Reverse Engineering Self-Modifying Code: Unpacker Extraction

Saumya Debray   Jay Patel
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {debray, jaypatel}@cs.arizona.edu

***Abstract*—An important application of binary-level reverse engineering is in reconstructing the internal logic of computer malware. Most malware code is distributed in encrypted (or "packed") form; at runtime, an unpacker routine transforms this to the original executable form of the code, which is then executed. Most of the existing work on analysis of such programs focuses on detecting unpacking and extracting the unpacked code. However, this does not shed any light on the functionality of different portions of the code so obtained, and in particular does not distinguish between code that performs unpacking and code that does not; identifying such functionality can be helpful for reverse engineering the code. This paper describes a technique for identifying and extracting the unpacker code in a self-modifying program. Our algorithm uses offline analysis of a dynamic instruction trace both to identify the point(s) where unpacking occurs and to identify and extract the corresponding unpacker code.**

***Keywords*-reverse engineering; binary analysis; malware analysis; self-modifying code**

I. Introduction

One of the most important applications of binary-level reverse engineering is in dealing with malware: when developing countermeasures against newly-discovered malware, it is necessary to reverse-engineer the code to understand its internal logic. However, this reverse engineering process is complicated by the fact that malware binaries are typically transmitted in "packed" form, i.e., they are encrypted or compressed; estimates of the prevalence of such packing in malware range from 79% [4] to 92% [3]. When the program is executed at runtime, it invokes an *unpacker* routine that converts the packed code into its original executable form and transfers control to the unpacked code. Packing serves a number of purposes. First, it can hamper reverse engineering of new malware and thereby slow down the development of countermeasures. Second, even if countermeasures have been developed and deployed, packing can make it harder for anti-virus scanners to detect the malware. Finally, packing can help reduce the size of the malware file and thus make it less conspicuous. When reverse engineering malware code, it is necessary to deal with the possibility of runtime code unpacking. This can be challenging, however: malware may be protected using multiple layers of packing, and in some cases the code may modify itself many hundreds of times during execution [12].

Because of the prevalence of packed code in malware, researchers attempting to extract the actual malware code usually resort to dynamic analysis: they execute the malware sample, obtain a trace of the instructions executed, and work back from this to obtain a more-abstract representation of the program. It is useful, in this context, to be able to distinguish unpacker code from unpacked code, for a number of reasons. First, automatic identification of unpacking code can reduce the time needed for reverse engineering by allowing researchers to focus on code with non-unpacker functionality. Second, unpacker identification can help improve the precision of similarity analyses and phylogeny analyses of malware [10], [11], [13]–[15] by allowing them to focus on the actual malware payloads and not be misled by similarities in unpacker code. Third, identifying the unpacker code can help shed light on some aspects of a malware's behavior, e.g., the predicates in any conditional invocation of an unpacker can help us understand the nature and scope of time bombs or logic bombs embedded in the malware code. Finally, the code and logic of unpacker routines—especially specially-crafted custom unpackers—may be interesting in themselves, and may shed light on the specific ways in which the malware code was cloaked.

In addition to the large-scale code modification involved in decrypting the entire payload of a malicious program, malware sometimes also resort to small, tightly-targeted code modifications whose primary aim is code obfuscation in order to hamper reverse engineering (an example is the *Netsky.AA* program discussed in Section IV). For the sake of simplicity, the remainder of this paper uses the term "unpacking" to refer to both kinds of code modification.

Existing tools for malware analysis include several that detect unpacking and extract the code after unpacking occurs [2], [12], [16], [19], but they typically do not offer additional support to specifically identify the unpacker code. Much of the literature on reverse engineering packed code seems to make the explicit or implicit assumption that unpacking is carried out by "tightly bound loops found immediately after the entry point of the program" [18]. This assumption may not be unreasonable for programs where the unpacker is more or less separate from the program code and is essentially layered on top of it. However, it is not difficult to envision scenarios where the unpacker code and the malware payload are tightly integrated. In such a scenario involving incremental unpacking, for example, the unpacker might decrypt a few payload instructions and execute them, then decrypt some more and execute these, and so on. In this case, unpacking is not manifested as a separate part of the program that is conceptually independent from the payload—instead, the two are closely interwoven, and pieces of unpacker code alternate with pieces of payload code through the execution. Without automated support for identifying unpacker code, distinguishing between unpacker code and payload code can be difficult in such situations.

The contribution of this paper is to describe an algorithm for automatically identifying code responsible for making modifications to the code of a program as it executes. Our approach is based on an offline analysis of a dynamic execution trace, and uses the notion of dynamic slicing to identify instructions that cause or contribute to code modification. Since traditional slicing algorithms assume static programs where the code does not change during execution, they do not carry over directly to self-modifying code; to address this, we describe a *phase semantics* where the execution of a self-modifying program can be modeled semantically as a sequence of code snapshots, each of which can be processed using classical algorithms [8].

The remainder of this paper is organized as follows. Section II provides background material on unpacking and phase semantics for self-modifying code, and introduces some definitions and notation. Section III discusses algorithms for identifying unpacking and the structure of the corresponding unpackers. Section IV describes the results of some of our initial experiments. Section V discusses some of the underlying assumptions made by our approach and obfuscations that could violate them. Section VI discusses related work, and Section VII concludes.

II. Background

A. Code Unpacking

Code unpacking refers to the runtime self-modification of code. Figure 1 shows the structure of a very simple unpacker: that for the Hybris-C email worm.[1] Basic block B0 initializes three registers: %edx to the size of the region to be unpacked, %eax to the address from where unpacking is to begin, and %esi to a decryption key. Block B1 is the decryption loop. At each iteration, instruction 4 subtracts register %esi from the contents of the memory word pointed at by %eax, causing the contents of that word to change to its unpacked executable form. After this the decryption key in %esi is updated (instruction 5), the code pointer %eax incremented

[1] For simplicity of exposition we use a quasi-C notation rather than conventional assembly code syntax. The numerical label to the left of each instruction is given to make it easier to refer to individual instructions.

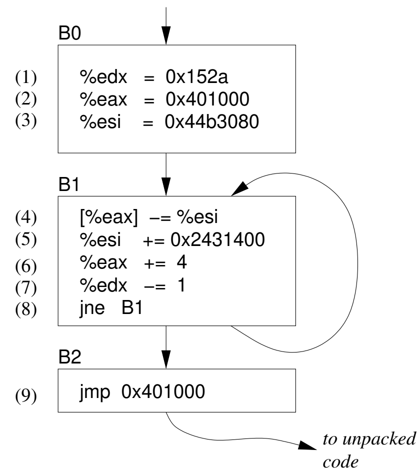


Figure 1. Structure of the unpacker routine for the Hybris-C email worm.

(instruction 6), and the count decremented (instruction 7). If the count is non-zero, control loops back to the beginning of the loop (instruction 8); otherwise, it falls through to an unconditional jump to the decrypted code (instruction 9).

While this code is very simple—unpackers very often have more complex structure—the example illustrates a number of important issues that arise. The first of these is that the identification of the unpacker code may require some analysis. To see this, note that in order to understand the structure and logic of the unpacker code it is not enough to focus only on the instruction(s) that actually generate or modify code in memory—in this example, instruction 4; it is also important to take into account other code that plays a supporting role. In Figure 1, for example, instructions 1, 7, and 8 do not directly affect the values that are written to memory; they don't even have any registers in common with the code-modifying instruction 4. Nevertheless, this set of instructions plays an important role in the unpacking process because it controls the location and size of the memory region where code self-modification takes place. In general, malware code may be littered with otherwise-useless instructions added for obfuscation purposes [6], and program analysis is necessary to tease apart their relationships to determine which instructions affect the unpacking and which are semantically irrelevant. This brings us to the second issue: how can we perform program analysis when the program may be changing during execution? The reason this is an issue is that traditional program analyses make the fundamental assumption that the program being analyzed is static and immutable, which means that they cannot be applied as is to self-modifying code.

This simple example is also misleading in some ways. First, Hybris.C has a simple execution structure consisting of a phase where the malware payload is unpacked, followed by a phase where this payload is executed (the notion of "phase" is discussed in more detail in Section II-B). Many malware have much more complex execution structures, and Kang *et al.* report malware samples with over 500 distinct execution phases [12]. Second, the Hybris.C unpacker is clean and minimal, and does not contain any extraneous obfuscation code, making the unpacker logic fairly obvious. This is not always the case, e.g., in the case of the unpacker for a sample of the Rustock.C spambot [5], the initial unpacker consists of 395 instructions, of which only 55—i.e., less than 15%—actually pertain to unpacking; the remaining 340 consist of small groups of instructions that cancel each other out, effectively behaving as NOPs whose only purpose is to obfuscate. A different kind of anti-detection trick often used by unpackers is to try to "out-wait" anti-virus scanners by stretching out the unpacking process over a large number of instructions; this can result in unpacker instructions accounting for a nontrivial portion of the execution trace. Finally, in the case of Hybris.C the unpacker code and the payload code are distinct and therefore relatively easy to tell apart. However, it is not difficult to imagine self-modifying code where the unpacker code is interspersed with the payload code, and teasing the two apart is nontrivial.

B. Phases

To deal with the issue of analysis of self-modifying code, we have developed a low-level formal semantics for self-modifying programs [8]. A detailed discussion of this work is beyond the scope of this paper; the essential intuition is that the semantics of a program is expressed in terms of its possible execution traces, and the effect of code self-modification during an execution is to partition the corresponding trace into a sequence of *phases*. A phase is a maximal sequence of instructions S in an execution trace that does not execute any location that has been modified by an instruction in S—in other words, one phase ends and another begins when a program attempts to execute code it has just modified. An execution of a self-modifying program can then be modeled as a sequence of distinct phases. Each phase ϕ induces a *code snapshot* $\mathsf{Code}(\phi)$ that consists of the program code as it exists at the beginning of the execution of ϕ together with an instruction in this code where execution begins. A key result is that $\mathsf{Code}(\phi)$ has the property that it is safely analyzable—in the sense that the runtime effects of the execution of ϕ on the instructions comprising ϕ can be computed safely—using traditional analyses [8].[2] The notion of phases forms the foundation for our analysis of self-modifying programs.

A phase ϕ is thus a dynamic notion, namely, a part of an execution trace; the corresponding code snapshot is a static notion, i.e., a code fragment which, when executed, produces the instruction sequence ϕ. This is illustrated in Figure 2, which shows an execution trace consisting of five phases. Each phase is shown together with the corresponding code snapshot. Notice that while each phase (except the first) necessarily has its code modified by the previous phase—this follows from the definition of a phase—there is no requirement that its code be modified *only* by the previous phase. In Figure 2, for example, phase 3 is modified by both phases 1 and 2, while phase 5 is modified by phases 1, 3, and 4. Such arbitrary code-modification relationships between phases can complicate the task of characterizing unpacker code in self-modifying programs.

We can identify phase boundaries in an execution trace by keeping track of the memory locations that are modified by each memory write; for any given phase ϕ, this information can then be used to determine the end of that phase and the beginning of the next one, namely, when execution is about to go to a location that was modified by some instruction in ϕ. In this way, each execution trace can be partitioned into a sequence of phases $\langle \phi_0, \phi_1, \phi_2, \ldots \rangle$, whose behavior can be understood in terms of a sequence of code snapshots

$$\langle \mathsf{Code}(\phi_0), \mathsf{Code}(\phi_1), \mathsf{Code}(\phi_2), \ldots \rangle$$

The case of a conventional program with static code P can then be seen as a degenarate case where every execution has a single phase and the corresponding code snapshot is P.

As a concrete example, the execution of the Hybris-C unpacker shown in Figure 1 consists of two phases: the first phase is the execution of the unpacker code, upto and including the execution of instruction 9, '`jmp 0x401000`,' which transfers control to the unpacked code; the corresponding code snapshot consists of just the code for the unpacker. The second phase consists of the execution of the unpacked code, and the code for this snapshot consists of the unpacked code together with that of the unpacker.[3]

C. Definitions and Notation

An instruction at the machine level occupies one or more adjacent bytes of memory. Let the number of bytes occupied by an instruction I be denoted by $\mathsf{sz}(I)$. Suppose that these bytes start at memory address a and comprise the set of locations with addresses $\{a, a+1, \ldots, a+\mathsf{sz}(I)-1\}$, then we say that *$I$ occurs at* address a.

This paper is concerned with dynamic execution traces, which makes it necessary to be able to reason about properties or behaviors of particular runtime occurrences of an instruction. We refer to a particular runtime instance of an instruction in an execution by its position in the

[2] A phase obviously cannot always be safely analyzed in isolation: it may be necessary to take into account the preceding or succeeding phases. For example, in order to remove obfuscation code via dead code elimination, it is necessary to take into account uses of registers and memory locations in later phases. The point is that phases allow us to deal with the effects of code modification in a precise and well-defined way.

[3] Since the unpacking process does not overwrite the unpacker code in this case, the latter remains part of the program.

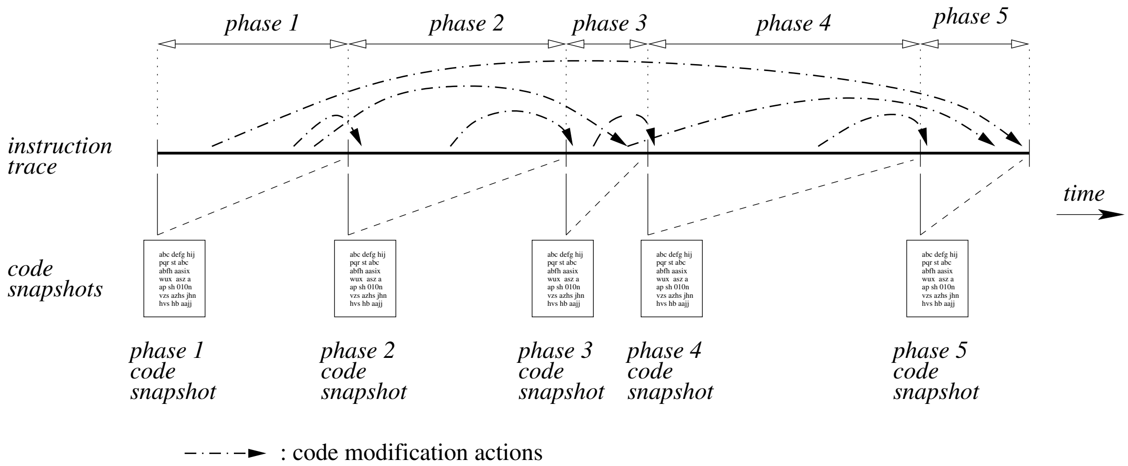


Figure 2. Phases, code modification, and code snapshots

corresponding execution trace. Let **S** be a sequence, then the i^{th} element of **S** is denoted by $\mathbf{S}[i]$. An execution trace for a program is a sequence of triples $(addr, instr, regs)$, where *addr* is a memory address, *instr* the instruction occurring at that address, and *regs* is a set of register-value pairs giving the values of the machine registers just before the instruction is executed. The reason for including register values in the trace is that they allow us to determine which memory locations are accessed—and, in particular, modified—by indirect memory references.

Consider a trace

$$\mathbf{T} = (a_1, I_1, R_1), (a_2, I_2, R_2), \ldots, (a_i, I_i, R_i), \ldots$$

The i^{th} element of this trace is $\mathbf{T}[i] = (a_i, I_i, R_i)$. The address, instruction, and register components of the i^{th} element of **T** are denoted by, respectively, $\mathsf{Addr}(\mathbf{T}[i])$, $\mathsf{Instr}(\mathbf{T}[i])$, and $\mathsf{Regs}(\mathbf{T}[i])$. The set of locations occupied by the instruction at position i in trace **T** is denoted by $\mathsf{Locns}(\mathbf{T}[i])$:

$$\mathsf{Locns}(\mathbf{T}[i]) = \{a_i, a_i + 1, \ldots, a_i + \mathsf{sz}(I_i) - 1\}.$$

We use the notation $\mathsf{Write}(\mathbf{T}[i])$ to denote the set of memory locations that are written by the instruction at position i of a trace **T**.

Since the instructions for a program reside in memory, the only way to modify code at runtime is to modify the contents of memory by writing to it. In order to identify unpackers, therefore, a first step is to identify which instructions have been unpacked, i.e., are being executed after being modified. To this end, we have the following definitions:

Definition 2.1: An instruction at position k in a trace **T** is *unpacked* if one or more of the locations it occupies have been modified earlier in the execution, i.e., if there exists $j < k$ such that $\mathsf{Locns}(\mathbf{T}[k]) \cap \mathsf{Write}(\mathbf{T}[j]) \neq \emptyset$. □

In general, a memory location may be modified many times during the execution of a program. When an unpacked instruction I is executed, therefore, we would like to focus on those instructions that actually wrote one or more of the bytes comprising I. To this end, we use the notion of the set of modifiers of I. Intuitively, an instruction I_j is a modifier of an unpacked instruction I_k if I_j makes some change to the memory locations occupied by I_k that "survives" until I_k is executed. More formally, we have:

Definition 2.2: An instruction at position j in a trace **T** is a *modifier* of an instruction at position k in **T** iff $\exists w \in \mathsf{Locns}(\mathbf{T}[k])$ such that the following hold:

(i) $w \in \mathsf{Write}(\mathbf{T}[j])$, i.e., $\mathsf{Instr}(\mathbf{T}[j])$ modifies some location occupied by $\mathsf{Instr}(\mathbf{T}[k])$; and

(ii) there is no position i such that $j < i < k$ and $w \in \mathsf{Write}(\mathbf{T}[i])$, i.e., location w is not overwritten by any intervening instruction between positions j and k.

The set of modifiers of an instruction at position i in a trace **T** is denoted by $\mathsf{Modifiers}(\mathbf{T}[i])$. □

In general, an unpacked instruction may have more than one modifier. For example, an instruction that occupies three bytes of memory may have three different modifiers, each of which modifies one of its constitutent bytes.

III. Unpacking and Unpackers

Our approach to identifying unpackers consists of three main steps. Starting with an execution trace for a program, we first identify the different phases and the unpacked instructions in each phase. The second step is to identify the modifier instructions for each of the unpacked instructions. Finally, we use slicing techniques to identify the unpacker code. The remainder of this section describes these steps in more detail.

A. Identifying Phases and Unpacked Instructions

Once an execution trace $\mathbf{T}$ has been collected, we identify its phases along with the set of unpacked instructions, in a single forward pass over its instructions. The algorithm maintains two sets of memory locations: *GlobalWriteSet* keeps track of the set of locations that have been modified since the beginning of the program's execution, while *CurrWriteSet* gives the set of locations modified so far in the current phase. For each position in the trace, the set of locations occupied by the corresponding instruction is compared with these two sets: if it overlaps with *GlobalWriteSet*, the instruction has been unpacked; if it overlaps with *CurrWriteSet*, that instruction position marks the beginning of a new phase. Pseudocode for the algorithm is as follows:

> *PhaseNo* := 1; *CurrWriteSet* := $\emptyset$; *GlobalWriteSet* := $\emptyset$;
> mark the first position in $\mathbf{T}$ as the start of phase 1;
> **for** each position i in the trace, going forward, **do**
>   **if** $\mathsf{Locns}(\mathbf{T}[i]) \cap \mathit{GlobalWriteSet} \neq \emptyset$ **then**
>   mark position i as unpacked;
>   **fi**
>   **if** $\mathsf{Locns}(\mathbf{T}[i]) \cap \mathit{CurrWriteSet} \neq \emptyset$ **then**
>   increment *PhaseNo*;
>   mark position i as the start of phase *PhaseNo*;
> $\mathit{CurrWriteSet} := \emptyset$;
>   **fi**
> $\mathit{GlobalWriteSet} := \mathit{GlobalWriteSet} \cup \mathsf{Write}(\mathbf{T}[i])$;
> $\mathit{CurrWriteSet} := \mathit{CurrWriteSet} \cup \mathsf{Write}(\mathbf{T}[i])$;
> **od**

B. Identifying Modifier Instructions

The set of modifier instructions can be identified via a single backward pass over the execution trace, as follows:

> $\mathit{ModInsLocs} := \emptyset$;
> **for** each position i in $\mathbf{T}$, going backwards, **do**
>   **if** $\mathsf{Write}(\mathbf{T}[i]) \cap \mathit{ModInsLocs} \neq \emptyset$ **then**
>   mark position i as a modifier;
> $\mathit{ModInsLocs} := \mathit{ModInsLocs} \setminus \mathsf{Write}(\mathbf{T}[i])$;
>   **fi**
>   **if** position i is unpacked **then**   /* *A* */
> $\mathit{ModInsLocs} := \mathit{ModInsLocs} \cup \mathsf{Locns}(\mathbf{T}[i])$;
>   **fi**
> **od**

At each iteration of this loop, the set *ModInsLocs* gives the set of locations that are occupied by unpacked instructions later in the execution trace. Thus, an instruction is a modifier if it writes to some location that is occupied by an instruction at a later position in the trace. Note that since an instruction that is not unpacked will never have its locations overlap with the locations written to by any instruction, we can make this loop a little faster by eliminating the check labeled '*A*' above, i.e., making the update to *ModInsLocs* unconditional, without affecting the correctness of the algorithm. The effect of this change is to remove a test from the loop; however, it results in a larger number of values in the set *ModInsLocs*, which can lead to more expensive set operations elsewhere in the loop.

C. Identifying Unpackers

The concepts introduced in the previous section allow us to identify which instructions in a phase have been unpacked and which instructions actually performed the corresponding memory modifications. This information, while useful, may not be enough, however. For example, consider the Hybris.C unpacker shown in Figure 1: for each unpacked instruction in the second phase of this program, the modifier is instruction 5 in basic block B1: '`[%eax] -= %esi`.' This instruction, by itself, does not tell us much about the unpacker.

To see what else we need, consider an unpacked instruction at position k in a trace $\mathbf{T}$; for simplicity of discussion, suppose that it has a single modifier instruction, which is at position j, i.e.,

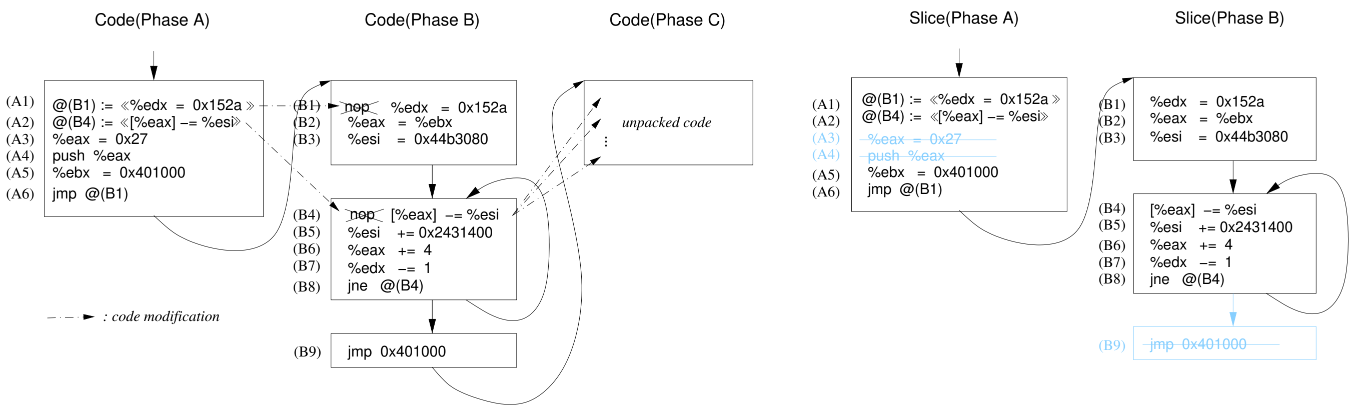
$$\mathsf{Modifiers}(\mathbf{T}[i]) = \{\mathsf{Instr}(\mathbf{T}[j])\}.$$

To understand the behavior of the unpacker, we need to identify the instructions and program logic involved with computing the value v that is written to memory by $\mathsf{Instr}(\mathbf{T}[j])$. But this is exactly the dynamic slice of the program for the slicing criterion (inp, v, j), where inp is the set of inputs to the program for trace $\mathbf{T}$. If the unpacked instruction has more than one modifier instruction, then this applies to each modifier: we combine the dynamic slice for each modifier. We can generalize the idea to an entire phase ϕ: compute the appropriate dynamic slice for each modifier of each modified instruction in ϕ and combine the results. This is illustrated by the following example.

Example 3.1: Figure 3(a) shows a variation on the Hybris.C unpacker of Figure 1, where an additional code modification phase has been added before the main unpacker loop. Execution now consists of three phases: Phase A, which modifies two instructions in the code for Phase B; Phase B, the main unpacker loop; and Phase C, the execution of the unpacked code. The program behaves as follows:

- Initially, when Phase A begins execution, the locations occupied by instructions B1 and B4 (in Phase B) are occupied by `nop` instructions.[4]

[4]For the sake of simplicity of discussion, we assume in this example that all instructions are the same size.



(a) Code snapshots for the original program (b) The unpacker code

Figure 3. An example unpacked program and its unpacker. Instructions in the original program that are not in the dynamic slice comprising the unpacker are shown in light color and crossed out.

- Instruction A1 writes the binary encoding of the instruction %edx = 0x152a to the memory location(s) occupied by instruction B1. In Figure 3(a), the binary-level encoding of an instruction I is denoted by $\ll I \gg$, while the location(s) occupied by an instruction J is written '$@J$'. The nop instruction at this location is thus overwritten, which is indicated by showing the nop as crossed out.
- Instruction A2 writes the binary encoding of the instruction [%eax] -= %esi to the locations occupied by instruction (B4) of Phase B.

Once Phase A ends, the code for Phase B is essentially the same as that in Figure 1—the one minor difference is that instruction B2, which is in Phase B, uses register %ebx, which is defined by instruction A5 in Phase A. This change was made to introduce a cross-phase data dependency and make the slicing problem more interesting.

The unpacked instructions in Phase C all have instruction B4 as their modifier. If the instruction in Phase C at position j in the execution trace $\mathbf{T}$ is unpacked by an execution of instruction B4 at position j' in the trace, then the slicing criterion is the value that is written to Locns($\mathbf{T}[j]$) by $\mathbf{T}[j']$, i.e., the value of the expression '[%eax] – %esi' at position j' in $\mathbf{T}$. This slice consists of instructions A5 and A6 from Phase A and instructions B1, …, B8 from Phase B. Since instructions B1 and B4 are themselves unpacked, with modifiers A1 and A2 respectively, we include the dynamic slices for these as well. When these slices are all merged together, the result is the code shown in Figure 3(b). This matches what we would intuitively consider to be the unpacker code for this program. □

To make this work, we have to compute program slices for self-modifying programs. We do this using the notion of phases described earlier. Recall that the code for a self-modifying program can be expressed as a sequence of static programs—i.e., code snapshots—one per phase. A dynamic slice for a self-modifying program can, correspondingly, be expressed as a corresponding sequence of dynamic slices, one for each phase. The main issue that has to be taken into account is that dependencies due to code modification have to be taken into account. These are very similar to ordinary data dependencies, with the difference that while a data dependence $A \longrightarrow B$ exists between two instructions (statements) A and B if A defines a location whose value is used by by B, a code-modification dependence $A \longrightarrow B$ exists if A defines a location that is occupied by B, i.e., if A is a modifier of B. The key intuition here is that in order to execute an instruction it is necessary to read—and, therefore, to "use"—the memory locations it occupies.

Existing dynamic slicing algorithms can be extended to incorporate the notion of code-modification dependences in a number of different ways; here we discuss one simple approach. The idea is to extend each instruction to take a number of "pseudo-arguments" that correspond to the memory locations it occupies. Thus, consider a k-byte instruction $I(y_1,\dots,y_n,x_1,\dots,x_m)$, with source operands $x_1,\dots,x_m$ and destination operands $y_1,\dots,y_n$, whose semantics is given as

$$\langle y_1,\dots,y_n\rangle := f_I(x_1,\dots,x_m)$$

for some appropriate function f_I that depends on the instruction I. We rewrite this instruction to incorporate k additional arguments, one corresponding to the address of each byte that it occupies:

$$I(y_1,\dots,y_n,x_1,\dots,x_m,a_1,\dots,a_k),$$

The semantics of the rewritten function remains the same as before; however, the set of locations it "uses" is defined to be $\{x_1, \ldots, x_m, \mathsf{Mem}[a_1], \ldots, \mathsf{Mem}[a_k]\}$, where $\mathsf{Mem}[a]$ denotes the memory location with address a. The additional arguments thus serve to capture the semantic property that the behavior of this instruction depends on the most recent instructions that modify any of the locations with addresses $a_1, \ldots, a_k$. Thus, suppose that $\mathsf{add}(y, x_1, x_2)$ is a four-byte instruction that computes $y = x_1 + x_2$, then a particular add instruction '$\mathsf{add}(r_2, r_0, r_1)$' occupying memory locations $\mathsf{1000} \ldots \mathsf{1003}$ would be rewritten as

$$\mathsf{add}(r_2, r_0, r_1, \mathsf{1000}, \mathsf{1001}, \mathsf{1002}, \mathsf{1003}).$$

With instructions rewritten in this manner to make explicit the memory locations they occupy, code modification dependences are translated to data dependences and can be handled in the same way without any additional changes to the slicing algorithm.

IV. Experimental Results

We have implemented our ideas in a prototype tool for reverse engineering malware code. We applied our tool to four different malware samples: *Breatle.J*, *Hybris.C*, *Mydoom.Q*, and *Netsky.AA*. Of these, *Hybris.C* and *Netsky.AA* use custom packers, *Mydoom.Q* uses the UPX packer [17], and *Breatle.J* uses a combination of two commercial packers: Aspack [1] and UPX. We collected traces for these programs by executing them under the control of the OllyDbg debugger [21] and using the tracing facility of OllyDbg to record the instructions executed (however, our approach is not tied to OllyDbg, and will work with any tracing tool that provides a minimal set of information about executing instructions (see Section II-C), and we are currently in the process of switching to a tracing tool called Ether [9]). These programs were run on Windows XP on a VMware virtual machine that, for security reasons, was configured to have no network connectivity; because of this, the behaviors we observed were most likely incomplete—the malware would typically attempt to connect to the Internet and quit after repeated unsuccessful attempts. However, in order to get to this point they had to unpack their code, so we were able to observe their unpacking behavior.

The results obtained using our unpacker extraction tool on the resulting execution traces are discussed below. We manually verified correctness for the smaller unpackers obtained (*Hybris.C*, *Mydoom.Q*, the first two phases of *Netsky.AA*); for the larger unpackers, where the size and complexity of the code made manual verification impractical, we checked that the modifying instructions were being found correctly and that dependencies between instructions were being computed correctly, which gives us confidence that the slices computed are also correct.

Hybris.C: This program, whose unpacker was discussed in Section II (see Figure 1), is the simplest program in our test suite. The execution of this program consists of two phases: an unpacking phase followed by the execution of the unpacked code.

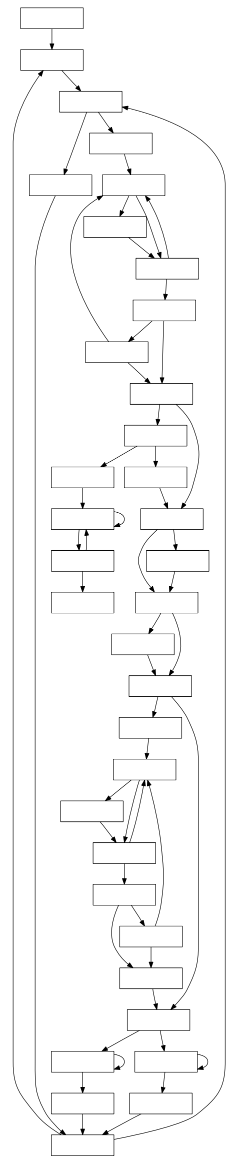


Figure 4. Unpacker structure for *Mydoom.Q*

The unpacker extracted by our algorithm for this program is identical to that shown in Figure 1, with the single difference that the unpacker slice computed does not contain instruction 9, which branches to the unpacked code. This exclusion is appropriate, since instruction 9 does not belong in the slice for the modifier instruction.

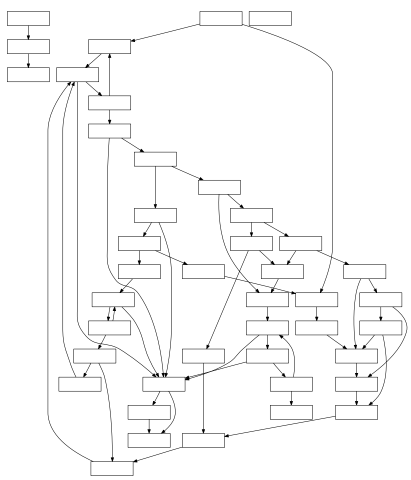
Mydoom.Q: This program uses a commercial packer, UPX [17], to pack its code. UPX was originally developed to compress executable files to reduce file size—and, therefore, the size of the packed code. In keeping with this goal, the unpacker code is such that the size of the unpacker

```
[0x403e5f]  mov eax, 0x403e6e
[0x403e64]  add byte [eax], 0x28
[0x403e67]  inc eax
[0x403e68]  add dword [eax], 0x1234567
```

(a) Phase 1

```
[0x403e84]  mov [eax], ecx
[0x5cbc32]  mov eax, 0xf05cabd3
[0x5cbc37]  lea ecx, [eax+0x10001082]
[0x5cbc3d]  mov [ecx+0x1], eax
[0x5cbc44]  mov edx, [edx+0xc]
[0x5cbc47]  mov byte [edx], 0xe9
[0x5cbc4a]  add edx, 0x5
[0x5cbc4d]  sub ecx, edx
[0x5cbc4f]  mov [edx-0x4], ecx
```

(b) Phase 2



(c) Phase 3

Figure 5. Unpacker structure for *Netsky.AA*

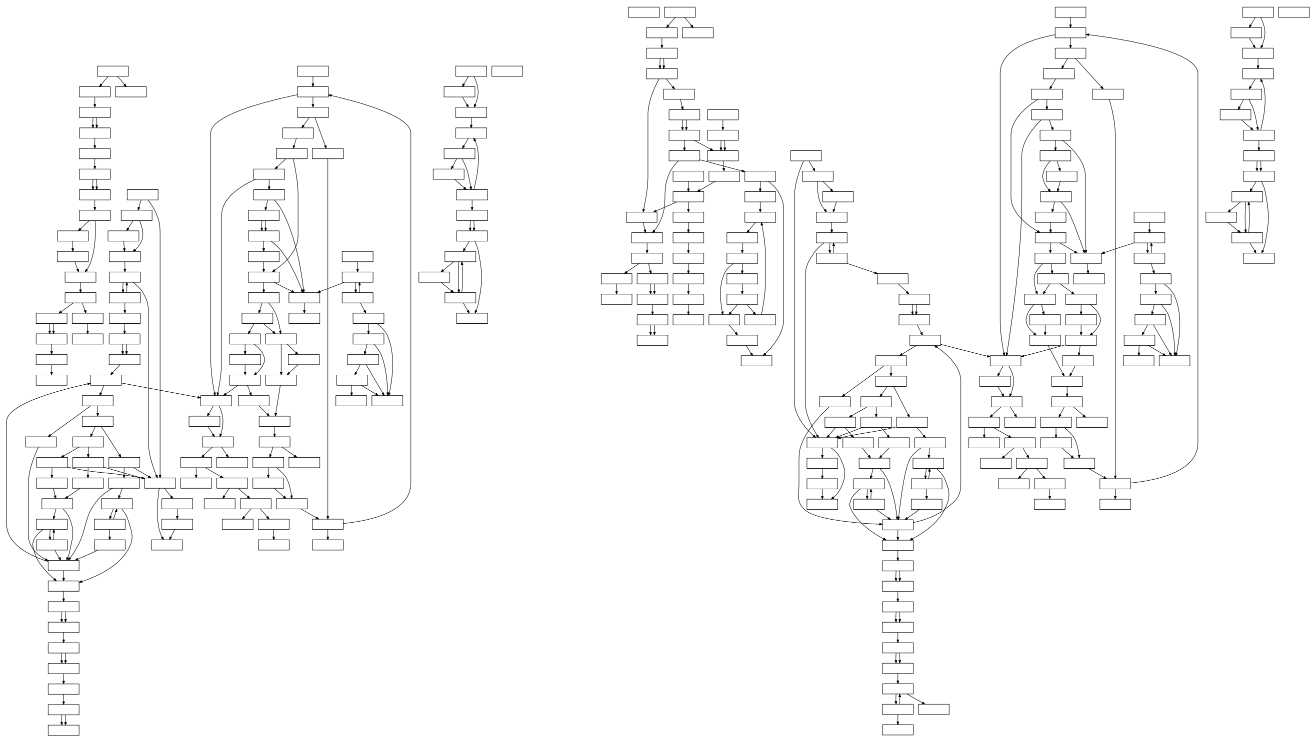
is kept as small as possible, even at the cost of more complex unpacking logic. Program execution consists of two phases: an unpacking phase followed by the execution of the unpacked code. The structure of the control flow graph for the unpacker is shown in Figure 4 (due to space constraints we show only the basic blocks and control flow edges, but not the instructions within the blocks).

Netsky.AA: This program's execution consists of four phases, of which the first three are given to unpacking; the code structure for these are shown in Figure 5. What is interesting about this is the wide disparity in size and complexity between the first two unpacker phases and the last. The first two unpacking phases involve only small amounts of code modification—in each case the unpacker consists of a small piece of straight-line code that modifies only a few bytes of memory—and seem to be intended purely for obfuscation:

- The two `add` instructions in Phase 1 (Figure 5(a)) modify the locations immediately following the second of these add instructions. Execution then falls through to this modified code (thus, there is no explicit jump to the unpacked code), which installs an exception handler, with code address `0x5cbc32`, and prepares to raise an exception.
- The second phase begins with a deliberate null-pointer-dereference via an indirect store through the register `eax`, which was zeroed out in the previous phase. This causes an exception and transers control to the exception handler installed in the previous phase (note the large difference in addresses between the first instruction in Figure 5(b), which raises the exception, and the second instruction, which handles it[5]). The exception handler modifies the code at two widely different places in the program: it overwrites code at address `0x5cbc56` (corresponding to the Phase 3 unpacker), and changes the instruction at address `0x403e84`, which originally raised the exception that transferred control to the handler, with an instruction '`jmp 0x5cbc55`'. Thus, when control returns from the exception handler it finds this newly-written unconditional jump that causes control to be transferred to the Phase 3 unpacker.
- The third phase is considerably larger than the two preceding phases and has more complex control flow logic. It is this phase that carries out the real unpacking of the malware payload. The control flow graph for this phase is shown in Figure 5(c): due to space constraints we only indicate the control-flow relationships between basic blocks but do not show the instructions within each block. (This graph seems to have disconnected components because it does not show call/return edges for indirect function calls.)

Breatle.J: Of the programs we tested, *Breatle.J* has the greatest complexity of unpacking: it has the largest number of execution phases: a total of six, of which five involve unpacking, and the unpackers for the different phases also have fairly complex control flow logic. Figure 6 shows the structures of the control flow graphs for the first two of the five unpacking phases of this program. The unpacker structure for these two phases, as well as those not shown, suggests that parts of the unpacker logic are similar across the various unpacking phases. The section names in the file suggest that it is protected using a combination of two commercial packers, Aspack [1] and UPX [17], but the actual unpacker code found seems to be significantly more complex than what one would expect for these two packers; this suggests that there may be additional layers of packing involved. We are currently investigating this in more detail.

[5]Arguably, these two instructions should be in different basic blocks that are connected by some sort of exception edge. Our control flow analysis is currently not smart enough to infer that the `mov` instruction in reality effects a control transfer through an exception.



(a) Phase 1 (a) Phase 2

Figure 6. Unpacker structure for *Breatle.J* (the first two of five unpacking phases)

V. DISCUSSION

A fundamental assumption in the work described above—and one that is also made by other tools that detect unpacking [2], [12], [16]—is that we can detect unpacking by comparing the virtual addresses of instructions that are executed with the virtual addresses of memory locations that are written to. This assumption may not hold if a technique known as *dual mapping* [20] is used. The basic idea of this obfuscation is to map a physical page to two virtual pages, one of which is mapped as a writable page and the other is mapped as an executable page. This has an effect similar to pointer aliasing: since both virtual pages refer to the same physical page, writing to the writable virtual page changes what is seen on the executable virtual page. However, since the two virtual pages have different virtual addresses, it appears as though the executed locations have disjoint virtual addresses from the modified ones.

It turns out that not all automatic unpackers are susceptible to this obfuscation technique [20]. The investigation of how the techniques used by such unpackers might be incorporated into our system is a topic of future work.

VI. RELATED WORK

A number of authors have described tools to detect runtime code modification and extract the dynamically unpacked code [2], [12], [16], [19]. However, these tools typically simply present all of the unpacked code in an execution phase as a collection of instructions without any indication of any of their possible roles, e.g., as unpacker vs. malicious payload.

Quist and Liebrock describe a visualization tool that can assist users with identifying unpacker code [18]. The tool's function is to provide a high-level visualization of various characteristics of different portions of the program rather than to specifically extract and identify unpacker code. It reconstructs a program's control flow graph from an execution trace and uses heuristic rules, based on a comparison of the program's code in memory with that in the original executable file, to identify unpacker loops. While the tool provides a great deal of information about the functionality of different parts of the program, the heuristic nature of unpacker identification means that errors in unpacker identification cannot be ruled out.

Coogan *et al.* have investigated the use of static analysis techniques for static identification and extraction of unpacker code [7]. While the goals of both works are conceptually similar, the details of the approaches used are very different. In particular, the work of Coogan *et al.* assumes accurate static disassembly, which is not always easy to guarantee, and requires a sophisticated pointer analysis to obtain reasonable precision; the dynamic approach described here, by contrast, allows the use of simpler analysis algorithms and generally obtains more precise results, albeit for the single execution path covered by the execution that is considered.

VII. Conclusions

An important application of binary-level reverse engineering is in dealing with malware. Since most malware are transmitted in encrypted form and are decrypted, i.e., "unpacked," at runtime, the reverse engineering process has to contend with runtime unpacking as well. In this context, an interesting question is that of automatically identifying and characterizing the code that carries out such unpacking. This paper describes an approach that can be used to identify code that carries out unpacking. Our approach is based on a low-level semantics that can be used to reason about self-modifying code. We use dynamic analysis to extract an execution trace for the program and then apply dynamic slicing (appropriately adapted to handle self-modifying code) to identify unpacker code. Our ideas have been implemented in a prototype tool; experiments indicate that it is effective in identifying unpacker code even for programs with fairly complex unpacking logic.

References

[1] ASProtect software. http://www.aspack.com/asprotect.aspx.

[2] Lutz Böhne. *Pandora's Bochs: Automated Unpacking of Malware*. PhD thesis, Aachen University, January 2008.

[3] T. Brosch and M. Morgenstern. Runtime packers: The hidden problem? In *Black Hat Briefings*, August 2006.

[4] P. Bustamante. Mal(ware)formation statistics, May 2007. PandaResearch Blog. http://research.pandasecurity.com/malwareformation-statistics/.

[5] K. Chiang and L. Lloyd. A case study of the Rustock rootkit and spam bot. In *Proc. HotBots '07: First Workshop on Hot Topics in Understanding Botnets*. Usenix, April 2007.

[6] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[7] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Proc. 16th. IEEE Working Conference on Reverse Engineering*, pages 167–176, October 2009.

[8] S. K. Debray, K. P. Coogan, and G. M. Townsend. On the semantics of self-unpacking malware code. Technical report, Dept. of Computer Science, University of Arizona, Tucson, July 2008. `http://www.cs.arizona.edu/~debray/Publications/self-modifying-pgm-semantics.pdf`.

[9] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 51–62, 2008.

[10] M. Gheorghescu. An automated virus classification system. In *Virus Bulletin Conference*, pages 294–300, October 2005.

[11] L. A. Goldberg, P. W. Goldberg, and C.A.Phillips, G.B.Sorkin. Constructing computer virus phylogenies. *J. Algorithms*, 26(1):188–208, January 1998.

[12] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, November 2007.

[13] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *J. in Computer Virology*, 1(1):13–23, 2005.

[14] A. Lakhotia, Md. E. Karim, A. Walenstein, and L. Parida. Malware phylogeny using maximal π patterns. In *EICAR 2005 Conference: Best Papers Proceedings*, pages 167–174, 2005.

[15] Z. Liang, T. Wei, Y. Chen, X. Han, and J. Zhuge. Component similarity based methods for automatic analysis of malicious executables. In *Virus Bulletin Conference*, September 2007.

[16] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC 2007, Miami Beach, Florida, USA*. IEEE Computer Society, December 2007.

[17] M. F. X. J. Oberhumer, L. Molnár, and J. F. Reiser. UPX: the Ultimate Packer for eXecutables. http://upx.sourceforge.net/.

[18] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In *Proc. VizSec 2009: Workshop on Visualization for Cyber Security*, October 2009.

[19] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300. IEEE Computer Society, 2006.

[20] skape. Using dual-mappings to evade automated unpackers. *Uninformed Journal*, 10(1), October 2008. `http://www.uninformed.org/?v=10&a=1&t=sumry`.

[21] O. Yuschuk. Ollydbg. `http://www.ollydbg.de/`.