Automatic Static Unpacking of Malware Binaries

Kevin Coogan Saumya Debray Tasneem Kaochar Gregg Townsend
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
{kpcoogan, debray, tkaochar, gmt}@cs.arizona.edu

***Abstract*—Current malware is often transmitted in packed or encrypted form to prevent examination by anti-virus software. To analyze new malware, researchers typically resort to dynamic code analysis techniques to unpack the code for examination. Unfortunately, these dynamic techniques are susceptible to a variety of anti-monitoring defenses, as well as "time bombs" or "logic bombs," and can be slow and tedious to identify and disable. This paper discusses an alternative approach that relies on static analysis techniques to automate this process. Alias analysis can be used to identify the existence of unpacking, static slicing can identify the unpacking code, and control flow analysis can be used to identify and neutralize dynamic defenses. The identified unpacking code can be instrumented and transformed, then executed to perform the unpacking. We present a working prototype that can handle a variety of malware binaries, packed with both custom and commercial packers, and containing several examples of dynamic defenses.**

***Keywords*-malware; analysis; static unpacking; dynamic defenses**

I. Introduction

Recent years have seen explosive growth in attacks on computer systems via a variety of malware such as viruses, worms, bots, etc. Computer system security has accordingly become an increasingly important concern, and a variety of products, such as virus scanners, have been developed to detect and eliminate malware before they can do any damage.

Modern malware are typically transmitted in "scrambled" form—either encrypted or packed—in an attempt to evade detection. The scrambled code is then restored to the original unscrambled form during execution. Here, *encryption* refers to the use of some kind of invertible operation, together with an encryption key, to conceal the malware. *Packing* refers to the use of compression techniques to reduce the size of the malware payload, which has the side effect of disguising the actual instruction sequence. The distinction between these approaches is not typically important for our purposes. Hence, both will be referred to generically as "packing." The code used to transform the binary to its scrambled form is referred to as a *packer*, and the code that undoes the scrambling is called the *unpacker*.

The use of packers poses a problem for security researchers, because in order to understand how a new virus or worm (or a new variant of an old one) works, it is necessary to reverse engineer its code. If the code is packed, then it must be unpacked as part of this reverse engineering process. In some cases, it may be possible to use syntactic clues to identify the packer used on a program [1]. If a known unpacker exists, such as for many commercial packing tools like UPX [2], [3], it can be used to unpack the file. However, malware writers can close this obvious hole by deliberately altering the "signatures" of the packer in the packed binary, or by using their own custom encryption and decryption routines. When confronted with a malware binary packed with an unknown packer, therefore, researchers rely almost exclusively on dynamic analysis techniques to identify its code, e.g., by running the malware binary under the control of a debugger or emulator [4], [5], [6].

Unfortunately, these dynamic techniques have a number of shortcomings. Execution of malware code may allow the malware to "escape." In some cases, elaborate infrastructure is required to prevent this, e.g., dynamic analysis of bluetooth-enabled devices carried out in a giant Faraday cage to prevent accidental infection through wireless transmission [7]. Dynamic techniques are also tedious and time-consuming [8]. New malware can spread very quickly [8], and the more time it takes to analyze these new threats, the longer the threat can spread unabated. Finally, and most importantly, dynamic analyses are vulnerable to conditional execution of the unpacker routine. Malware may deploy anti-debugging and anti-monitoring defenses [9], [10], [11], [12], and skip the unpacking routine if it finds its execution is being monitored, or it could be designed to execute only under certain environmental conditions, such as a particular date.

These problems with dynamic analysis motivate our search for an alternative approach to identifying the code generated when a packed malware binary is unpacked. Our overall goal is to use static program analyses to construct detailed behavioral models for malware code, which can then be used by security researchers to understand various aspects of the behavior of a malware binary: how the code for a program may change as it executes; the control and data flow logic of the various "layers" of the binary that are created by successive unpacking operations; and the static and dynamic

Instr	*Address*	*x86 assembly code*
	0x401000:	{*... encrypted malware body ...*}
...		
I_0	0x4064a8:	movl %edx ← $0x152a
I_1	0x4064ad:	movl %eax ← $0x401000
I_2	0x4064b2:	movl %esi ← $0x44b3080
I_3	0x4064b7:	subl (%eax) ← %esi
I_4	0x4064b9:	addl %esi ← $0x2431400
I_5	0x4064bf:	addl %eax ← $4
I_6	0x4064c2:	decl %edx
I_7	0x4064c3:	jne .-0xc
I_8	0x4064c5:	jmp 0x401000

Figure 1. Unpacker code for Hybris-C worm

defenses deployed by the binary. This paper takes a first step in this direction by describing a general and automatic approach to statically unpacking malware binaries. Its main contributions are as follows:

1) It shows how well-understood program analyses can be used to identify whether a program may be self-modifying (which may indicate unpacking).
2) For programs that are found to be possibly self-modifying, it shows how the code modification mechanism, i.e., the code that carries out the unpacking, can be identified and used to unpack the binary without any prior knowledge about the packing algorithm used.
3) It shows how standard control and data flow analyses can be applied to this code modification mechanism to find (and possibly neutralize) dynamic defenses, time/logic bombs, etc. that activate the unpacking conditionally.

Our approach is intended to complement — and not necessarily replace — dynamic analysis techniques currently used by researchers for analyzing malware.

II. Background

Figure 1 shows an example of a simple unpacker, in this case for the Hybris-C email worm. Instructions I_0 and I_1 load registers with the size (5418 words) and start address (0x401000) of the region to be unpacked. Instruction I_2 loads the encryption key. I_3 through I_7 iterate over each word of the region, performing the decryption by means of the subtract instruction, and rotating the key value with addition. When the value of the `%edx` register (the number of words left to decrypt) is greater than zero, execution jumps back to I_3. At zero, it falls through and branches to the unpacked code. Other unpackers may differ from this code in various aspects, but they all share the property that they modify memory to create new code that was not present (in that form) in the original binary, then execute the code so created. This observation forms the key to our approach to static unpacking.

Unfortunately, at the level of a program binary, code bytes may be indistinguishable from data bytes. In some cases, the bytes at a memory address may be used for both [13]. Hence, it may not be possible to tell which memory writes target code, and which modify data. Some researchers have addressed this question using heuristics, e.g., by considering all writes to the text section of a binary as a code modification [14]. This approach does not always work, since code can be generated in memory regions other than the text section, e.g., the data section or the heap, and the text section can contain embedded data whose modification does not, intuitively, constitute code modification. Furthermore, packers may rename sections arbitrarily, e.g. UPX-packed binaries typically have sections named `UPX0`, `UPX1`, etc. It may not even be possible to tell, by simple inspection, which sections contain executable code. ASPACK [2] changes the flags in the section header table of the packed binary so that no section appears to be executable. For these reasons, we consider a more behavioral approach, where a location is considered to be code if it is possible for execution to reach that location. As we will see, however, even this is difficult to identify statically, and we will have to resort to conservative approximations.

In general, a program may modify its code in different ways and enter the modified code from different program points. Figure 2 shows a simple example. Execution begins at block B_0 and immediately checks to see whether it is being monitored. If it is not, control falls through to the unpacker code in block B_2, which unpacks the malware payload and branches to it. If monitoring is detected, however, control branches to block B_1, which overwrites the code in blocks B_0 and B_2 with no-ops—thereby presumably covering its tracks. Execution falls through the no-op sequence into unrelated code following B_2. In this example, there are essentially two unpackers: the one in B_2 which unpacks the code and branches to it, and the one in B_1 which hides the malicious intent of the program and falls through to the unrelated code. In general, there may be an arbitrary number. Thus, our approach focuses on identifying points in the program where control can enter unpacked code, and treats the unpackers for each case separately.

III. Identifying the Unpacker

In order to carry out unpacking statically, we first have to identify the code in the packed binary that carries out unpacking. In order to do this, we have to be able to distinguish between the part of the malware's execution when unpacking is carried out from the part where it executes the unpacked code. This section discusses the essential semantic ideas underlying the notion of "transition points," i.e., points in the code where execution transitions from unpacker code to the unpacked code newly created by the unpacker. This notion of transition points underlies our approach to identifying and extracting the unpacker code, which is then used to carry out unpacking.

The discussion in Section II implies that it suffices, for our purposes, to focus on an individual transition from ordinary (i.e., unmodified) code to modified code. To this end, given a program P, consider a trace of a single execution of P. This consists of a sequence of states $\mathbf{S} = S_0, S_1, \ldots$, where each state S_i consists of (i) the contents of memory, denoted by $Mem(S_i)$;[1] and (ii) the value of the program counter, denoted by $pc(S_i)$, specifying the location of the instruction to be executed next. At any state S_i, we can determine which memory locations (if any) have changed relative to the previous state S_{i-1} by comparing $Mem(S_i)$ with $Mem(S_{i-1})$. This notion generalizes in a straightforward way to the set of memory locations modified over a sequence of states $\langle S_i, \ldots, S_j \rangle$, which we will denote by

$$ModLocs(\langle S_i, \ldots, S_j \rangle)$$

We can divide the execution trace $\mathbf{S}$ into two phases: an initial unpacking phase $\mathbf{S}_{unpack}$, followed by the subsequent execution of unpacked code $\mathbf{S}_{exec}$:

$$\mathbf{S} = \underbrace{S_0, \ldots, S_k}_{\mathbf{S}_{unpack}}, \ \underbrace{S_{k+1}, \ldots,}_{\mathbf{S}_{exec}} .$$

The boundary between these two phases is marked by the execution of a memory location that was modified earlier in the execution. Thus, $\mathbf{S}_{exec}$ begins (and $\mathbf{S}_{unpack}$ ends) at the first state S_{k+1} for which $pc(S_{k+1}) \in ModLocs(\langle S_0, \ldots, S_k \rangle)$; if no such S_{k+1} exists, no unpacking has taken place on this execution. If we assume complete knowledge about the trace $\mathbf{S}$, we can give an idealized definition of the set of unpacked locations, UL_{ideal}, as those locations in $ModLocs(\mathbf{S}_{unpack})$ that are subsequently executed:

$$UL_{ideal}(\mathbf{S}) = ModLocs(\mathbf{S}_{unpack}) \cap \{pc(S_i) \mid S_i \in \mathbf{S}_{exec}\}.$$

In practice, of course, we do not have *a priori* knowledge of the set of locations that will be executed after unpacking (in fact, until unpacking has been carried out, we do not even know which locations *could* be executed after unpacking). We therefore use the set of memory locations modified during the unpacking phase up to the point where control enters an unpacked location, i.e., the set of $ModLocs(\mathbf{S}_{unpack})$, as a conservative approximation to the idealized set of unpacked locations $UL_{ideal}(\mathbf{S})$. We then define the dynamic unpacker $\mathcal{U}_D$ for the trace $\mathbf{S}$—i.e., the code that actually carries out the memory modifications in the unpacking phase of this trace—to be the fragment of the program P that was executed during $\mathbf{S}_{unpack}$ and which could have affected the value of some location in $ModLocs(\mathbf{S}_{unpack})$. This is nothing but the dynamic slice of P for the set of locations $ModLocs(\mathbf{S}_{unpack})$ and the execution trace $\mathbf{S}_{unpack}$.

There are two key pieces of information used to define the dynamic unpacker here: the state where control is about to flow into unpacked code, and the set of memory locations that get modified by the time control reaches this state. For static analysis purposes, we consider the natural static analogues for these. Reasoning analogously to the distinction between $\mathbf{S}_{unpack}$ and $\mathbf{S}_{exec}$ above, we find a pair of locations (ℓ, ℓ') such that control can go from ℓ to ℓ', and ℓ' may have been modified earlier in the execution. We refer to such pairs as transition points:

[1]For simplicity of discussion, we consider registers to be a special part of memory.

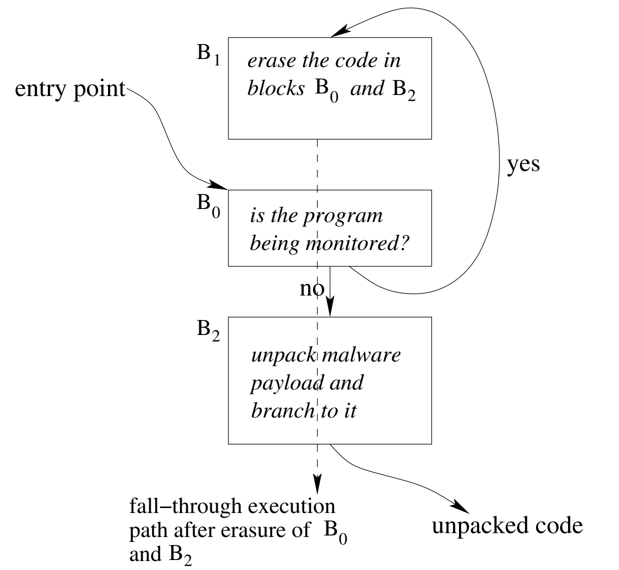


Figure 2. A schematic of a program that can modify its code in different ways

Definition 3.1: A transition point *in a program P is a pair of locations (ℓ, ℓ') satisfying the following:*

1) *ℓ is not modified during the execution of P;*
2) *there is an execution path from the entry point of P to the point ℓ along which ℓ' may be modified; and*
3) *the next instruction to be executed after ℓ may be at ℓ'.*

$\blacksquare$

Intuitively, a transition point (ℓ, ℓ') gives a static characterization of the point where control goes from the unpacker to the unpacked code: ℓ corresponds to the program counter in S_k, the last state in the unpacking phase $\mathbf{S}_{unpack}$ in the trace shown above, while ℓ' corresponds to the program counter in S_{k+1}, the first state of the unpacked code execution phase $\mathbf{S}_{exec}$.

Given a transition point t for a program P, let $Mods(t)$ denote (an upper approximation to) the set of memory locations that may be modified along execution paths from the entry point of the program to t. We define $\mathcal{U}_S(t)$, the static unpacker for t, to be the static backward slice of P from the program point t with respect to the set of locations $Mods(t)$, i.e., the set of instructions whose execution could possibly affect any of the locations in $Mods(t)$.

The remainder of this paper focuses on how, given an executable program P, we identify the set of static unpackers for it and use them to carry out unpacking statically.

IV. Handling Dynamic Defenses

As mentioned earlier, one of the drawbacks with dynamic analysis of malware binaries is that it allows the malware to deploy dynamic defenses. Examples of such defenses include anti-debugging code, which attempt to detect whether the program's execution is being monitored; time bombs, which cause the malware to be activated only at certain times or dates; and logic bombs, which activate the malware upon the detection of some environmental trigger.

We can classify dynamic defenses into three categories, whose conceptual structures are shown in Figure 3. Here, the variable *src* refers to the packed code. The first kind, shown in Figure 3(a), is *simple*: here, the dynamic defense predicate is executed after the malicious code has been unpacked. The second kind, shown in Figure 3(b), is *control-based*: here the dynamic defense predicate is executed first, and the unpacker is invoked conditionally based on the outcome of this test. The *W32.Divinorum* virus attempts to use such a technique [15] (though a bug in the code renders the defense ineffective). Finally, Figure 3(c) shows *data-based dynamic defense*, whose effect is to pass different values to the unpacker based on the outcome of the test. As a result, the outcome of unpacking is different based on whether or not the dynamic defense predicate is true.

Many of the dynamic defenses currently encountered in malware use the simple defense shown in Figure 3(a). Existing emulation-based techniques are sufficient to identify the malware in this case, since the malicious code is materialized in unpacked form in memory regardless of whether or not it is executed. We therefore do not consider such defenses further. However, *control-based* and *data-based* may cause a dynamic analyzer to miss, or incorrectly unpack, the true malware, thus leading to tedious and time-consuming manual intervention.

Using static analysis, we can use the control-flow structure of the malware code to detect dynamic defenses. To this end, we recall the notions of dominators and post-dominators from static control-flow analysis [16]. Given two basic blocks B and B' in the control flow graph of a program P, B *dominates* B' if every execution path from the entry point of P to B' goes through B. B *post-dominates* B' if every execution path from B' to the exit node of P passes through B. We can use these notions to identify dynamic defenses, as follows:

- **Control-based defenses:**
  Unpacking is control-dependent on a conditional branch C if the unpacker code is reachable from C but does not post-dominate C.
- **Data-based defenses:**
  Unpacking is data-dependent if the instructions that define the unpacking parameters do not dominate the unpacker code.

In the first case, since the unpacker code does not post-dominate the conditional branch C, control may or may not reach the unpacker at runtime depending on the outcome of C. In the second case, since the instructions defining the unpacking parameters do not dominate the unpacker, different execution paths can assign different values for these parameters. Note that in both cases these are necessary but not sufficient conditions (the usual undecidability results for static analysis make it difficult to give nontrivial sufficient conditions).

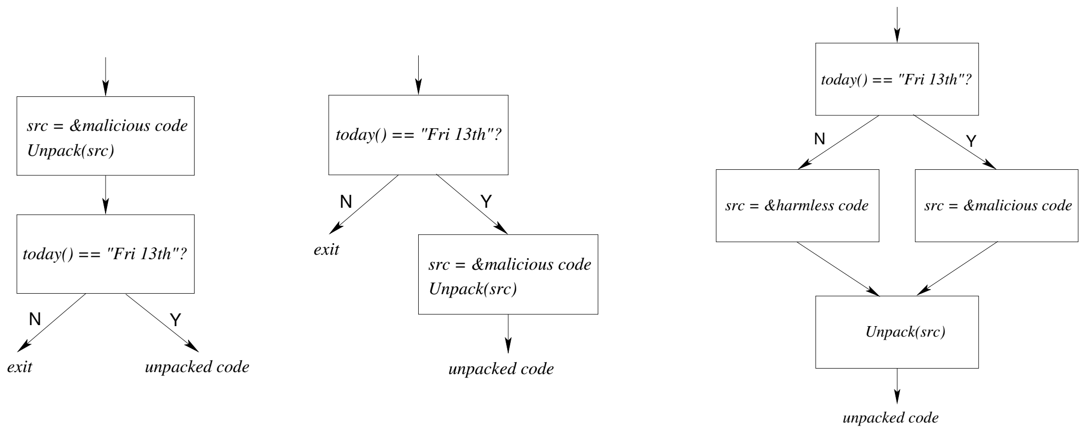
It turns out that a data-based dynamic defense can be transformed to a control-based one using a code transformation known as *tail duplication*. Fom the perspective of static unpacking, therefore, data-based dynamic defenses can be handled by transforming them to control-based defenses and then handling these as discussed in Section V-B2. The remainder of this paper therefore focuses on dealing with control-based dynamic defenses.

While dynamic defenses can be detected as discussed above, in general they cannot always be completely eliminated while preserving the unpacking behavior. To see this, consider the situation where an external input is read in and used as a password and also as a decryption key: in this case, eliminating the dynamic defense would be equivalent to automatically guessing the password. However, automatic elimination of dynamic defenses may be possible if the value that is tested in the dynamic defense predicate is unrelated to the decryption key(s) used for unpacking (here, the two are considered to be "related" if there is some value v such that, for some functions f and g, the value $f(v)$ is used in the dynamic defense predicate and $g(v)$ is used as an unpacking key). This is usually true of malware code, where the dynamic defense is related to some aspect of the external environment, e.g., execution under the control of a debugger or in a virtual machine, while the unpacking key is typically stored within the program executable itself.

V. Our Approach

The overall organization of our static unpacker is as follows:

1) *Disassembly and control flow analysis.* Read in the input binary and use information about the program entry point (found in the file header) to obtain an initial disassembly of the binary. We perform control flow analysis using standard techniques to identify basic blocks and construct the control flow graph of the disassembled code [16].
2) *Alias analysis.* Perform binary-level alias analysis to determine the possible target addresses of indirect memory operations in the disassembled code. Our current implementation uses the value-set analysis described by Balakrishnan [17], [18].



(a) simple (b) control-based (c) data-based

Figure 3. Different kinds of dynamic defenses

3) *Potential Transition point identification.* Use the results of alias analysis to identify potential transition points i.e., points where control may be transferred to unpacked code (see Definition 3.1). We refer to these as "potential" because imprecision in the alias analysis may identify some locations as possible transition points even though in reality they are not.
4) *Static unpacker extraction.* For each potential transition point t identified above, we use the results of alias analysis to determine the set of memory locations that may be modified along execution paths to t, and use backward static slicing on this to identify the static unpacker $\mathcal{U}_S(t)$.
5) *Static unpacker transformation.* Various analyses and transformations are applied to the unpacker $\mathcal{U}_S(t)$, extracted in the previous step, to enable it to be executed as part of a static unpacking tool. These include the detection and elimination of dynamic defenses that effect control-dependent unpacking, as well as address translation and code change monitoring.
6) Finally, the transformed code is invoked to effect unpacking.

A. Potential Transition Point Identification

As outlined above, we begin by disassembling the binary, then carrying out alias analysis for all indirect memory references (the targets of direct references are readily apparent, and do not need additional analysis). Given the aliasing information, for each instruction I we compute two sets: $write(I)$, the set of memory locations that may be written to by I; and $next(I)$, the set of locations that control may go to after the execution of I. These sets are computed as follows, with $alias(x)$ denoting the possible aliases of a memory reference x:

$$write(I) = \begin{cases} \{a\} & \text{if } I \text{ is a direct write to} \\ & \quad \text{a location } a; \\ alias(r) & \text{if } I \text{ is an indirect write} \\ & \quad \text{through } r; \\ \emptyset & \text{otherwise.} \end{cases}$$

$$next(I) = \begin{cases} \{a\} & \text{if } I \text{ is a direct control} \\ & \quad \text{transfer to a location } a; \\ alias(r) & \text{if } I \text{ is an indirect control} \\ & \quad \text{transfer through } r; \\ \{addr(I')\} & \text{otherwise, where } I' \text{ follows} \\ & \quad I \text{ in the instruction} \\ & \quad \text{sequence.} \end{cases}$$

We next identify potential transition points, which indicate points where control may go from the unpacker into the unpacked code (i.e., modified locations). More formally, the idea is to collect all instructions I such that there is some instruction J that can modify some location in $next(I)$ and where there is a control flow path from J to I. Imprecision in the alias analysis will lead to multiple potential transition points. We extract and execute a slice for each one to identify true transition points.

B. Static Unpacker Extraction

Once potential transition points have been identified as described above, we process each transition point t in turn and extract the corresponding unpacker $\mathcal{U}_S(t)$. To this end, let *ep* denote the entry point of the program (i.e., instruction sequence) P under consideration, and define the set of memory locations $Mods(t)$ that may be modified along some execution path leading to t as follows:

$$Mods(t) = \bigcup\{write(I) \mid I \in P \text{ is reachable from } ep \text{ and } t \text{ is reachable from } I\}.$$

The unpacker $\mathcal{U}_S(t)$ associated with t is then computed as the backward static slice of the program from t with respect to the set of locations $Mods(t)$. Note that because of the unstructured nature of machine code, slicing algorithms devised for structured programs will not work; we use an algorithm, due to Harman and Danicic, intended for unstructured programs [19]. Since the computation of this slice considers all of the memory locations that can be modified in any execution from the entry point of the program up to the point t, most of the code in the initial disassembly is usually included; however, obfuscation code that is dead or which has no effect on any memory location will be excluded.

Once identified, we want use the unpacker to unpack the code. However, in its raw form it is not suitable for execution. For example, virtual addresses in the code will not point to their intended locations since the unpacker will be loaded into allocated memory on the heap. Additionally, any dynamic defenses will still be included and may disrupt the unpacking process. Our next step, therefore, if to transform the code to a form suitable for execution using the following transformations.

1) Instruction Simplification: The Intel x86 architecture (targeted by a great deal of malware because of its ubiquity) has a number of instructions with complex semantics and/or *ad hoc* restrictions. The simplification step rewrites such instructions, which are difficult to handle during the address translation step (Section V-B3), to an equivalent sequence of simpler instructions.

As an example, the *repz/repnz* prefixes on certain string instructions cause repeated execution of the instruction. The effect of the prefix is to decrement the `%ecx` register, then—depending on the prefix, the value of `%ecx`, and the result of the last comparison operation in the string instruction—either repeat the execution of the string instruction, or else exit the repetition. The problem here is that the side effect of the *repz/repnz* prefix on the `%ecx` register interferes with register save/restore operations in the address translation step. We address this by replacing the *repz/repnz* prefix with explicit arithmetic on the `%ecx` register together with a conditional jump that reexecutes the string instruction when necessary.

2) Dynamic Defense Elimination: As mentioned in Section IV, data-based dynamic defenses can be transformed to control-based ones in a straightforward way, so it suffices to deal with control-based dynamic defenses. We do this as follows. After the slice $\mathcal{U}_S(t)$ has been constructed, we check each conditional branch that is in the slice to see whether it might be a dynamic defense test. For each such branch J, we check to see whether there is some instruction I in the slice that does not post-dominate J. If this is the case, we transform the code as follows:

1) if the slice is reachable along the *true* edge of J but not along the *false* edge, we "unconditionalize" J, i.e., replace J by a direct jump to the target of J;
2) if the slice is reachable along the *false* edge of J (i.e., along the fall-through) but not along the *true* edge, we remove J;
3) if the slice is reachable along both the *true* and *false* edges of J, J is left unchanged.

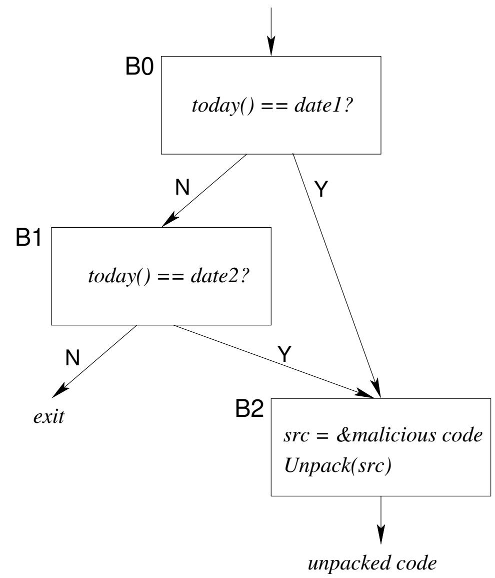


Figure 4. A dynamic defense with a compound predicate

This process is repeated until there is no further change to the slice.

The first two cases above are fairly obvious. To see the need for the third case, consider Figure 4, which modifies the code in Figure 3(b) so that the unpacker now runs on two different days. In this case, suppose that block B0 is processed first by our algorithm. The unpacker code is reachable from both its *true* and *false* branches of this test, so the test is left unchanged (case 3 above). Block B1 is processed next, and case 1 is found to apply, so the conditional branch in B1 is replaced by an unconditional jump to B2. In the next iteration, we consider block B0 again, and again find that the unpacker code in B2 is reachable along both the *true* and *false* edges out of B0, so there is no change to the slice, and the algorithm terminates. Notice that in this example, even though the test in block B0 remains as part of the slice, the dynamic defense has effectively been disabled: control goes to the unpacker code regardless of the outcome of this test.

As noted earlier in Section IV, it may not be possible to eliminate a dynamic defense if the value tested in the defense predicate is related to a value used for unpacking. We can identify this situation by examining data dependency relationships in the slice code.

3) Sandboxing: Once dynamic defenses have been eliminated, we further transform the code to ensure that memory accesses are handled correctly. There are two components to this: *address translation*, which redirects accesses to global memory regions (code and static data) to the appropriate locations; and *stack shadowing*, which deals with stack accesses from the malware code.

I. Address Translation: The need for address translation arises out of the fact that the runtime unpacking of a malware binary takes place within an executing malware file, while in our static unpacking tool it occurs within a tool where each section comprising a malware binary is represented as a dynamically-allocated data object. Each such object—which we refer to as an *s-object* (for "section object")—contains meta-data about the section it represents, such as its name, size, virtual address, etc., as well as the actual byte sequence of the contents of the section (where appropriate). These section meta-data are obtained from the section header table of the binary. Because of these different representations, memory references in the unpacking code $\mathcal{U}_S(t)$—which refer to virtual addresses in the malware binary, e.g., 0x401000 for the Hybris code of Figure 1—have to be translated to addresses that refer to s-objects in the static unpacker's memory.

We achieve this translation by traversing the instruction sequence resulting from the instruction simplification step, discussed in the previous section. For each instruction that accesses memory (except those that access through the stack pointer), the following instrumentation and transformation is performed. First, a new instruction is added that calculates the virtual address used by the original instruction and stores the result in a register r_0. The function *VirtualAddr2UnpackerAddr*() is called with the value in r_0 as a parameter, and returns the value of the corresponding address in static unpacker memory. The return value is stored back into some register r_1 (it could be r_0, but doesn't have to be), and the original instruction is transformed so that it accesses memory indirectly through r_1. Finally, instrumentation code is added before and after these instructions that save and restore the values of all registers as needed. Thus, the correct memory location is used by the original instruction, and the instruction acts on the current machine state. The implementation of *VirtualAddr2UnpackerAddr*() is straight forward. We note that virtual address space forms a contiguous block of memory addresses starting with the base address as given in the file header. Our static unpacker memory likewise forms a contiguous block of memory of the same size with a known start address, thus there is a one-to-one correspondence between virtual addresses and unpacker addresses. Translation, then, amounts to calculating the offset of the virtual address from the file header base address, and adding that offset to the start of the unpacker memory space. This approach also allows us to identify attempts to access memory outside of the program address space.

To deal with calls to library routines, we use a set of "wrapper" routines we have created for commonly-used library functions. At program startup, we construct our own version of the executable's Import Address Table (IAT) and build a mapping from these IAT functions and addresses to our known wrapper functions and addresses. This mapping is maintained as a global data structure. Function calls are handled as follows. If it is a direct call and the target is within the unpacker slice, it is rewritten to transfer control to the appropriate instruction within the slice. If it is an indirect call, we instrument the code with a call to a handler routine. At runtime, the call handler first tries to determine if the call target is within the slice. If it is, it translates the address and returns it. The returned value is substituted for the original value, and the call instruction is executed. If the target is not in the slice, the call handler assumes the call is a library call. In this case, if the tool is running under *cygwin* (a Unix-like environment within Microsoft Windows), and the target is one of the set of wrapper routines we have created, the wrapper library routine is called; otherwise we skip the call instruction.

II. Stack Shadowing: There are two main reasons we must explicitly handle stack accesses. First, correct execution of the program may depend on values on, or below, the stack. For example, the *Peed-44* trojan uses an offset from the stack pointer to reach below the stack to the Thread Execution Block (TEB) to access a value that is used to carry out the unpacking (the TEB is actually stored in higher addresses, but we say "below" because the stack grows towards lower address values.) Second, it is necessary to protect the static unpacker's runtime stack should the malware try to write garbage to it or use the stack in an unpredictable way. For example, the *Rustock.C* unpacker uses a number of *push* and *pop* instructions to obfuscate its code; at runtime, this has the effect of writing garbage onto the stack.

We handle these issues by allocating a region of memory, called the *shadow stack region*, that holds the contents of two contiguous memory areas from the malware code's exectuion environment: its runtime stack and its TEB. The stack area of the shadow stack region grows from high to low addresses, similar to the actual runtime stack; the address of its top is recorded in a global variable, the *shadow stack pointer*. We locate the our TEB just below the stack, as is done in Windows. Memory within the TEB area of this region, as well as the shadow stack pointer, are initialized with values one would expect when a Windows process begins execution. Additionally, code is added to slice $\mathcal{U}_S$ so that the runtime stack and shadow stack are switched immediately before each instruction and switched back immediately after the instruction.

C. Transition Point Detection

As mentioned above, not all potential transition points are actual transition points. We can test for actual transition points as follows. Execution of the static unpacker acts on and records changes to our own copy of the program memory M. Before execution, we create a read-only copy of memory M'. Further, we instrument the slice code so that before each instruction is executed, we can compare the contents of the current memory to the contents of the original memory at the address of the instruction. If these contents have changed, then the instruction has been modified, and we stop execution of the unpacker, otherwise we continue. This approach assumes that given a transition point (ℓ, ℓ') there is an instruction at both of the addresses ℓ and ℓ'. This may not be true, e.g. if the packed bytes of ℓ' did not disassemble to a legal instruction. In this case, we can add nop instructions as need to potential targets.

D. Putting it all together

Once the slice code has been generated, transformed and instrumented as described above, we add wrapper code around it to save the appropriate components of program state on entry (e.g., stack and frame pointers, flags) and restore this state prior to exit. The resulting instruction sequence is then run through an assembler that traverses the list of instructions and emits machine code into a buffer allocated for this purpose. A driver routine in our static unpacker then executes a function call to the beginning of this buffer to effect unpacking; control returns from the buffer to the code that invoked it once unpacking is complete.

If a slice completes execution without finding a transition point, it returns control to the driver routine. The driver then restores the contents of malware memory M from the read-only copy M', and executes the next slice.

After the malware binary has been unpacked in this fashion, we still have to extract the resulting unpacked code. Since we know the transition point for the unpacker, i.e., the address of the unpacked code, we can do this by disassembling the code starting at this address. The resulting disassembled unpacked code can then be processed using standard control and data flow analyses.

VI. Experimental Results

A. Static Unpacking

To evaluate the efficacy of our ideas, we implemented a prototype static unpacker and tested it on five files – four viruses, and one non-malicious program packed with a common packer. (i) *Hybris-C* uses a single arithmetic decryption operation where the decryption key is changed via a rotation at each iteration of the unpacking routine, (ii) *Trojan.Peed-44* looks into the TEB to get the value to the top of the SEH chain. Since it has not loaded any exception handlers, this value will be -1 if it is running natively. It uses this value to calculate the start address of the unpacking, then iterates through addresses and performs a series of bit shifting and arithmetic on each. (iii) *Rustock* rootkit uses two sequential decryptor loops that operate on the same memory. Additionally, more than three quarters of its unpacker instructions are obfuscation code that perform various memory and stack operations which have no effect. (iv) *Mydoom* is packed with commercial packer UPX, and has a fairly elaborate algorithm consisting of nested loops that may write to memory under different conditions. (v) TextPad is not a malicious file, but we packed it with *tElock*, a program often used to hide malware. *tElock* uses the aam instruction, which ordinarily adjusts the result of multiplication between two unpacked BCD values. Here, it has the effect of an implicit decryption key. *tElock* also uses several anti-disassembly tricks such as jumping into the middle of instructions, near calls to load a value on the stack, and *int $0x20* instructions that appear to be in the control flow, but never execute. For all the files above, our approach does not require knowledge of the unpacking algorithm. It only needs to identify the correct slice.

Figure 5 summarizes the results of our experiments. These numbers were obtained as follows:

1) We dump the program's memory image P_{orig} at the point where it begins execution. We execute the code in a debugger, setting a break point at the first unpacked instruction, and dump the program's unpacked memory image P_{unpD}. The size of this unpacked image $\mid P_{unpD} \mid$, is reported in column 1 of Figure 5.
2) Column 2 gives the number of bytes unpacked N_{unp}, calculated as the number bytes that differ between P_{orig} and P_{unpD}.
3) We run our static unpacker on a file, and if it finds that a potential transition point is a true transition point, it dumps the memory image. We denote this P_{unpS}. The value of Δ given in column 3 is the number of bytes where P_{unpD} and P_{unpS} differ.
4) Column 4 gives the accuracy of static unpacking, expressed as the percentage of bytes where P_{unpS} and P_{unpD} agree.

For all programs in Figure 5 , we have verified that the differences between P_{unpS} and P_{unpD} are the result of differences in program metadata, specifically entries in the Import Address Table, and not the bytes that are actually being unpacked. (The IAT is a section of a file used to deal with the invocation of dynamically-linked library routines.) *Rustock* loads no functions from external .dll files, *Hybris-C* loads one function, *Peed-44* loads six functions, and *tElock* loads two. The differences for these files are the result of the four byte addresses of these functions not getting loaded into the IAT by our static unpacker. The case of *Mydoom* is similar. UPX-packed binaries have two separate parts to the unpacker. First, original program bytes are uncompressed and loaded into memory. Second, the list of imported functions is unpacked and the IAT is rebuilt manually. We have confirmed through manual analysis that the 2,708 bytes that differ for *Mydoom* all result from this second step. The results of Figure 5 represent a single phase of unpacking. We do not handle cases where multiple phases, e.g. malware packed multiple times, are required to fully unpack the original binary. This case is left for future work.

Program	Memory image size $\mid P_{unpD} \mid$ (bytes)	Bytes unpacked N_{unp}	Memory difference Δ (bytes)	% correct $1 - \Delta/\|P_{unpD}\|$
Hybris-C	28,672	21,576	4	99.99
Mydoom.q	73,728	57,367	2,708	96.33
Peed-44	151,552	1,872	48	99.97
Rustock.C	69,632	22,145	0	100.0
tElock	1,974,272	6,140	8	99.99

Figure 5. Experimental results: static unpacking

B. Handling Dynamic Defenses

To evaluate the detection of dynamic defenses, we constructed several variants of the *Hybris* program incorporating various control-based dynamic defenses. We varied the structure of the code so that, for different variants, the dynamic defense code appeared above, or below, or intermixed with, the actual unpacker code. In each case, the static unpacker was able to successfully identify and eliminate the dynamic defense code.

VII. Related Work

The usual approach to dealing with packed malware binaries is via dynamic analysis, i.e., by monitoring the behavior of the binary as it executes [4], [5], [6]. The only other work that we are aware of on static unpacking of malware binaries is that of signature-based schemes such as PEid [1], which look for syntactic signatures in the packed binary in an attempt to identify the packer used to create it. If a packer is identified, the corresponding unpacker can then be invoked to unpack the binary. The problem with such an approach is that it only works for known packers and requires that the signature be found. It therefore does not work if the malware binary uses a custom packer (or one not previously encountered) or if the packed binary is modified to expunge the signatures that identify its packer. Our approach, by contrast, does not rely on knowledge about the packer to obtain the unpacker; rather, it uses static analyses to identify and extract the unpacker code.

There is a great deal of research literature on alias analysis (see, for example, the discussion by Hind and Pioli [20] and the survey by Rayside [21]) and program slicing (see the survey papers by Tip [22] and Xu *et al.* [23]). Bergeron *et al.* discuss the use of static slicing techniques for identifying malicious behavior in unpacked binaries [24]. We are not aware of the application of any of this work towards automatic unpacking of malware.

VIII. Conclusions

This paper proposes an alternative approach to dynamic analysis techniques used by researchers. It uses well-understood static analysis techniques to identify the unpacking code that comes with a given malware binary, then uses this code to construct a customized unpacker for that binary. This customized unpacker can then be executed or emulated to obtain the unpacked malware code. Our approach does not presuppose any knowledge about the software or algorithm used to create the packed binary. Preliminary results from a prototype implementation suggest that our approach can effectively unpack a variety of packed malware, including some constructed using custom packers and some obtained using commercial binary packing tools.

Acknowledgment

This work was supported in part by the Air Force Office of Scientific Research via grant no. FA9550-07-1-0019, and by the National Science Foundation via grant nos. CNS-0410918 and CNS-0615347, as well as by a GAANN fellowship from the Department of Education award no. P200A070545.

References

[1] "PEid," http://www.secretashell.com/codomain/peid/.

[2] "Aspack software," http://www.aspack.com/asprotect.html.

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

[4] 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)*, Nov. 2007.

[5] P. Szor, *The Art of Computer Virus Research and Defense*. Symantek Press, Feb. 2005.

[6] R. Warrior, "Guide to improving polymorphic engines," `http://vx.netlux.org/lib/vrw02.html`.

[7] M. Hypponen, "Mobile malware," Aug. 2007, invited talk, USENIX Security '07.

[8] V. A. Mahadik, "Reverse engineering of the honeynets SOTM32 malware binary," the Honeynet Project, `http://www.honeynet.org/scans/scan32/sols/4-Vinay_A_Mahadik/Analysis.htm`.

[9] A. Danielescu, "Anti-debugging and anti-emulation techniques," *CodeBreakers Journal*, vol. 5, no. 1, 2008, `http://www.codebreakers-journal.com/`.

[10] Black Fenix, “Black Fenix’s anti-debugging tricks,” `http://in.fortunecity.com/skyscraper/browser/12/sicedete.html`.

[11] S. Cesare, “Linux anti-debugging techniques (fooling the debugger),” Jan. 1999, vX Heavens. `http://vx.netlux.org/lib/vsc04.html`.

[12] L. Julus, “Anti-debugging in win32,” 1999, vX Heavens. `http://vx.netlux.org/lib/vlj05.html`.

[13] H. Chang and M. Atallah, “Protecting software code by guards,” in *Security and Privacy in Digital Rights Management, ACM CCS-8 01, Philadelphia, PA, USA, November 5, 2001, Revised Papers*, 2001, pp. 160–175.

[14] J. Maebe and K. De Bosschere, “Instrumenting self-modifying code,” in *Proc. Fifth International Workshop on Automated Debugging (AADEBUG2003)*, Sep. 2003, pp. 103–113.

[15] P. Ferrie, “Prophet and loss,” *Virus Bulletin*, Sep. 2008, `www.virusbtn.com`.

[16] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers – Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, 1985.

[17] G. Balakrishnan, “Wysinwyx: What you see is not what you execute,” Ph.D. dissertation, Computer Science Department, University of Wisconsin, Madison, 2007.

[18] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *Proc. 13th. International Conference on Compiler Construction*, Mar. 2004, pp. 5–23.

[19] M. Harman and S. Danicic, “A new algorithm for slicing unstructured programs,” *Journal of Software Maintenance*, vol. 10, no. 6, pp. 415–441, 1998.

[20] M. Hind and A. Pioli, “Which pointer analysis should I use?” in *ISSTA ’00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000, pp. 113–123.

[21] D. Rayside, “Points-to analysis,” MIT OpenCourseWare, Massachusetts Institute of Technology. `http://ocw.mit.edu/`.

[22] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[23] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[24] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari, “Static analysis of binary code to isolate malicious behaviors,” in *Proc. IEEE 8th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE ’99)*, Jun. 1999, pp. 184–189.