by the unknown emulator in a malware instance. Each bytecode fetch will appear in the execution trace as a memory read instruction whose accessed location is bound to the VPC variable. The emulator likely executes many other memory reads unrelated to bytecode fetch, and these may have their own bindings to other variables in the program. Steps 2 and 3 of our algorithms, presented in Sections 3.2 and 3.3, whittle down the bindings to only those of the VPC.

Our analysis of x86 instruction traces rather than source code complicates abstract variable binding in fundamental ways. First, a binary program has no notion of high-level language variables. A compiler translating an emulator's high-level code into low-level x86 instructions will assign each variable to a memory location or register in a way unknown to our analysis. Second, the x86 architecture requires all memory read and write operations accessing dynamically computed addresses to use register indirect addressing. In case of performing a memory read using an address stored in a variable, if the variable is assigned a memory location, then that value will be transferred into the register rather than being accessed directly. For example, the pseudo-x86 translation of (1) may be the two-instruction sequence:

$$\texttt{eax} \longleftarrow [\texttt{VPC}] \tag{2}$$

$$\texttt{instruction} \longleftarrow [\texttt{eax}] \tag{3}$$

where `VPC` represents a pointer variable (assigned a particular memory address), `eax` is a register, and `instruction` is a register or memory location. The first instruction loads the value of the variable `VPC` to `eax`. This value is the address used in the second instruction where the register `eax` can be considered as a temporary place holder for the variable `VPC`. In other words, (2) binds `eax` to `VPC`, and this binding is propagated to the read operation in (3). With limited number of registers, the same register can be bound to different variables at different points of execution. In case of updating a variable with a non-immediate value, the new value must be loaded into a register using an instruction similar to (2) before transferring it to the variable's assigned memory location. In this case, the register is already bound to a variable and the binding is not changed when loading the value to the register. Without knowledge of how variables in the program are assigned to memory or registers, it is hard to determine whether a register load operation such as (2) is a new variable binding to the register or a new value assigned to an already bound variable. We draw three conclusions that impact the design of our abstract variable binding algorithm. First, we use absolute memory addresses as our description of a high-level language variable. Second, we must analyze data flows along an entire trace to determine variable binding information appropriately. Third, to be able to identify all abstract variables, we must conservatively consider both the above possible scenarios for each instruction similar to (2).

Abstract variable binding propagates binding information using dynamic data-flow analysis across an execution trace of the emulated malware. We use both a forward and a backward propagation steps, corresponding to the two different ways in which a variable's value may be set. A variable's value may be an incremental update to its previous value; forward binding identifies the abstract variables (memory locations) from which a read operation's address is derived in this case. A variable's value may also be directly overwritten with a new value unrelated to its previous contents. Backward binding determines the appropriate bindings in this case based on the variable's future use. Our backward binding algorithm is conservative and introduces imprecision.

A malicious emulator may also attempt to complicate analysis by obfuscating its use of a VPC. For example, it may replace a single VPC with a collection of variables and switch among the collection during execution. However, the collection must together still follow an orderly progression and update sequence to ensure that the emulator correctly executes the bytecode. This fundamental need to maintain consistency will produce data flows between two elements of the VPC collection whenever the emulator switches from one element to another. Our data-flow analysis will track these flows and remains robust to this type of emulator obfuscation.

We use the following notations for the presentation of our algorithms. Let M denote the memory address space of the emulator and R the set of registers available on the target hardware. We uniquely identify abstract variables by memory addresses using the set $\alpha \subseteq M$. We represent instructions in an intermediate representation that expresses only simple move and compute operations on registers and memory and has one source and one destination. Let the loading of constant c into register r be denoted as $r \leftarrow c$. A memory read operation is $r \overset{v}{\leftarrow} [m]$, which indicates that the value v is read from memory address m and loaded into register r. Here m can be a constant or register holding an address. Memory writes are denoted by $[m] \overset{v}{\leftarrow} r$. A register-to-register move operation is $r_1 \leftarrow r_2$. If the right-hand side of any of these operations involves computation using the specified variable or address, then we denote the assignment as $\twoheadleftarrow$ rather than $\leftarrow$. We identify each instruction in an execution trace with a sequence number $i \in \mathbb{N}$. Let the set $\rho \subseteq \mathbb{N}$ consist of all read operations of the form $r \leftarrow [m]$ or $r \twoheadleftarrow [m]$. Lastly, let the function $Addr : \rho \rightarrow M$ be defined as: $Addr(i)$ represents the address of a read operation $i \in \rho$.

3.1.1. Forward Binding. Forward binding identifies those variables supplying the address used by a read operation. The algorithm works on the execution trace and propagates information forward along the instructions in the trace. For each register, we track both the set of abstract variables bound to the register and the value stored in the register. A memory read instruction is bound to the same variable as that bound to the register specifying the address of the read. The set $B_i(r)$ denotes the abstract variables bound to register $r \in R$ at instruction i. $V_i(r)$ represents the value in register r at i. Forward propagation updates bindings and values according to the following six rules based upon the instruction type. For convenience, we use the function f_i to indicate the computation of instruction i.

F1 $r \leftarrow c$: $B_i(r) = \{\}$ and $V_i(r) = c$
F2 $r \overset{v}{\leftarrow} [c]$: $B_i(r) = \{c\}$ and $V_i(r) = v$
F3 $r_1 \overset{v}{\leftarrow} [r_2]$: $B_i(r_1) = \{V_{i-1}(r_2)\}$ and $V_i(r_1) = v$
F4 $r \twoheadleftarrow c$: $B_i(r) = B_{i-1}(r)$ and $V_i(r) = f_i(V_i(r), c)$
F5 $r \overset{v}{\twoheadleftarrow} [c]$: $B_i(r) = B_{i-1}(r) \cup \{c\}$ and $V_i(r) = f_i(V_{i-1}(r), v)$
F6 $r_1 \overset{v}{\twoheadleftarrow} [r_2]$: $B_i(r_1) = B_{i-1}(r_1) \cup \{V_{i-1}(r_2)\}$ and $V_i(r_1) = f_i(V_{i-1}(r_1), v)$

Rules F1–F3 apply new register value assignments where values are taken directly from a constant, register, or memory location. Bindings reset to those of the data source. Rules F4–F6 compute assigned values from the previous register value and other data; these correspond to incremental updates to a value and do not cause bindings to reset. Rules F3 and F6 correspond to reads using indirect addressing and are points where instruction bindings indicate possible VPC use.

All rules update B and V based on the current and immediate predecessor instructions, so propagation operates from the first instruction to the last. The algorithm outputs the mapping $FB : \mathbb{N} \rightarrow \alpha^*$ describing bindings from memory read operations to abstract variables. Algorithm 1 presents the pseudo-code for forward binding; l is the trace length.

Algorithm 1 Forward Binding

```
Initialize: $\forall r \in R : B_0(r) = \{\}$ and $V_0(r) = \texttt{NONE}$
for $i$ = 1 to $l$ do
    $\forall r \in R : B_i(r) = B_{i-1}(r)$ and $V_i(r) = V_{i-1}(r)$
    Update $B_i$ and $V_i$ using rules F1 to F6
    if $i \in \rho$ and instruction is $r_1 \leftarrow^v [r_2]$ or $r_1 \twoheadleftarrow^v [r_2]$ then
        $FB(i) = B_i(r_2)$
    end if
end for
```

3.1.2. Backward Binding. We expect realistic bytecode languages to provide one or more types of control transfer operations such as jumps and branches. Forward binding propagates abstract variable bindings when operations compute an incremental update to an already-bound variable, which does not reflect the semantics of a control transfer. Control transfers will instead cause the emulator to directly overwrite the VPC with the target address. The emulator will execute instructions matching rule F1 or F2, which reset the binding information. Backward binding, a second variable binding step that follows the forward algorithm, computes bindings in such cases by identifying bound variables when a value is *written out* to memory and then propagating the binding backward to all previous uses. This algorithm is conservative and may over-approximate the actual variable bindings.

Backward binding operates in the reverse order of forward binding. It first binds a variable to a register when that register's value is stored to memory, and then propagates the binding backwards to other registers using the following rules:

B1	$[r_1] \leftarrow r_2 : B'_i(r_2) = \{V_i(r_1)\}$
B2	$[c] \leftarrow r : B'_i(r) = \{c\}$
B3	$r_1 \leftarrow r_2 : B'_i(r_2) = B'_{i+1}(r_1)$

The algorithm outputs the mapping for backward binding $BB : \mathbb{N} \rightarrow \alpha^*$. Algorithm 2 presents pseudo-code for backward binding.

Algorithm 2 Backward Binding

```
Initialize: $B'_{l+1}(r_j) = \{\sigma_{r_j}\}$ for $1 \leq j \leq k$
Generate $V_i(r_j)$ for $1 \leq j \leq k$ and $1 \leq i \leq l$ using Algorithm 1
for $i$ = $l$ to 1 do
    $\forall r \in R : B'_i(r) = B'_{i+1}(r)$
    Update $B'_i$ using rules B1 to B3
    if $i \in \rho$ and instruction is $r_1 \leftarrow^v [r_2]$ or $r_1 \twoheadleftarrow^v [r_2]$ then
        $BB(i) = B'_i(r_2)$
    end if
end for
```

3.1.3. Identifying Dependent Abstract Variables. Variables used to contain memory read addresses may themselves be interdependent. For example, if the obfuscation technique of utilizing a collection of VPCs is used, the VPC related variables have a relationship with each other. Likewise memory reads that access operands may occur at short, fixed offsets from the VPC value. We identify dependencies among abstract variables by tracking the flow of data values from one abstract variable to another. As this is forward data-flow, we incorporate dependent variable identification as part of our forward binding algorithm.

Let the mapping $DV : \alpha \rightarrow \alpha^*$ denote abstract variable dependencies, where $Y \in \alpha$ is a dependent variable of X if $Y \in DV(X)$. To populate this mapping, we add the following rule to Algorithm 1:

D1 $[r_1] \leftarrow r_2 : DV(r_1) = DV(r_1) \cup \{B_i(r_2)\}$

Dependencies are commutative. After the completion of Algorithm 1, we add the converse of each dependency identified by the algorithm: $\forall X, Y \in M$: if $X \in DV(Y)$ then add $Y \in DV(X)$. Identifying these abstract variable dependencies thwart attacks that introduce extraneous store operations or copy operations from one variable to another before use.

3.1.4. Lifetime of Variables. Our x86 execution trace analysis introduces one final challenge that is often not present when using high-level language variables. We use absolute memory locations as our abstract variables, but the same memory location may be used for different variables at different points of execution. Although variables in the static data region have the lifetime of the entire execution, variables on the stack and heap have shorter lifetimes. The same address can be shared among multiple variables in different execution contexts depending on the allocation and deallocation operations performed during execution.

We address the limited lifetime of stack variables by including stack semantics and analysis of the stack pointer register `esp` as part of our algorithms. Our set of abstract variables α is made of tuples $\alpha \subseteq M \times \mathbb{N} \times \mathbb{N}$ that use integers to denote the start and end of the variable's live range within the execution trace. At the first access to a memory address $m \in M$ at instruction s, we add (m, s, ∞) to α. If the t^{th} instruction modifies the `esp` register such that an abstract variable's memory address has been deallocated from the stack, then the end of its lifetime is set to t. Any access to the same memory address after its lifetime expired creates a new abstract variable. For pedagogy, we presented Algorithms 1 and 2 without live ranges; updates to the algorithms to include lifetimes are straightforward.

In this work, we do not address lifetimes for variables allocated on the heap. Prior heap analysis research [3] often assumed that the analyzer understood the heap allocation and deallocation routines. We cannot make this assumption for malware binaries, which may be stripped of debugging information and deliberately obfuscate the heap routines. Further research is needed to address this open problem.

3.2. Identifying Candidate VPCs

We use the computed variable bindings to identify candidate variables that may be the malware emulator's virtual program counter. We first combine the bindings identified by the forward and backward algorithms to compute the complete abstract variable bindings for each memory read. Let the function *Vars* : $\mathbb{N} \rightarrow \alpha^*$ be computed as the transitive application of the dependence function DV to $FB(i) \cup BB(i)$ for a read operation $i \in \rho$. We then cluster all read operations within the execution trace and group together those reads that are bound to common abstract variables. Our clustering uses a simple similarity metric that treats two reads $i_1, i_2 \in \rho$ as similar if $Vars(i_1) \cap Vars(i_2) \neq \emptyset$, and dissimilar otherwise. The clustering algorithm will output n clusters $C_1, \ldots, C_n$ where each cluster C_i is a set of read operations.

The malware bytecode should be fetched for execution exclusively by memory read operations contained within one of the n clusters. Abstract variable binding over-approximates actual bindings due to the backward algorithm, which results in two reads clustered together if they *may* use the same abstract variable to specify the accessed address. The transitive closure of the dependencies among abstract variables ensures two reads will be similar even if the reads use two distinct abstract variables. Therefore, the bytecode program will be completely contained within a cluster. Each cluster is then a candidate collection of instruction fetches into bytecode, and the common abstract variables at each cluster are candidate VPCs.

3.3. Identifying Emulation Behavior

We analyze each candidate cluster and VPC to find a cluster containing memory reads characteristic of emulation. Decode-dispatch emulators have fundamental execution properties: a main loop with a bytecode fetch through the VPC, decoding of the opcode within the bytecode, dispatch to an opcode handler, and a change to the VPC value. For each candidate cluster, we hypothesize that the memory region read by the cluster corresponds to bytecode and then test that hypothesis. We determine whether there exists an iterative pattern of bytecode fetches through the associated candidate VPC and updates to that possible VPC. To detect loops, we first create a partial dynamic control-flow graph (CFG) of the program in execution. We use the control-flow semantics of the executed instructions to create new basic blocks and split already created blocks. We use function call semantics to create separate CFGs for each function. Then, we use the standard loop detection methods used for static intra-procedural CFGs [1].

To find decoding, dispatching, and execution of bytecode after the memory read fetches it from the bytecode buffer, we analyze how read values are used by other instructions within the execution trace. We use multi-level dynamic tainting [38] to track the propagation of the data read from instructions in a candidate cluster through the emulator's code. In contrast to traditional taint analysis with 0/1 taint labels, we apply multiple labels to memory contents and registers at the byte level.

Different labels track individual data read from the cluster and maintain state information related to which phase—fetch, decode, dispatch, or execute—that the emulator may be in for a particular read.

We use dynamic taint analysis as follows. For each candidate cluster, we taint the data bytes in the hypothesized bytecode buffer region of the cluster with the label $\langle opcode, id \rangle$ where an id is a unique per-byte identifier. When a read operation in the execution trace accesses a tainted byte, we mark the instruction as an *opcode fetch* for the particular id in the label. If the instruction sequence number is i, then we also taint the forward bound variables $FB(i)$ and the register holding the address accessed by the read operation with the label $\langle vpc, id \rangle$, indicating that it is a VPC for the emulator. Execution continues until our analyzer detects opcode dispatch behavior.

We identify dispatch behavior by looking for control-flow transfer instructions executed by the emulator that are influenced by data read from the cluster's hypothesized bytecode buffer: these are transfers into handlers for specific bytecode opcodes. In the simplest scenario, an x86 instruction like `jmp` or `call` can target an address read from a tainted register or from a dispatch table accessed through a tainted register. More complex code patterns may include arbitrary data and control flows between a control-flow target lookup and the actual dispatch. Taint propagation ensures that taint labels transfer from address to values read through that address, to copies of those values, and to the control-flow transfer. Once detecting a dispatch-like behavior, the analyzer marks the dispatch instruction with the id of the taint label and the analysis now tracks the target of the control-transfer as a probable *execute* routine.

Each subsequent read in the candidate cluster may be accessing a new bytecode, operands for the current bytecode, or an unrelated memory value included in the cluster due to the imprecision of backward variable binding. We first identify new bytecode accesses by analyzing the dynamic CFG to see if execution looped since the previous bytecode fetch. If a loop is not detected, we then check to see if the read is accessing a probable operand in the bytecode buffer. If the register used to perform the read operation is tainted as $\langle vpc, id \rangle$ with the id of the current iteration, and the computation of the accessed address added a small constant to the candidate VPC value, then the memory access is likely that for an operand. We consider all other accesses to be spurious.

We consider every candidate cluster containing iterative memory reads in a loop that includes dispatch behavior. There must be at least two loop executions in the dynamic trace for our analysis to identify the loop.

3.4. Extracting Syntax and Semantics

Once the analyzer identifies the emulation behavior, it reverse engineers each iteration of the emulator loop to extract the syntax and semantics of the bytecode instruction executed on that iteration. The syntax of bytecode details how to parse the instruction: its length and the placement of its opcode and operands. Bytecode semantics describe the bytecode's effect upon execution of the malware instance. We are particularly interested in identifying bytecode instructions exhibiting control-flow transfer semantics, as these are the locations where malware analysis techniques such as multipath exploration [21] should be applied.

We identify the syntax of bytecode instructions by observing the memory reads made from the data regions containing bytecode, as determined in Section 3.3. To identify the opcode part of the instruction, we apply our taint analysis to determine which portion was used by the emulator's dispatch stage for selection of an execution handler. We can identify opcodes at the granularity of one or more bytes within a bytecode instruction, as our taint analysis works at byte-level. An emulator may dispatch several different opcodes to the same execution routine because their semantics may be similar. As a result, we count the number of instructions in the bytecode instruction set as the number of unique execution routines identified in our analysis.

The execution routine invoked by the emulator for the bytecode's opcode encodes the semantics of the opcode. We find control flow transfers by analyzing the changes made by an execution routine upon the VPC of the emulator. Unconditional transfers, including fall-through instructions, will always set the VPC to the same value on every execution of that instruction. Commonly, fall-throughs simply advance the VPC to the next instruction in sequential order, a regular update pattern that can be readily identified. Conditional control-flow transfers and transfers to dynamically-computed targets will update the VPC in different ways upon repeated execution of the bytecode instruction.

By determining how to parse the bytecode buffer and by locating control-flow transfer opcodes, we are then able to construct a control-flow graph (CFG) for the bytecode. The locations of the control-flow transfers and their target addresses within the bytecode stipulate how to divide the entire bytecode buffer into basic blocks. The transfers then produce edges between the blocks corresponding to possible VPC changes during emulated execution. The CFG structure provides a foundation for subsequent malware analysis.

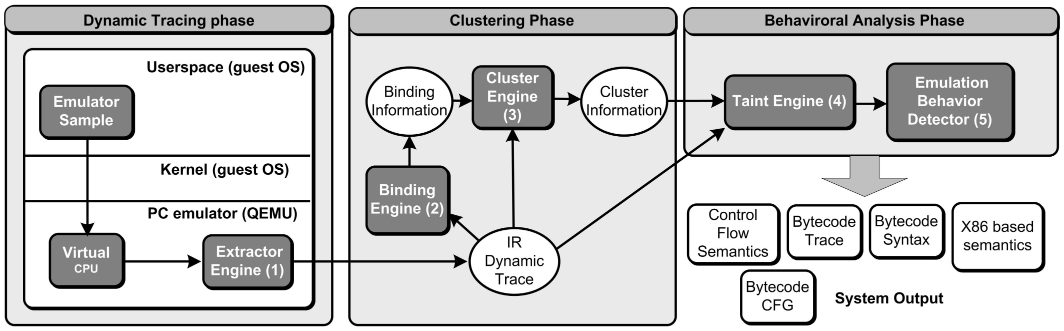


Figure 3. Analysis Process Overview

4. Implementation

Our automatic reverse engineering occurs in three different phases: *dynamic tracing*, *clustering*, and *behavioral analysis*. Figure 3 shows the different phases of our process and the interactions among the architectural components used by the different analysis steps. The dynamic tracing phase gathers run-time data related to a malware emulator's execution, and allows the clustering and behavioral analysis phases to extract malware bytecode and the syntactic and semantic information for the bytecode instruction set.

There are two important requirements for the run-time environment of the dynamic tracing phase: instruction-level tracing, and isolation from malware and attacks. Since the analysis techniques in Rotalumè are orthogonal to the underlying run-time environment and our goal here is to develop and evaluate these techniques, we implemented our dynamic analysis techniques on top of QEMU [6], which emulates an x86 computer system. For a deployable version of Rotalumè, we suggest using a more transparent and robust environment, such as a hardware virtualization based system like Ether [12]. The components in the latter two phases were developed as an offline analyzer written in C++. In our current prototype implementation, each individual phase is activated manually using the result of the previous phase. However, our design can be completely automated to process large numbers of malware samples.

4.1. Dynamic Tracing

The first phase collects the dynamic instruction trace of the emulator program that is executing as the QEMU guest operating system. We modified QEMU by inserting a callback function that invokes Rotalumè's *Trace Extractor Engine* (EE) for every instruction executed in QEMU. The EE component collects necessary context information related to the executed instruction and stores the intermediate-representation (IR) that is used in latter phases of the system. Our IR is self-contained—we store the instruction representation as well as the values of the operands involved in the instruction. We log all information so that we may perform off-line analysis without requiring additional dynamic analysis. The output information of this phase is represented by the dynamic trace of the program in IR form.

4.2. Clustering

The second phase clusters the memory read operations visible in the trace. We group together every read operation performed by the program based on the common variable used to access that read memory location. This phase is performed by two main components: the *Binding Engine* (BE) and *Clustering Engine* (CE). The BE component is a program that takes as input the IR dynamic program trace and applies the *backward* and *forward* abstract variable binding algorithms described in Section 3.1. For each algorithm, we store binding information differently. More specifically, for each instruction in forward binding, we store the following information: *instruction id* (a unique identifier for each instruction present in the dynamic program trace IR), the *destination register operand* of the instruction, and the bound variables associated with the destination register according to the rules described in Section 3.1.1. For each instruction in backward binding, we store the *instruction id* and the *bound variables* associated with the registers or memory locations according to the rules defined in Section 3.1.2. The BE component provides the binding information to the CE. The CE component is a program that inputs the IR dynamic trace and the binding information, and applies the clustering algorithm. At a high level, CE takes the union of forwarding and backward binding information, applies the dependence function in Section 3.2, and provides the cluster information. The cluster information contains a vector of sets where each set contains the addresses of the memory read instructions that are accessed by the same variable. At the end of this phase, the cluster information is saved to a file.

4.3. Behavioral Analysis

The behavioral analysis phase provides the final information output of Rotalumè. We implemented a behavioral analyzer composed of two sub-components: the *Taint Engine* and the *Emulation Behavior Detector*. The behavioral analyzer is a program that takes as input the IR dynamic trace and clustering information, and analyzes one cluster at a time. For each cluster, the Taint Engine taints the memory address contained in the cluster and activates the Emulation Behavior Detector. This analyzer is a state machine that follows the tainted addresses and identifies the emulation behavior, as described in Sections 3.3 and 3.4. Whenever the analyzer recognizes an opcode, the system stores information of the opcode into a file. More specifically, the analyzer stores for each opcode executed: its opcode value, the operands' values, and the x86 code in assembly format associated with the executed opcode.

5. Evaluation

We evaluated Rotalumè using both synthetic and real programs that include both legitimate applications and malware. These programs are obfuscated to run on three commercially available packers that support emulation: *Code Virtualizer* [23], *Themida* [24], and *VMProtect* [35]. VMProtect and Code Virtualizer convert selective functions of a given binary program into a bytecode program with a randomly generated bytecode language. Themida, which is more widely used for malware, does not apply emulation to the given malicious binary program but rather to the unpacking routine and the code that invokes API calls.

5.1. Synthetic Tests

We first experimented with synthetic test programs. Our goal was to use the ground truth of the synthetic programs to evaluate the information about the extracted bytecode program and the syntax and semantics of the virtual instruction set architecture identified by Rotalumè. We used Code Virtualizer and VMProtect because they can obfuscate any user-specified function in a program.

Table 1. Description of Synthetic Test Programs

Function	Description	x86 Program		x86 Trace	
		Inst.	C-Flow	Inst.	C-Flow
Synth1	No branch	24	1	24	1
Synth2	Nested `if`	61	11	21	7
Synth3	Loop and `if`	55	10	270	54

We wrote three simple synthetic test programs in C. Each test program contained a function with distinguishable control-flow characteristics that we wanted to obfuscate. We compiled these programs and converted them to x86 binaries. We analyzed the static characteristics of the compiled code using IDAPro [13] and the dynamic characteristics by tracing the programs in our QEMU-based system. Table 1 lists information about the three functions of these test programs. For each function, the table shows the total numbers of x86 instructions ("Inst.") and control-flows instructions ("C-Flow") obtained from static analysis. The total numbers of x86 and control-flow instructions in an execution trace of the functions, obtained from dynamic analysis, are also shown. `synth1` involves simple computation without any conditional branch or function. `synth2` contains nested `if` statements, and hence its execution trace contains only a part of its (static) program instructions. Finally, `synth3` contains both `if` statements and a `for` loop. Its trace length was larger than the static x86 instruction count because of loops.

We used VMProtect and Code Virtualizer to obfuscate the selected functions in our three test (binary) programs. We then applied Rotalumè to analyze them. Rotalumè was able to correctly identify emulation behavior in all of the test cases, and Tables 2 and 3 summarize respectively the results of reversing Code Virtualizer and VMProtect. The results show information for bytecode instructions traced and identified at run-time in terms of the instruction counts (of all types and the control-flow instructions) of the bytecode execution trace and the program itself. The results also show the virtual instruction set architecture (ISA) discovered by Rotalumè in terms of the number of unique bytecode instructions, information regarding

Table 2. Results of Synthetic Programs Obfuscated with Code Virtualizer

Subject	Bytecode Trace (inst. count)		Bytecode Program (inst. count)		Virtual Instruction Set Architecture				
	All types	C-Flow	All types	C-Flow	All types	C-Flow (Cond.)	0 Opr	1 Opr	2 Opr
Synth1	277	1	277	1	23	1 (0)	3	6	15
Synth2	254	7	254	7	27	3 (1)	4	8	16
Synth3	3481	54	684	8	31	3 (1)	4	9	18

Table 3. Results of Synthetic Programs Obfuscated with VMProtect

Subject	Bytecode Trace (inst. count)		Bytecode Program (inst. count)		Virtual Instruction Set Architecture				
	All types	C-Flow	All types	C-Flow	All types	C-Flow (Cond.)	0 Opr	1 Opr	2 Opr
Synth1	497	1	497	1	16	1 (0)	4	8	4
Synth2	442	7	442	7	18	2 (0)	4	9	5
Synth3	5709	54	785	8	18	2 (0)	4	9	5

the syntax of the bytecode language in terms of number of operands, and information regarding the semantics of conditional control-flow transfers.

In both VMProtect and Code Virtualizer, the bytecode trace of a program was significantly longer than its original x86 binary. For example, `synth3` executed 3,481 bytecode instructions of Code Virtualizer and 5,709 of VMProtect, compared to just 270 x86 instructions in the original program. The results also show that for all test cases, Rotalumè accounted for the same number of control-flow instructions in the bytecode execution trace as in the original x86 execution trace. This shows that Rotalumè was able to extract the control-flow information of the original programs.

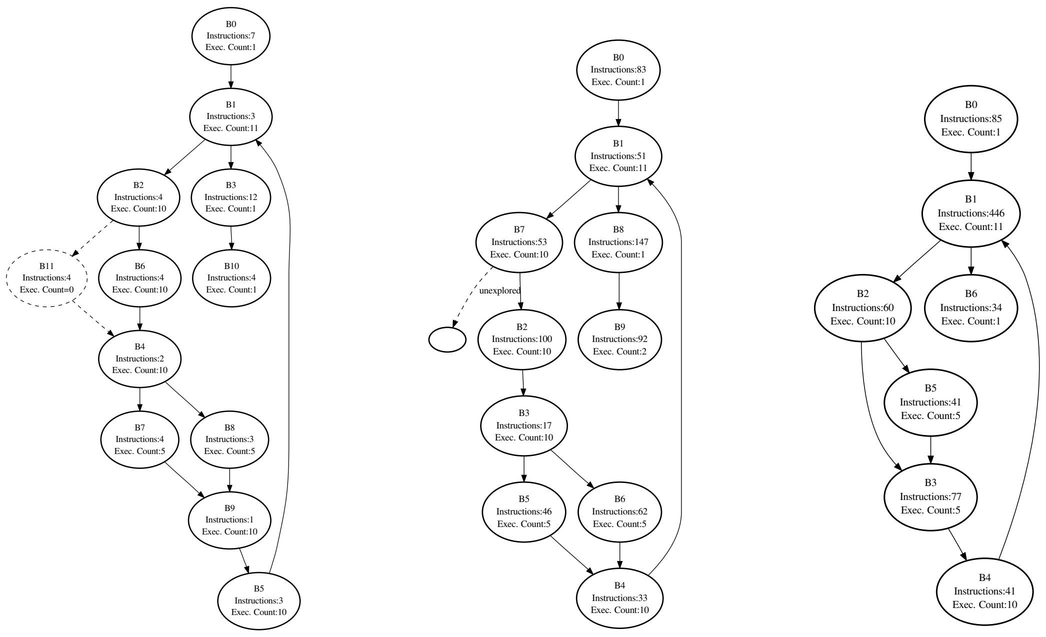
Table 3 shows that for both `synth2` and `synth3`, the VMProtect virtual ISA extracted by Rotalumè does not have conditional control-flow instructions, unlike the results from Code Virtualizer. We investigated this discrepancy by analyzing the x86 execution traces of VMProtected software and then comparing with the bytecode information provided by Rotalumè. We found that Rotalumè correctly identified the decode, dispatch, and execution routines of the emulator. We manually analyzed the traces of the execution routines and did not find any x86 conditional branch instruction. This means that there were no conditional jumps in the bytecode program traces. By carefully analyzing the semantics of the instructions before the control-transfer instruction, we confirmed that VMProtect emulates conditional branches by dynamically computing the target address and using a single jump instruction.

Figure 4 shows that the control-flow graphs extracted by Rotalumè for `synth3` are very similar to that of the original x86 program. Figure 4(a) shows the original x86 program's CFG, and it contains a loop with two conditional branches. The graph shows that basic block `B12` was not executed during execution. Figure 4(b) shows the CFG of the Code Virtualizer bytecode program trace as extracted by Rotalumè. The two CFGs show identical control-flow semantics. Interestingly, we also could identify that there is an unexplored path from basic block `B7`. This was possible because Code Virtualizer's bytecode language has a conditional branch instruction that was identified by Rotalumè even though it was not executed. This shows a key benefit of our approach: other analyses such as multipath exploration [21] can be selectively applied to explore such paths in the emulated malware bytecode rather than in the entire emulator.

The CFG in Figure 4(c) is for the VMProtect bytecode trace. Since we found that VMProtect's bytecode has no explicit conditional branches, we are unable to provide information about a possible path that was not executed in the trace. However, we can identify the dynamically computed control-flow instructions in the trace and mark where analysis of possible branch target addresses needs to be applied. Thus, we can still uncover the control-flow information of the bytecode program. The CFG shows the existence of the loop and the condition but the number of basic blocks is fewer than the original CFG. This likely occurs because VMProtect applies optimization on the bytecode.

5.2. Real (Unpacked) Programs

We next tested on a real program obfuscated with emulation by comparing the extracted bytecode information against the original x86 program. We selected a malware program that is not packed because self-modifying code can not be translated into bytecode. We randomly selected the *Killav.PS* malware identified as a Trojan by *Avira Antivir* antivirus software [2]. We then applied VMProtect on the binary. We were unable to use Code Virtualizer on this real software because Code Virtualizer requires a `.map` file, which is usually generated at compile time and hence not available with malware. Table 4 shows the results of using Rotalumè with various levels of obfuscation applied to the binary.


B0 Instructions:7 Exec. Count:1
B1 Instructions:3 Exec. Count:11
B2 Instructions:4 Exec. Count:10
B3 Instructions:12 Exec. Count:1
B11 Instructions:4 Exec. Count=0
B6 Instructions:4 Exec. Count:10
B10 Instructions:4 Exec. Count:1
B4 Instructions:2 Exec. Count:10
B7 Instructions:4 Exec. Count:5
B8 Instructions:3 Exec. Count:5
B9 Instructions:1 Exec. Count:10
B5 Instructions:3 Exec. Count:10
B0 Instructions:83 Exec. Count:1
B1 Instructions:51 Exec. Count:11
B7 Instructions:53 Exec. Count:10
B8 Instructions:147 Exec. Count:1
unexplored
B2 Instructions:100 Exec. Count:10
B9 Instructions:92 Exec. Count:2
B3 Instructions:17 Exec. Count:10
B5 Instructions:46 Exec. Count:5
B6 Instructions:62 Exec. Count:5
B4 Instructions:33 Exec. Count:10
B0 Instructions:85 Exec. Count:1
B1 Instructions:446 Exec. Count:11
B2 Instructions:60 Exec. Count:10
B6 Instructions:34 Exec. Count:1
B5 Instructions:41 Exec. Count:5
B3 Instructions:77 Exec. Count:5
B4 Instructions:41 Exec. Count:10

(a) Original x86 CFG (b) Bytecode CFG (Code Virtualizer) (c) Bytecode CFG (VMProtect)

Figure 4. Comparison of x86 and bytecode CFGs of the Synth3 test program

Table 4. TR/Killav.PS obfuscated by VMProtect

Description	Dyn. x86 CFG Inst. (BB)	Dyn. BC CFG Inst. (BB)
Original	1528 (435)	×
1 function packed	3618 (738)	2617 (16)
5 functions packed	4103 (801)	3830 (49)

We selected one large function in the malware and used VMProtect to convert it into bytecode. The table shows that the x86 code size grows after obfuscation because the new binary additionally contains the emulator's code. Rotalumè extracted the bytecode trace and the dynamic CFG of the obfuscated function. We compared the results with the original x86 version of the obfuscated function. Although the bytecode version had only 16 basic blocks compared to the 24 blocks of the original (not shown here; the table instead shows the size of the whole binary including the emulator), the control-flow attributes were very similar. Figure 5 in Appendix A shows the graphs of the two functions side by side for comparison. From the CFGs, it seems that some basic blocks may have been combined due to code optimizations performed on the bytecode by VMProtect, but similar loops and branches were identifiable. This shows that Rotalumè indeed correctly extracted the bytecode syntax and semantics. We tested another obfuscated version of the malware where we selectively obfuscated four additional functions. In that case, the x86 code increased less substantially, and Rotalumè successfully extracted the bytecode syntax and semantics of those functions.

Table 5. Tests on `CMD.EXE` obfuscated by VMProtect

Description	Dyn. x86 CFG Inst. (BB)	Dyn. BC CFG Inst. (BB)
Original	8458 (1143)	×
1 function packed	10429 (1345)	3488 (31)
5 functions packed	10512 (1394)	12345 (103)

Finally, we tested Rotalumè after applying emulation to a legitimate program. Using `CMD.EXE`, we performed experiments similar to those for the unpacked malware. Table 5 presents the results. The bytecode CFG that we obtained contained 31 basic blocks instead of the 36 in the original function. Figure 6 in Appendix B shows the control-flow graphs of a large function of CMD.EXE. We show the original x86 code's CFG in Figure 6(a) and the bytecode version extracted by Rotalumè from the VMProtect obfuscated sample in Figure 6(b). We found that some parts of the graphs matched perfectly, with differences in other parts likely due to code transformation and optimization differences.

5.3. Emulated Malware

We next evaluated Rotalumè on real malware samples that use emulation based packers. We selected samples that are packed with Themida, VMProtect, and Code Virtualizer, the three known commercial packers that use emulation. We have access to thousands of malware samples, from which we identified the ones packed using these three tools. We then applied Rotalumè to a randomly selected set of these malware samples.

Table 6. Malware Packed With Themida

Description	x86 Trace	Dyn x86 CFG Inst. (BB)	BC Trace	Dyn BC CFG Inst. (BB)
Themida5	15.3M	9753 (2156)	15232	3421 (57)
Themida1	1.4M	5961 (1156)	1339	1339 (6)
Themida3	14.8M	10211 (2125)	2142	2142 (15)
Themida7	21.4M	14205 (3529)	5171	3042 (28)
Themida8	3.5M	6011 (2125)	1534	1534 (9)
Themida11	11.1M	9021 (2925)	1784	1784 (10)
Themida13	11.4M	10211 (3194)	19642	4142 (65)
Themida14	17.3M	11492 (2877)	14219	3751 (75)

Among the three obfuscators, Themida is the most widely used within our malware samples. Themida is known not to emulate the code of the original program but rather the unpacking routine. Nevertheless, we wanted to evaluate whether Rotalumè can reverse engineer the emulator. Table 6 shows the results of Rotalumè's output on 8 randomly selected samples. We obtained the names of these samples by submitting them to VirusTotal [34] and selecting the name that was most common among the AV tools. For each sample, we gathered the execution trace when running it for 20 seconds. The table first shows the length of the x86 trace as well as the counts of instructions ("Inst.") and basic blocks ("BB") in the dynamically created CFG of the x86 code. Our analyzer was able to detect the emulator in all cases: the table shows information for the extracted bytecode trace, and we built the control-flow graph using the extracted control-flow semantics of the bytecode language. We do not show the syntax and semantic information of the bytecode instruction set here because we found that the instruction sets consistently contain 31 instructions. However, the syntax of the instructions varied, showing that the instruction sets were highly randomized. We also manually analyzed the instruction set and observed that the semantics were very close to that from Code Virtualizer. This is not surprising given that both tools are from the same vendor [22] In all samples, the x86 CFG is very large compared to the corresponding bytecode CFG. Again, this shows that Themida was not designed to obfuscate a program completely with emulation.

Table 7. Malware Packed With VMProtect

Description	x86 Trace	Dyn. x86 CFG Inst. (BB)	BC Trace	Dyn. BC CFG Inst. (BB)
Win32.KGen.bxp	3.1M	2122 (591)	1112	1112 (9)
Win32.KillAV.ahb	1.4M	4104 (1156)	1231	1231 (12)
Graybird	131K	823 (275)	2926	1584 (18)
Win32.Klone.af*	5.0M	4263 (707)	1241	1241 (17)
Win32.Klone.af*	3.2M	4123 (484)	1149	1149 (14)

We then experimented with a group of randomly selected samples that use VMProtect. We present the results of 5 samples in Table 7. Rotalumè was able to detect the emulation process in each of the samples and produced the syntactic and semantics information of the bytecode language, the bytecode trace, and the CFG. Rotalumè identified 18 bytecode instructions in the instruction set for each case. This matched the output from the synthetic test samples `synth2` and `synth3` in Table 3. Interestingly, the syntax was also the same. Like the Themida samples, these samples had very small amounts of code emulated. We conjecture that the malware authors likely had used the demo version of VMProtect, which only allows conversion of one function of the binary into bytecode.

Table 8. Malware Packed With Code Virtualizer

Description	x86 Trace	Dyn x86 CFG Inst. (BB)	BC Trace	Dyn BC CFG Inst. (BB)
Win32.Delf.Knz*	7.0M	2249 (608)	114526	10054 (343)
Win32.Delf.Knz*	15.5M	2594 (720)	234012	25221 (742)
Win32.Delf.Knz*	14.5	2531 (738)	215892	19850 (771)

We also experimented with recent malware samples that use Code Virtualizer. Table 8 shows the results. All of the samples were identified with the same name in VirusTotal even though their program sizes and MD5 checksums varied. After analyzing these malware samples with Rotalumè, we found that unlike samples we tested with Themida and VMProtect, these samples have large portions of their code converted into bytecode. The bytecode CFGs of these programs varied significantly, showing that they may be quite different programs even though they share the same name.

6. Discussion

In this section, we discuss three open problems and challenges: alternative emulator designs, incomplete bytecode reconstruction, and code analysis limitations.

First, our current work assumes a decode-dispatch emulation model, thus, malware authors may implement variations or alternative approaches to emulation [5], [16], [19], [30] to evade our system. For example, our loop identification strategies of Section 3.3 are not directly applicable to malware emulators using a threaded approach. However, the methods of identifying the candidate bytecode regions and VPC's are still applicable. As discussed in Section 2.2, our approach is not applicable to dynamic translation based emulation as well. In dynamic translation, the emulator dynamically generates new code that the program subsequently executes, thus, we expect that heuristics used by unpackers to detect unpacked code will be able to detect the translated instructions. The translated native code should provide opportunities to trace back to the translation routines and then utilize our methods for identifying bytecode region and the VPC context. More generally, we believe that our fundamental ideas and techniques are applicable to other emulation models: by analyzing an emulator's execution trace using a given emulation model, we can identify the bytecode region and discover the syntax and semantics of the bytecode instructions. The main challenge in future research is to identify observable and discernible run-time behavior exhibited by sophisticated emulation approaches.

Malware using decode-dispatch emulation may attempt to evade accurate analysis by targeting specific properties of our analysis. For example, since our approach expects each unique address in memory to hold only one abstract variable, an adversary may utilize the same location for different variables at different times to introduce imprecision in our analysis. Our system will put the memory reads performed using these variables into the same cluster due to the conservativeness of our analysis. If the additional data included in the cluster containing the bytecode program is used in decode or dispatch-like behavior, they may be incorrectly identified as bytecode instructions.

The second open problem is how to reconstruct complete information about the bytecode instruction syntax and semantics, so that a system can extract the entire emulated malware bytecode program. Using dynamic analysis, we extracted execution paths in the bytecode program and the syntax and semantics of the bytecode instructions used in those paths. However, the paths may not have utilized all of the possible bytecode instructions supported by the emulator, though they may be used in other execution paths of the program. A plausible approach would apply static analysis on the dispatch routine once our system has identified the emulation phases correctly. More specifically, once the dispatching method is identified, static analysis and symbolic execution may identify other execution routines and the opcodes of the bytecode instructions that invoke their dispatch. This provides the syntactic and semantic information of the bytecode instructions even though they are not part of the executed bytecode.

A subsequent open problem is utilizing the discovered syntax and semantics to completely convert bytecode to native instructions. A solution is possible only when all execution paths of the bytecode program can be explored. A potential solution is to use previous techniques employed for multi-path exploration [21] with the help of discovered control-flow semantics of the bytecode. However, emulators may be written so that specific control-flow semantics need not be supported in the bytecode language. Such is the case for VMProtect, where we have only identified unconditional branches. In such bytecode languages, the effects of conditional branches are performed in the program by dynamically computing the target address based on the condition and then using an unconditional branch to the specific target (an example was provided in Figure 2). More research is required before multi-path exploration can be applied to programs written in such languages.

Another related problem is the use of recursive emulation, which converts the emulator itself to another bytecode language and introduces an additional emulator to emulate it. The recursive step can be performed a number of times by a malware author, with size and performance increases as the limiting factors. The solution is to first apply our reverse engineering method to the malware instance, use the discovered syntax and semantics to completely convert the bytecode program into native binary code, and then apply our method (recursively) on the converted program to identify any additional emulation-like behavior.

Third, as with all program analysis tasks, reverse engineering of emulators also faces the challenges of heap analysis imprecision, limitations of loop detection, and so on. The techniques to address these problems are orthogonal to our techniques in reverse engineering.

7. Related Work

Malware authors have developed obfuscation schemes designed to impede static analysis [8], [18], [25], [26]. Dynamic analysis approaches that treat malware as a black box can overcome these obfuscation schemes, but they are able to observe only a small number of execution paths. Several approaches have been proposed to address this limitation. Moser et al. proposed a scheme [21] that explored multiple paths during malware execution. Another approach [36] forces program execution along different paths but disregards consistent memory updates. Rotalumè, these solutions are unable to properly analyze emulated malware because they will explore execution paths of the emulator rather than that of the bytecode program.

Malware authors have broadly applied packing to impede and evade malware detection and analysis. Several approaches based on the general unpacking idea have been proposed [14], [20], [27]. For example, Polyunpack performs universal unpacking based on a combination of static and dynamic binary analysis. Given a packed executable, Polyunpack first constructs a static view of the code. If the executable tries to execute any code that is not present in the static view, Polyunpack detects this as unpacked code.

Recently we observed a new trend in using virtualizers or emulators such as Themida [24], Code Virtualizer [23], and VMProtect [35] to obfuscate malware. These emulators all use a basic interpretation model [31] and transform the x86 program instructions into its own bytecode in order to hide the syntax and semantic of the original code and thwart program analysis. Moreover, by using a randomized instruction set for the bytecode language together with a polymorphic emulator, the reverse engineering effort will have to be applied to every new malware instance, making it very difficult to reuse the reverse engineered information of one emulator for another. We argue that this trend will continue and that a large portion of malware in the near future will be emulation based. There is no existing technique that can reliably counter an emulation-based obfuscation technique.

Researchers have proposed using a randomized instruction set with emulation as a software defense against code injection attacks. Kc et al. [15] and Barrantes et al. [4] developed approaches that converted a binary program's native instructions into a per-process randomized instruction set. Since an adversary trying to exploit vulnerabilities will not have knowledge about the random instruction set, injected code will not run properly and will cause the program to crash. Subsequent work by Sovarel et al. [32] discussed the effectiveness of instruction set randomization techniques against various attacks.

There are research approaches for analysis and reverse engineering of bytecode for high-level languages such as Java [9], [33]. However, these approaches assume that the syntax and semantics of the bytecode are public or already known. This assumption fails to hold for malware constructed using emulators such as Themida, CodeVirtualizer, or VMProtect [23], [24], [35]. These emulators perform a random translation from bytecode to destination ISA, so the connection between the bytecode and final ISA is unknown.

In order to overcome these emulation-based obfuscation techniques, we need analyzers that are able to reverse engineer the emulator model and extract the bytecode syntax and semantics. This is a new research area. In a related area, protocol reverse engineering techniques [7], [17], [37] have been proposed to understand network protocol formats by automatically extracting the syntax of the protocol messages. Tupni [10] automatically reverse engineers the formats of all general inputs to a program. The analysis techniques for extracting the input or network message syntax assume that they can be found at predefined locations in the program. In contrast, one of the main challenges in malware emulator analysis is to find where the bytecode program resides.

8. Conclusion

In this paper, we presented a new approach for automatic reverse engineering of malware emulators. We described the algorithms and techniques to extract a bytecode trace and compute the syntax and semantics of the bytecode instructions by dynamically analyzing a decode-dispatch based emulator. We developed Rotalumè, a proof-of-concept system, and evaluated it on synthetic and real programs obfuscated with Code Virtualizer and VMProtect. The results showed that Rotalumè was able to extract bytecode traces and syntax and semantic information. For future work, we plan to address the challenges of reverse engineering other types of emulators. We also plan to develop algorithms to extract higher level instruction semantics that include data-flow information, and to completely convert an extracted bytecode trace back to x86 form. We hope that our work will help spawn research in several other related areas, such as reverse engineering of script interpreters.

Acknowledgments

The authors would like to thank Paul Royal for providing the malware samples that were used in our experiments and the anonymous reviewers for their constructive feedback. This material is based upon work supported in part by the National Science Foundation under grants no. 0716570 and 0831300 and the Department of Homeland Security under contract no. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Homeland Security.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers—Principles, Techniques, & Tools.* Addison Wesley, 2006.
[2] Avira Antivirus. http://www.free-av.com. Last accessed Mar. 6, 2009.
[3] G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. Technical Report 1533, Computer Sciences Department, University of Wisconsin–Madison, 2005.
[4] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.
[5] J. R. Bell. Threaded code. *Commun. ACM*, 16(6), June 1973.
[6] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.
[7] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.
[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1998.
[9] J. J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6), 2002.
[10] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.
[11] E. H. Debaere and J. M. V. Campenhout. *Interpretation and Instruction Path Coprocessing*. MIT Press, Cambridge, MA, 1990.
[12] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.
[13] The IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro/. Last accessed Mar. 6, 2009.
[14] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM)*, 2007.
[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.
[16] P. Klint. Interpretation techniques. *Software Pratice and Experience*, 11(9), Sept. 1981.
[17] Z. Lin, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceeding of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.
[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.
[19] Magnusson and Samuelsson. A compact intermediate format for simics. Technical report, Swedish Institute of Computer Science, 1994.
[20] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[21] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.

[22] Oreans Technologies. http://www.oreans.com/. Last accessed Mar. 6, 2009.

[23] Oreans Technologies: Code Virtualizer. http://www.oreans.com/codevirtualizer.php. Last accessed Mar. 6, 2009.

[24] Oreans Technologies: Themida. http://www.oreans.com/. Last accessed Mar. 6, 2009.

[25] S. Pearce. Viral polymorphism. VX Heavens, 2003.

[26] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of the USENIX Security Symposium*, 2007.

[27] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[28] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Rotalumè: A tool for automatically reverse engineering malware emulators. Technical Report ???, School of Computer Science, Georgia Institute of Technology, 2009.

[29] Silicon Realms. Armadillo/software passport professional. http://siliconrealms.com/. Last accessed Mar. 6, 2009.

[30] R. L. Sites, A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson. Binary translation. 36(2), Feb. 1993.

[31] J. E. Smith and R. Nair. *Virtual Machines: Versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[32] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the USENIX Security Symposium*, 2005.

[33] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1), Jan. 1999.

[34] Virus Total. http://www.virustotal.com/. Last accessed Mar. 6, 2009.

[35] VMPsoft VMProtect. http://www.vmprotect.ru/. Last accessed Mar. 6, 2009.

[36] J. Wilhelm and T. cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[37] G. Wondracek, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceeding of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[38] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

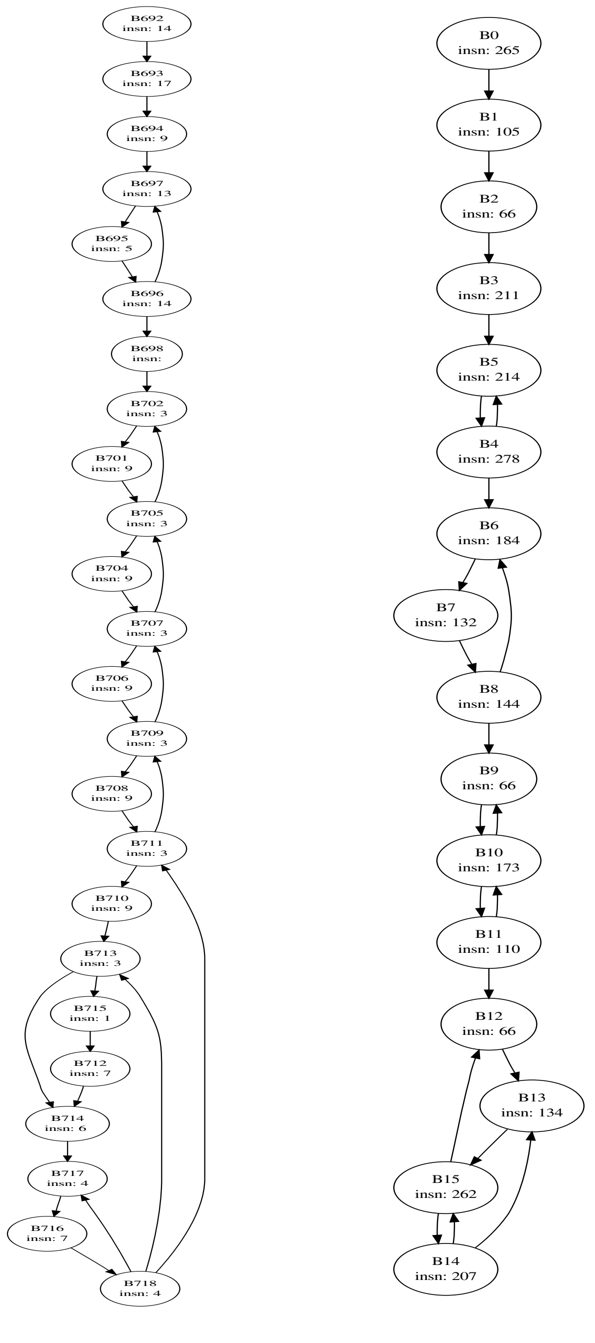
Appendix

1. Control-Flow Graphs for the Unpacked Malware Sample

Figure 5 shows the control-flow graphs of the x86 trace and the bytecode trace extracted by Rotalumè for the experiment on malware sample `Killav.PS`. Each node in the graphs show the number of x86 or bytecode instructions identified in the basic block. The loop backedges show similarity between the extracted CFG and the original CFG.

2. Control-Flow Graphs for CMD.EXE

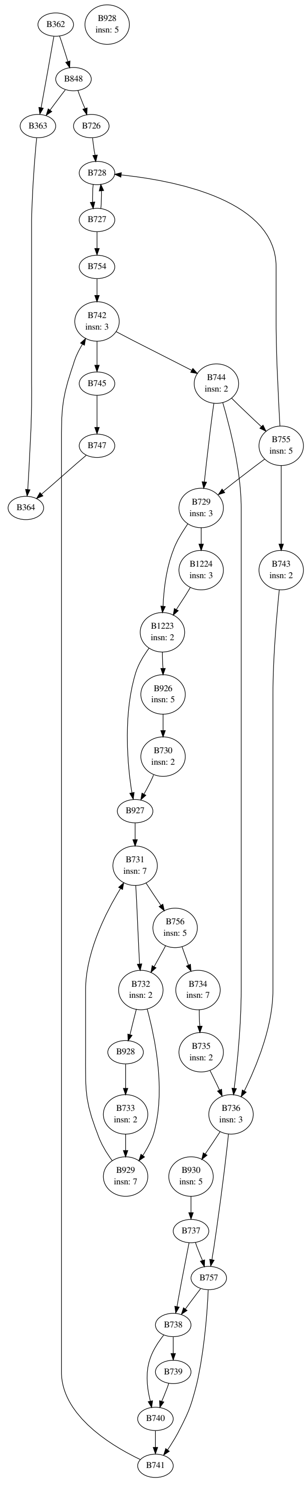
Figure 6 shows the control-flow graphs of a large function of CMD.EXE. We show the original x86 code's CFG in Figure 6(a) and the bytecode version extracted by Rotalumè from the VMProtect obfuscated sample in Figure 6(b). This huge function contained many nested loops and conditional branches. Although the two graphs may look very different at a fist glance, they are actually similar upon close inspection. For example, B928 and B733 in x86 version were collapsed to form B13 in the bytecode version. The loop backedge from B929 to B731 is evident in the bytecode version as the edge from B14 to B10. There are more similarities like this in a few parts of the graphs. The reason for most of the differences may be due to code optimization, block collapsing, or the ways conditional branches are performed and branch target is computed (e.g., B20 for this reason has two outgoing edges).



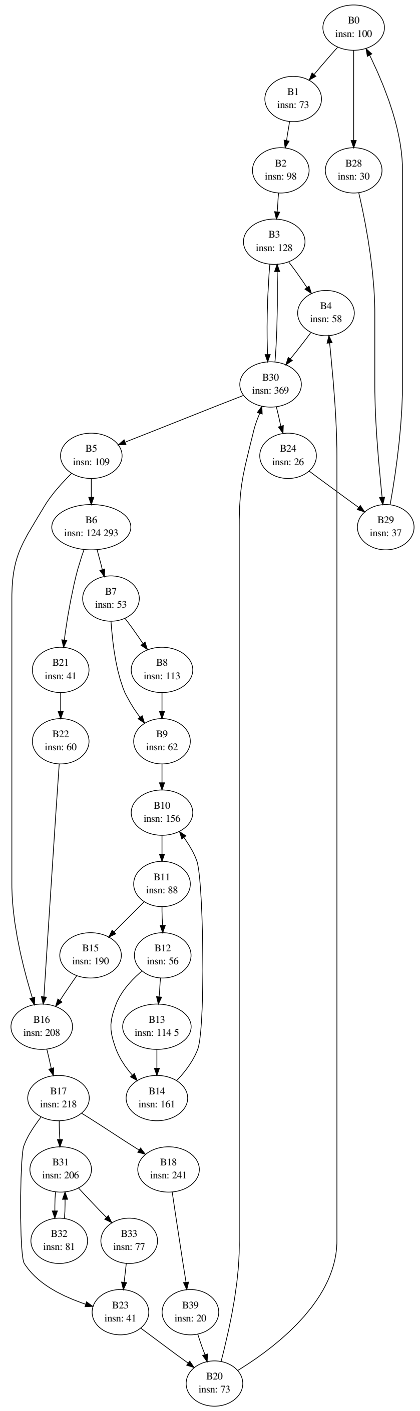
(a) Original x86 CFG

(b) Extracted bytecode CFG (VM-Protect obfuscated)

Figure 5. The x86 and extracted bytecode dynamic CFGs of a function in Killav.PS.



(a) Original x86 CFG



(b) Extracted bytecode CFG (VMProtect obfuscated)

Figure 6. The x86 and extracted bytecode dynamic CFGs of an obfuscated function in CMD.EXE.