Binary-Code Obfuscations in Prevalent Packer Tools

KEVIN A. ROUNDY and BARTON P. MILLER
University of Wisconsin

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive Software*

General Terms: Security

Additional Key Words and Phrases: Malware, obfuscation, program binary analysis

1. INTRODUCTION

Security analysts' understanding of the behavior and intent of malware samples depends on their ability to build high-level analysis products from the raw bytes of program binaries. Thus, the first steps in analyzing defensive malware are understanding what obfuscations are present in real-world malware binaries, how these obfuscations hinder analysis, and how they can be overcome. To this end, we present a thorough examination of the obfuscation techniques used by the packer tools that are most popular with malware authors [Bustamante 2008]. Though previous studies have discussed the current state of binary packing [Yason 2007], anti-debugging [Falliere 2007], and anti-unpacking [Ferrie 2008a] techniques, there have been no comprehensive studies of the obfuscation techniques that are applied to binary code. While some of the individual obfuscations that we discuss have been reported independently, this paper consolidates the discussion while adding substantial depth and breadth to it.

We describe obfuscations that make binary code difficult to discover (e.g., control-transfer obfuscations, exception-based control transfers, incremental code unpacking, code overwriting); to accurately disassemble into instructions (e.g., ambiguous code and data, disassembler fuzz-testing, non-returning calls); to structure into functions and basic blocks (e.g., obfuscated calls and returns, call-stack tampering, overlapping functions and basic blocks); to understand (e.g., obfuscated constants, calling-convention violations, chunked control-flow, do-nothing code); and to manipulate (e.g., self-checksumming, anti-relocation, stolen-bytes techniques). We also discuss techniques that mitigate the impact of these obfuscations on analysis tools such as disassemblers, decompilers, instrumenters, and emulators. This work is done in the context of our project to build tools for the analysis [Jacobson et al. 2011; Rosenblum et al. 2008] and instrumentation [Bernat and Miller 2011; Hollingsworth et al. 1994] of binaries, and builds on recent work that extends these analyses to malware binaries that are highly defensive [Bernat et al. 2011; Roundy and Miller 2010].

We begin by describing the methodology and tools we used to perform this study. We proceed to a taxonomy of the obfuscation techniques, along with current approaches to dealing with these techniques, and conclude by presenting statistics and observations on the various obfuscation techniques and tools.

2. METHODOLOGY

We used a combination of manual and automated analysis techniques for this study. We began by creating a set of defensive program binaries that incorporate the obfuscation techniques found in real-world malware. We created these binaries by obtaining the latest versions of the binary packer and protector tools that are most popular with malware authors [Bustamante 2008] and applying them to program binaries of various sizes. We carefully analyzed the binaries, paying special attention to the obfuscated bootstrap code with which the modified program unrolls the original binary payload into the address space, and to any changes that the obfuscation tool makes to the payload code itself. We report on obfuscations in this *metacode* that that the obfuscation tools add to the binaries they operate on.

We obtained most of our observations on these obfuscated binaries by adapting the Dyninst binary code analysis and instrumentation tool for the analysis of highly defensive program binaries, and then using

[1]February 21, 2012

it for that purpose [Bernat et al. 2011; Roundy and Miller 2010]. Adapting Dyninst to handle defensive binaries required a detailed understanding of the obfuscation techniques employed by these packer tools. Dyninst's ambitious analysis and instrumentation goals include the ability to discover, analyze, monitor, and modify binary code. Our obfuscation-resistant version of Dyninst applies parsing techniques to disassemble obfuscated code and constructs control-flow graphs (CFGs) and data-flow analyses of the program, updating these analyses at runtime based on observations of the monitored program's behavior. We stress-tested our tool's analysis and instrumentation techniques by instrumenting every memory access and every basic block in the obfuscated programs. Our instrumentation tool is designed to be resistant to any errors in the analysis [Bernat et al. 2011], however, our initial prototype was not, and therefore ran head-on into nearly every obfuscation technique employed by these programs [Roundy and Miller 2010]. We automatically generated statistical reports of defensive techniques employed by these packer tools with our obfuscation-resistant version of Dyninst, and we present those results in this study.

We also spent considerable time perusing each binary's obfuscated code by hand in the process of getting Dyninst to successfully analyze these binaries, aided by the OllyDbg [Yuschuk 2000] and IdaPro [Guilfanov 2011] interactive debuggers (Dyninst does not have a code-viewing GUI). In particular, we systematically studied the metacode of each packed binary to achieve a thorough understanding of its overall behavior and high-level obfuscation techniques.

3. THE OBFUSCATION TECHNIQUES

We structure this discussion around foundational binary analysis tasks. For each task, we describe solutions to those tasks, present defensive techniques that malware authors use to complicate the tasks, and survey any counter-measures that analysts take to deal with the defensive techniques.

We proceed by presenting foundational analysis tasks in the following sequence. The analyst must begin by finding the program binary's code bytes. The next task is to recover the program's machine-language instructions from the code bytes with disassembly techniques. The analyst can then group the disassembled bytes into functions by identifying function boundaries in the code. To modify and manipulate the code's execution, the analyst may patch code in the program binary. Finally, the analyst may attempt to bypass the defensive code in malware binaries by rewriting the binary to create a statically analyzable version of the program.

The analyst must begin by recovering the program binary's code bytes. From there, the analyst applies disassembly techniques to recover the program's machine-language instructions from the code bytes. The analyst then groups these instructions into functions by identifying function boundaries in the code. To modify and manipulate the code's execution, the analyst may patch code in the program binary. Finally, the analyst may attempt to bypass the defensive code in malware binaries by rewriting the binary to create a statically analyzable version of the program.

3.1 Binary Code Extraction

The most fundamental task presented to a binary analyst is to capture the program binary's code bytes so that the code itself can be analyzed. This is trivially accomplished on non-defensive binaries that do not generate code at run-time, because compilers typically put all of the code in a `.text` section that is clearly marked as the only executable section of the program binary. *Static analysis* techniques, which extract information from program files, can collect the program's code bytes by simply reading from the executable portions of the binary file. The code bytes of non-defensive binaries can also be extracted from the program's memory image at any point during its execution, as the code does not change at run-time and binary file formats clearly indicate which sections of the program are executable.

Binary code extraction becomes much harder, however, for programs that create and overwrite code at run-time. Defensive malware binaries are the biggest class of programs with this characteristic, though just-in-time (JIT) compilers such as the Java Virtual Machine [Lindholm and Yellin 1999] (which compiles java byte code into machine-language sequences just in time for them to execute) also fit this mold. Since the same analysis techniques apply both to obfuscated programs and JIT compilers, we discuss these techniques after our description of code packing and code overwriting.

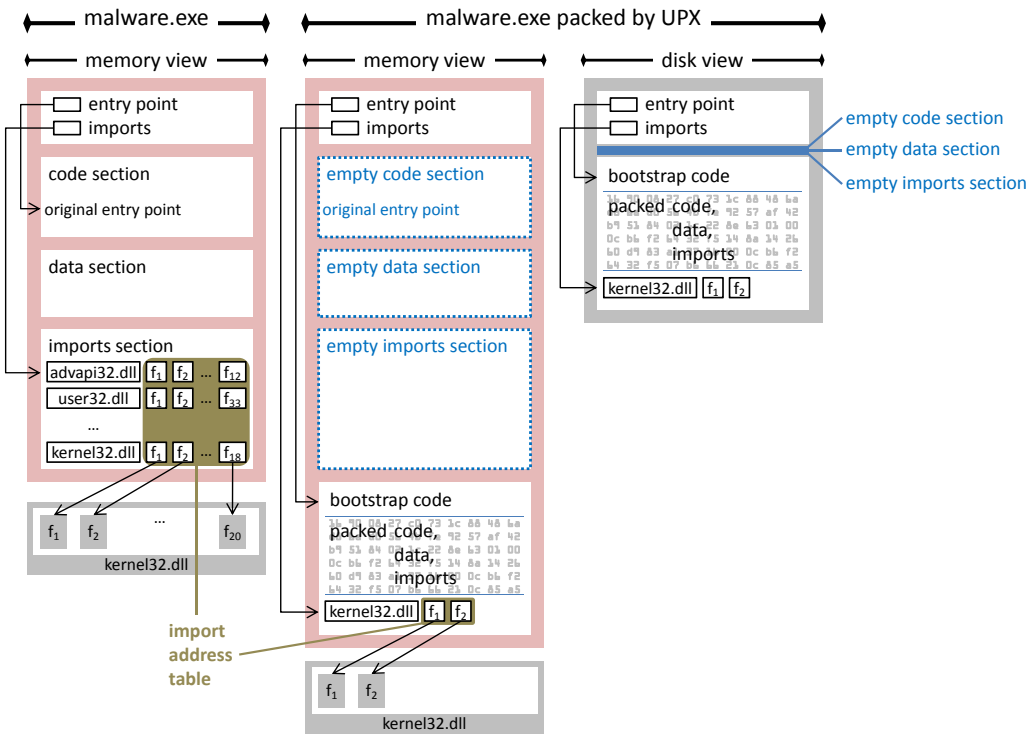

Fig. 1. Abstract view of a representative packing transformation performed by the UPX packer. UPX compresses malware.exe's code and data, setting the packed binary's entry point to its bootstrap code, which will unpack the code and data into memory at run-time. UPX replaces malware.exe's Import Table and Import Address Table with its own, much smaller tables that import only the LoadLibrary and GetProcAddress functions. UPX uses these functions to reconstruct the original import table at run-time.

3.1.1 *Code packing.* At least 75% of all malware binaries use code-packing techniques to protect their code from static analysis and modification [BitDefender 2007; Trilling 2008]. A packed binary is one that contains a payload of compressed or encrypted code that it unpacks into its address space at run-time. In practice, most malware authors incorporate this technique by compiling their code into a normal program binary and then processing the binary with a packer tool to create a packed version. Figure 1 illustrates the packing transformation performed by UPX; most other packer transformations can be thought of as elaborations on UPX's basic scheme. The packer tool sets the executable's entry point to the entry point of bootstrap code that unpacks the payload and then transfers control to the payload's original entry point (OEP). When the bootstrap code unrolls packed code and data, it places them at the same memory addresses that they occupied in the original binary so that position-dependent instructions do not move and data accesses find the data in their expected locations. UPX also packs the Portable Executable (PE) binary format's Import Table and Import Address Table (IAT) data structures, which list functions to import from other shared libraries. These tables are packed both because they would reveal significant information about the payload code and because they are highly amenable to compression. Upon loading a binary into memory, the Windows linker/loader processes these tables and writes imported function addresses into the IAT, but since packed binaries decompress the payload binary's import tables after load time, packer bootstrap code must fill in the payload binary's IAT itself.

The most common elaboration on this basic recipe for binary packing is one in which portions of the packer's bootstrap code itself are packed. Most packers use a small unpacking loop to decompress a more sophisticated decompression or decryption algorithm that unpacks the actual payload. This incremental approach achieves some space savings, but more importantly, it protects the bulk of the bootstrap code itself from static analysis. The latter is clearly the motivation for ASProtect, 99.1% of whose metacode is

dynamically unpacked, and for other similar "protector" tools that unpack their metacode in many stages, often at no space savings, and frequently overwriting code as they do so.

Approaches. Analysis tools use both static and dynamic techniques to retrieve packed code bytes. The widely used *X-Ray* technique [Perriot and Ferrie 2004] statically examines the program binary file with the aim of seeing through the compression and encryption transformations with which the payload code is packed. This technique leverages statistical properties of packed code to recognize compression algorithms and uses *known cipher-text attacks* to crack weak encryption schemes (i.e., the analyst packs a binary of their choice and therefore has the unencrypted payload in advance). The weakness of the X-Ray technique is its ineffectiveness against strong encryption and multiple layers of compression or encryption. An alternative static approach is to extract the portion of the bootstrap code that does the unpacking and use it to create an unpacker tool. A research prototype by Coogan et al. makes strides towards automating this process, and Debray and Patel built an improved prototype that incorporates dynamic analysis to help better identify and extract the code that does the unpacking [Coogan et al. 2009; Debray and Patel 2010]. Though promising, this approach has not been shown to work on a broad sample of real-world malware.

Dynamic analysis is an obvious fit for unpacked code extraction since packed binaries unpack themselves as they execute, and because this approach works equally well for JIT-style code generation. Dynamic binary translation and instrumentation techniques such as those used by Qemu [Bellard 2005] and DynamRIO [Bruening 2004] have no difficulty in finding dynamic code, since they do not attempt to discover the program's code until just before it executes. However, this approach does not distinguish between code that is statically present in the binary and code that is created at run-time. Dynamic unpacking tools such as Renovo [Kang et al. 2007] and EtherUnpack [Dinaburg et al. 2008] detect and capture unpacked code bytes by tracing the program's execution at a fine granularity and logging memory writes to identify written-then-executed code. They leverage the Qemu [Bellard 2005] whole-system emulator and the Xen virtual-machine monitor [Barham et al. 2003], respectively, which allows them to observe the execution of the monitored malware without being easily detected. The same approach of identifying written-then-executed code is used by unpackers that monitor programs with first-party dynamic instrumenters [Quist and Valsmith 2007], the debugger interface [Royal et al. 2006], interactive debugger tools [Guilfanov 2005; Prakash 2007; Stewart 2007], and sandboxed emulators [Graf 2005; Stepan 2005]. Unpacker tools that have monitored packed malware from the operating system track execution and memory writes at the coarser granularity of memory pages [Guo et al. 2008; Martignoni et al. 2007]. The aforementioned dynamic unpacker tools run packed programs either for a certain timeout period, or until they exhibit behavior that could indicate that they are done unpacking [Martignoni et al. 2007; Yegneswaran et al. 2008]. The primary limitations of the fine-grained monitoring techniques are that they only identify code bytes that actually executed and they incur orders-of magnitude slowdowns in execution time. Meanwhile, the efficient design of coarse-grained techniques makes them suitable for anti-virus products, but their coarse memory-page granularity means that they cannot identify the actual code bytes on unpacked code pages and that they cannot identify or capture overwritten code bytes.

The aforementioned dynamic unpacking techniques deliver the captured code bytes so that static analysis techniques can be used afterwards to recover the program's instructions and code structure. Our Dyninst instrumenter and the Bird interpreter [Nanda et al. 2006] instead apply static code-parsing techniques before the program executes, and use dynamic instrumentation to identify gaps in that analysis at run-time. Dyninst differs from Bird by capturing dynamically unpacked code and re-applying its code-parsing techniques (discussed in Section 3.2) to always provide users with structural analysis of the program's code prior to its execution.

3.1.2 *Code Overwriting. Self-modifying* programs move beyond unpacking by overwriting existing code with new code at run-time. Code overwrites often occur on the small end of the spectrum, affecting a single instruction, or even just a single instruction operand or opcode. For example, the ASPack packer modifies the push operand of a `push 0, ret` instruction sequence at run-time to push the OEP address onto the call stack and jump to it. On the other hand, the UPack packer's second unpacking loop unrolls payload code on top of its first unpacking loop, removing several basic blocks at once from the function that is currently executing. Code overwrites range anywhere from one byte to several kilobytes, but the packers we survey in this paper only overwrite their own metacode. More complex code overwriting scenarios are possible, for

example, the MoleBox packer tool and DarkParanoid virus [16 1998] repeatedly unpack sensitive code into a buffer, so that only one buffer-full of the protected code is exposed to the analyst at any given time. However, this approach is sufficiently hard to implement [Debray and Evans 2002] that relatively few malware binaries have attempted it.

Approaches. Code overwriting presents a problem to both static and dynamic approaches to binary code identification and analysis, as there is no point in time at which all of the program's code is present in the binary. For this reason, most unpacking tools do not capture overwritten bytes. The exception are tools that monitor the program's execution at a fine granularity and capture snapshots of each program basic block as soon as it executes [Anckaert et al. 2007; Kang et al. 2007].

Representing self-modifying code is challenging, as most binary analysis products do not account for code overwriting (e.g., control-flow graphs). Anckaert et al. propose an extended CFG representation that incorporates all versions of the code existing in the program at different points in time [Anckaert et al. 2007]. However, most CFG-building tools for defensive code are not equipped to build Anckaert-style CFGs, since they do not capture overwritten code bytes and do not build the CFG until after the program is done executing, when the overwritten code is gone [Madou et al. 2005; Yegneswaran et al. 2008]. Since Dyninst keeps its analysis up to date as the program executes, it could be used to generate an Anckaert-style CFG, but by default it maintains a normal CFG and provides CFG deltas whenever it updates its analysis in response to code unpacking or overwrites.

3.2 Disassembly

Once the code bytes have been captured, static analysis techniques can accurately disassemble most of the code in compiler-generated program binaries, even when those binaries have been stripped of all symbol information [Guilfanov 2011; Rosenblum et al. 2010]. The underlying technique employed by disassembly tools is to disassemble the binary code starting from known entry points into the program. *Linear-sweep* parsing [GNU Project - Free Software Foundation 2011; Schwarz et al. 2002] disassembles sequentially from the beginning of the code section and assumes that the section contains nothing but code. Since the code section is not always clearly indicated as such, and frequently contains non-code bytes such as string data and padding, this approach yields unsatisfying results. The alternate *recursive-traversal* approach [Sites et al. 1993; Theiling 2000] finds instructions by following all statically traversable paths through the program's control-flow starting from known function addresses. This technique is far more accurate, but misses the control transfer targets of instructions that determine their targets dynamically based on register values and memory contents. This weakness of recursive-traversal parsing is not significant for binaries that identify function entry address with symbol information, as most of the missing control transfer targets are to function entries or to jump table entries that can be identified through additional symbol information or static analysis [Cifuentes and Van Emmerik 1999]. Even for binaries that are stripped of symbol information, machine-learning techniques can identify enough function entry points (by recognizing instruction patterns that compilers use at function entries) to help recursive-traversal parsing achieve good code coverage [Rosenblum et al. 2008].

Unfortunately for the analyst, binary code can easily flout compiler conventions while remaining efficacious. Anti-disassembly techniques aim to violate the assumptions made by existing parsing algorithms so that they can both hide code from the disassembler and corrupt the analysis with non-code bytes. Defensive binaries remove all symbol information, leaving only one hint about the location of code in the binary: the executable's entry point. To limit the amount of code that can be found by following control transfer edges from this entry point, these binaries obfuscate their control flow. Compensating for poor code-coverage with compiler-specific knowledge is usually not an option since much of the code is hand-written assembly code and therefore, highly irregular. Additionally, code obfuscations deliberately blur the boundaries between code and non-code bytes to make it difficult to distinguish between the two [Collberg et al. 1998; Linn and Debray 2003; Popov et al. 2007]. We begin our discussion with anti-disassembly techniques that hide code and transition into techniques that corrupt the analysis with non-code bytes or uncover errors in the disassembler.

3.2.1 *Non-returning calls.* The `call` instruction's intended purpose is to jump to a function while pushing a return address onto the call stack so that the called function can use a `ret` instruction to pop the return

Call	Emulated Call
`call <target>`	`push <PC + sizeof(push) + sizeof(jmp)>` `jmp <target>`

(a)

Misused Call

```
 call <target>

.target
 pop <register-name>
```

(b)

Fig. 2. Part (a) illustrates a `call` and equivalent instruction sequence while Part (b) illustrates an unconventional use of the `call` instruction that gets a PC-relative value into a general-purpose register.

address from the top of the stack and jump there, resuming execution at the instruction following the `call`. However, the `call` instruction also lends itself to be used as an obfuscated `jmp` instruction; its semantics are equivalent to the `push-jmp` sequence of Figure 2a. Since the `call` instruction pushes a return address onto the stack, a misused `call` is usually paired with a `pop` instruction at the call target to remove the PC-relative return address from the stack (see Figure 2b). This easily implemented obfuscation attacks analysis tools in in three ways. First, parsers assume that there is a function at the call's target that will return to the call's *fall-through address* (i.e., the address immediately following the `call` instruction). Based on this assumption, recursive-traversal parsers assume that the bytes following a non-returning `call` instruction represent a valid instruction sequence, and erroneously parse them as such. Second, the attack breaks an important assumption made by code parsers for identifying function boundaries, namely, that the target of a `call` instruction belongs to a different function than that of the `call` instruction itself. Finally, a binary instrumenter cannot move the `call` instruction of such a `call-pop` sequence without changing the PC-relative address that the pop stores into a general-purpose register. Moving the call instruction without compensating for this change usually results in incorrect program execution, as packer metacode frequently uses the PC-relative address as a base pointer from which to access data. On average, 7% of all `call` instructions used in packer metacode are non-standard uses of the instruction, and as seen in Table I of Section 4, most packer tools use this obfuscation.

Approaches. The recursive-traversal parsing algorithm's assumption that there is valid code at the fall-through address of each `call` instruction means that it includes many non-code bytes in its analysis of obfuscated code. Unfortunately, removing this assumption drastically reduces the percentage of code that the parser can find, owing to the ubiquity of the `call` instruction and the fact that there usually is valid code at call fall-through addresses, even in obfuscated code. Researchers have proposed removing the recursive-traversal algorithm's assumption that calls return and compensating for the loss of code coverage through additional code-finding techniques. Kruegel et al. [Kruegel et al. 2004] compensate by speculatively parsing after call instructions and use their statistical model of real code sequences to determine whether the speculatively parsed instruction sequences are valid. Unfortunately, their tool targeted a specific obfuscator [Linn and Debray 2003] and its code-finding techniques rely heavily on the presence of specific instruction sequences at function entry points; most obfuscated code does not exhibit such regularity. Dyninst also modifies the recursive-traversal algorithm, to assume that a call does not return unless a binary slice [Cifuentes and Fraboulet 1997] of its called function can conclusively demonstrate that it does. When the binary slice of a called function is inconclusive, Dyninst does not parse at the call fall-through and instruments the `ret` instructions in the function to discover their targets at run-time. Though pure dynamic analysis approaches also identify instructions accurately in the face of non-returning calls, they only find those instructions that execute in a given run of the program.

3.2.2 *Call-stack tampering.* While non-returning calls involve non-standard uses of the `call` instruction, call-stack tampering adds non-standard uses of the `ret` instruction as well. Figure 3 illustrates three call-stack tampering tricks used by the ASProtect packer. Figure 3a shows an obfuscated `push-ret` instruction sequence that is used as an equivalent to a `jmp ADDR` instruction. Figure 3b is a somewhat more complex instruction sequence that is also a `jmp ADDR` equivalent. Figure 3c shows a function that increments its return address by a single byte, causing the program to resume its execution at a location that recursive-traversal parsing would not detect without additional analysis.

```
push ADDR
...
ret
```
(a)

```
push ADDR
call .foo
ret
.foo
ret
```
(b)

```
pop  ebp
inc  ebp
push ebp
ret
```
(c)

Fig. 3. Code sequences that tamper with the call stack. (a) and (b) are equivalent to `jmp ADDR`, while (c) shows a procedure that jumps to return-address + 1

Approaches. To our knowledge, ours is the only tool to apply static analysis techniques to `ret` target prediction, which allows us to improve the coverage and accuracy of our pre-execution analyses. Dyninst uses backwards slices at return instructions to determine whether a `ret` jumps to the function's return address as expected or to another address. When the slice fails to identify the `ret`'s target, Dyninst falls back on dynamic analysis to determine its target at run-time.

3.2.3 *Obfuscated control-transfer targets.* All of the packers in our study use indirect versions of the `call` and `jmp` instructions, which obfuscate control transfer targets by using register or memory values to determine their targets at run-time. Of course, even compiler-generated code contains indirect control transfers, but compilers use direct control transfers whenever possible (i.e., for transfers with a single statically known target) because of their faster execution times. One reason that packers often opt for indirect control transfers over their direct counterparts is that many packers compete for the bragging rights of producing the smallest packed binaries, and the IA-32 `call <register>` instruction is only 2 bytes long, while the direct `call <address>` instruction occupies 6 bytes. This small savings in packer metacode size is clearly a significant motivation for small packers, the smallest three of which choose indirect call instructions 92% of the time, while the remaining packers use them at a more moderate 10% rate. Obfuscation is also an important motivating factor, as many indirect control transfers go unresolved by static analysis tools. Obfuscators can increase the penalty for not analyzing an indirect control transfer by causing a single indirect control transfer to take on multiple targets [Linn and Debray 2003]. ASProtect uses this obfuscation more than any other packer that we have studied; our analysis of its code revealed 23 indirect call instructions with multiple targets.

Approaches. Indirect control-transfer targets can be identified inexpensively when they get their targets from known data structures (e.g., the Import Address Table) or read-only memory. By contrast, static analysis of indirect control-transfer targets is particularly difficult in packed binaries, as they frequently use instructions whose targets depend on register values, and because these binaries typically allow writes to all of their sections. Though value-set analyses [Balakrishnan and Reps 2004] could theoretically reveal all possible targets for such indirect control transfer instructions, this technique requires a complete static analysis of all memory-writing instructions in the program, and therefore does not work on binaries that employ code unpacking or code overwrites. Because of these difficulties in resolving obfuscated control transfers in the general case, most static analysis tools restrict their pointer analyses to standard uses of indirect calls and jumps that access the IAT or implement jump tables [Cifuentes and Van Emmerik 1999; Guilfanov 2011; Yuschuk 2000]. While dynamic analysis trivially discovers targets of indirect control transfers that execute, it may leave a significant fraction of the code un-executed and usually will not discover all targets of multi-way control transfer instructions. Researchers have addressed this weakness with techniques that force the program's execution down multiple execution paths [Babic et al. 2011; Moser et al. 2007], but even these extremely resource intensive techniques do not achieve perfect code coverage [Sharif et al. 2008].

3.2.4 *Exception-based control transfers.* Signal- and exception-handling mechanisms allow for the creation of obfuscated control transfers whose source instruction and target address are well-hidden from static analysis techniques [Popov et al. 2007]. Statically identifying the sources of such control transfers requires predicting which instructions will raise exceptions, a problem that is difficult both in theory and in practice [Muth and Debray 2000]. This means that current disassembly algorithms will usually parse through fault-raising instructions into what may be non-code bytes that never execute. Another problem is that on Windows it is hard to find the exception handlers, since they can be registered on the call stack at run-time

with no need to perform any Windows API or system calls. An additional difficulty is that the exception handler specifies the address at which the system should resume the program's execution, which constitutes yet another hidden control transfer.

Approaches. Though static analyses will often fail to recognize exception-based control transfers, these transfers can be detected by two simple dynamic analysis techniques. The OS-provided debugger interface provides the most straightforward technique, as it informs the debugger process of any fault-raising instructions and identifies all registered exception handlers whenever a fault occurs. However, use of the debugger interface can be detected unless extensive precautions are taken [Falliere 2007], so the analysis tool may instead choose to register an exception handler in the malware binary itself and ensure that this handler will always execute before any of the malware's own handlers. On Windows binaries, the latter technique can be implemented by registering a Vectored Exception Handler (vectored handlers execute before Structured Exception Handlers) and by intercepting the malware's attempts to register vectored handlers of its own through the AddVectoredExceptionHandler API function provided by Windows. Whenever the analysis tool intercepts a call to this API function, the tool can keep its handler on top of the stack of registered vectored handlers by unregistering its handler, registering the malware handler, and then re-registering its own handler.

The analysis tool must also discover the address at which the exception handler instructs the OS to resume the program's execution. The handler specifies this address by setting the program counter value in its exception-context-structure parameter. Analysis tools can therefore identify the exception handler's target by instrumenting the handler at its exit points to extract the PC value from the context structure.

3.2.5 *Ambiguous code and data.* Unfortunately, there are yet more ways to introduce ambiguities between code and non-code bytes that cause problems for code parsers. One such technique involves the use of conditional branches to introduce a fork in the program's control flow with only one path that ever executes, while *junk-code* (i.e., non-code or fake code bytes) populate the other path. This technique can be combined with the use of an opaque branch-decision predicate that is resistant to static analysis to make it difficult to identify the valid path [Collberg et al. 1998]. Surprisingly, we have not conclusively identified any uses of this well-known obfuscation in packer metacode, but the similarities between this obfsucation and valid error handling code that executes rarely, if ever, may be preventing us from identifying instances of it.

A far more prevalent source of code and data ambiguity arises at transitions to regions that are populated with unpacked code at run-time. In some cases the transitions to these regions involve a control transfer instruction whose target is obviously invalid. For example, the last instruction of UPX bootstrap code jumps to an uninitialized memory region (refer back to Figure 1), an obvious indication that unpacking will occur at run-time and that the target region should not be analyzed statically. However, binaries often contain data at control transfer targets and fall-through addresses that will be replaced by unpacked code at run-time. The most problematic transitions from code to junk bytes occur when the transition occurs in the middle of a straight-line sequence of code; ASProtect's seven sequentially layed-out unpacking loops provide perhaps the most compelling example of this. ASProtect's initial unpacking loop is present in the binary, and the basic block at the loop's exit edge begins with valid instructions that transition into junk bytes with no intervening control transfer instruction. As the first loop executes, it performs in-place decryption of the junk bytes, revealing the subsequent unpacking loop. When the second loop executes, it decrypts the third loop and falls through into it, and so on until the seven loops have all been unpacked. Thus, to accurately disassemble such code, the disassembler should detect transitions between code bytes and junk bytes in straight-line code. These transitions are hard to detect because the IA-32 instruction set is sufficiently dense that random byte patterns disassemble into mostly valid instructions.

Approaches. Using current techniques, the only way to identify code with perfect exclusion of data is to disassemble only those instructions that appear in an execution trace of a program. Since this technique achieves poor code coverage, analysts often turn to techniques that improve coverage while limiting the amount of junk bytes that are mistakenly parsed as code. Another approach to dealing with ambiguity arising from transitions into regions that will contain dynamically unpacked code is to avoid the problem altogether by disassembling the code after the program has finished unpacking itself. However, this approach is not suitable for applications that use the analysis at run-time (e.g., for dynamic instrumentation), does

not generalize to junk code detection at opaque branch targets, and doesn't capture overwritten code. Not capturing overwritten code is significant because many packers overwrite or de-allocate code buffers immediately after use.

Code parsing techniques can cope with ambiguity by leveraging the fact that, though random bytes usually disassemble into valid x86 instructions, they do not share all of the characteristics of real code. Kruegel et al. build heuristics based on instruction probabilities and properties of normal control-flow graphs to distinguish between code and junk bytes [Kruegel et al. 2004]. Unfortunately, they designed their techniques for a specific obfuscator that operates exclusively on GCC-generated binaries [Linn and Debray 2003] and their methods rely on idiosyncrasies of both GCC and Linn and Debray's obfsucator, so it is not clear how well their techniques would generalize to arbitrarily obfuscated code. Dyninst uses a cost-effective approach to detect when its code parser has transitioned into invalid code bytes; it stops the disassembler when it encounters privileged and infrequently used instructions that are usually indicative of a bad parse. This technique excludes most junk bytes and takes advantage of Dyninst's hybrid analysis mechanisms; Dyninst triggers further disassembly past a rare instruction if and when the rare instruction proves itself to be real code by executing.

3.2.6 *Disassembler fuzz-testing.* Fuzz-testing [Miller et al. 1990] refers to the practice of stress testing a software component by feeding it large quantities of unusual inputs in the hope of detecting a case that the component handles incorrectly. IA-32 disassemblers are particularly vulnerable to fuzz-testing owing to the complexity and sheer size of the instruction set, many of whose instructions are rarely used by conventional code. By fuzz-testing binary-code disassemblers, packer tools can cause the disassembler to mis-parse instructions or mistake valid instructions for invalid ones. A simultaneous benefit of fuzz-testing is that it may also reveal errors in tools that depend on the ability to correctly interpret instruction semantics (e.g., sandbox emulators and taint analyzers), which have a harder task than the disassembler's job of merely identifying the correct instructions. For example, one of the dark corners of the IA-32 instruction set involves the optional use of instruction prefixes that serve various purposes such as locking, segment-selection, and looping. Depending on the instruction to which these prefixes are applied, they may function as expected, be ignored, or cause a fault. The ASProtect packer does particularly thorough fuzz-testing with segment prefixes, while Armadillo uses invalid lock prefixes as triggers for exception-based control flow. Other inputs used for fuzz-testing include instructions that are rare in the context of malware binaries, and that might therefore be incorrectly disassembled or emulated (e.g., floating point instructions).

Various factors increase the thoroughness of disassembler fuzz-testing. The polymorphic approach whereby PolyEnE and other packers generate bootstrap code allows them to create new variations of equivalent sequences of rare instructions each time they pack a binary. In the most aggressive uses of polymorphic code, the packed binary itself carries the polymorphism engine, so that each execution of the packed binary fuzz-tests analysis tools with different permutations of rare instructions (e.g., ASProtect, EXEcryptor). Finally, packers include truly random fuzz-testing by tricking disassembly algorithms into mis-identifying junk bytes as code, as we have discussed in Section 3.2.5. This last fuzz-testing technique stresses the disassembler more thoroughly than techniques that fuzz instructions that must actually execute, as the non-code bytes disassemble into instructions that would cause program failures if they actually executed (e.g., because the instructions are privileged, invalid, or reference inaccessible memory locations). On the other hand, the repercussions of incorrectly disassembling junk instructions are usually less significant.

Approaches. Packed malware's use of fuzz-testing means that disassemblers, emulators, and binary translators must not cut corners in their implementation and testing efforts, and that it is usually wiser to leverage an existing, mature disassembler (e.g., XED [Charney 2010], Ida Pro [Guilfanov 2011], ParseAPI [Paradyn Tools Project 2011]) than to write one from scratch. Correctly interpreting instruction semantics is an even more difficult task, but also one for which mature open-source tools are available (e.g., Rose [Quinlan 2000], Qemu [Bellard 2005]).

3.3 Function-Boundary Detection

In compiler-generated code, function-boundary detection is is aided by the presence of `call` and `ret` instructions that identify function entry and exit points. Analysis tools can safely assume that the target of each `call` is the entry point of a function (or thunk, but these are easy to identify) and that `ret` instructions are

only used at function exit points. Function boundary detection is not trivial, however, as it is not safe to assume that every function call is made with a `call` instruction or that every exit point is marked by a `ret`. Tail-call optimizations are most often to blame for these inconsistencies. At a tail call, which is a function call immediately followed by a return, the compiler often substitutes a `call <addr>, ret` instruction pair for a single `jmp <addr>` instruction. This means that if function foo makes a tail call to function bar, bar's `ret` instructions will bypass foo, transfering control to the return address of the function that called foo. A naïve code parser confronted with this optimization would confuse foo's call to bar with an intraprocedural jump. Fortunately, parsers can usually recognize this optimization by detecting that the jump target corresponds to a function entry address, either because there is symbol information for the function, or because the function is targeted by other call instructions [Hollingsworth et al. 1994].

Function-boundary-detection techniques should also not expect the compiler to lay out a function's basic blocks in an obvious way, viz., in a contiguous range in the binary with the function's entry block preceding all other basic blocks. A common reason for which compilers break this assumption is to share basic blocks between multiple functions. The most common block-sharing scenarios are functions with optional setup code at their entry points and functions that share epilogues at their exit points. An example of optional setup code is provided by Linux's libc, several of whose exported functions have two entry points; the first entry point is called by external libraries and sets a mutex before merging with the code at the second entry point, which is called from functions that are internal to libc and have already acquired the mutex. Meanwhile, some compilers share function epilogues that perform the task of restoring register values that have been saved on the call stack. As we will see in this section, function boundary identification techniques for obfuscated code must confront the aforementioned challenges as well as further violations of `call` and `ret` usage conventions, scrambled function layouts, and extensive code sharing between functions.

3.3.1 *Obfuscated calls and returns.* As we noted in our discussion of non-returning calls, the semantics of `call` and `ret` instructions can be simulated by alternative instruction sequences. When `call/ret` instructions are used out of context, they create the illusion of additional functions. On the other hand, replacing `call/ret` instructions with equivalent instruction sequences creates the illusion of fewer functions than the program actually contains. By far the most common of the two obfuscations is the use of `call` and `ret` instructions where a `jmp` is more appropriate. Since the `call` and `ret` instructions respectively push and pop values from the call stack, using these instructions as jumps involves tampering with the call stack to compensate for these side-effects. Though there is a great deal of variety in how the call-stack is tampered with, most of the examples we have seen are variations on the basic examples that we presented in Figure 3 to illustrate call-stack tampering.

Approaches. Lakhotia et al. designed a static analysis technique to correctly detect function-call boundaries even when `call` and `ret` instructions have been replaced with equivalent instruction sequences [Lakhotia et al. 2005]. Dyninst addresses the opposite problem of superfluous `call` instructions with look-ahead parsing at each call-target through the first multi-instruction basic block, to determine whether the program removes the return address from the call stack. Though this is not a rigorous solution to the problem, it works well in practice for the packer tools we have studied.

3.3.2 *Overlapping functions & basic blocks.* Of the 12 packer tools we selected for this study, 7 share code between functions. As in compiler-generated code, the most common use of code sharing is for optional function preambles and epilogues. Some packed binaries achieve tiny bootstrap code sizes by outlining short instruction sequences (i.e., moving them out of the main line of the function's code) into mini-functions and calling into various places in these functions to access only those instructions that are needed. Outlining code in this way results in a lot of code sharing and strange function layouts; for example, the function's entry block is often at a larger address than thoase of other basic blocks belonging to the function.

Some packers (e.g., EXEcryptor) extensively interleave the blocks of different functions with one another and spread function blocks over large address ranges in the binary. To make matters worse, Yoda's Protector and other packers fragment the basic blocks of some of their functions into chunks of only one or two instructions and spread these around the code section of the binary.

Since a program basic block is defined as a sequence of instructions with a single entry point and a single exit point, one might assume that basic blocks cannot overlap each other. However, in dense variable-length

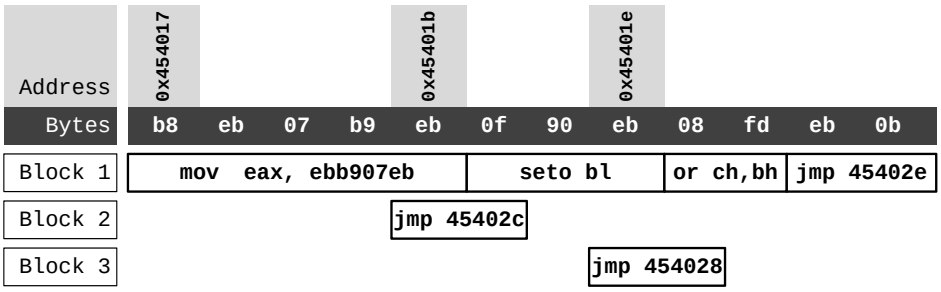

Fig. 4. An example of overlapping instructions and basic blocks taken from the obfuscated bootstrap code of a binary packed by Armadillo. All three basic blocks actually execute.

instruction sets such as IA-32, valid instructions can overlap when they start at different offsets and share code bytes (see Figure 4). Since the overlapping code sequences share the same code range but not the same instructions, they constitute separate basic blocks if the program's control flow is arranged such that each of the overlapping blocks can execute. Basic-block overlap is frequently dismissed as being too rare to be of any real concern, yet we have observed it in 3 of the 12 obfuscation tools in our study and in a custom obfuscation employed by the Conficker worm. The primary purpose of overlapping basic blocks in packed binary code is to trip up analysis tools that do not account for this possibility [Nanda et al. 2006; Vigna 2007] and to hide code from the analyst, as most disassemblers will only show one of the overlapping code sequences.

Approaches. Function-block interleaving makes it difficult for analysts to view a whole function at once, as most disassembly tools show code in a small contiguous range. Ida Pro is a notable exception; it statically analyzes binary code to build a control-flow graph and can show the function's disassembly structured graphically by its intraprocedural CFG [Guilfanov 2011]. Unfortunately, Ida Pro does not update its CFG as the program executes, and therefore it does not produce CFG views for code that is hidden from static analysis (e.g., by means of code-packing, code-overwriting, control-flow obfuscations, etc.). Dyninst does update its CFG of the program at run-time, but lacks a GUI for interactive perusal of the disassembled code. A marriage of Ida Pro's GUI and Dyninst's techniques for updating CFGs would allow for easy perusal of functions with interleaved blocks, but no such tool exists at present. With regard to the problem of overlapping basic blocks, some analysis tools do not account for this possibility and therefore their data structures make the assumption that 0 or 1 basic blocks and instructions correspond to any given code address [Nanda et al. 2006; Vigna 2007]. Since multiple blocks and instructions may indeed map to any given address (see Figure 4), tool data structures should be designed to account for this possibility.

3.4 Code Comprehension

Despite the existence of many tools that automate important analysis tasks, human analysts spend a lot of time manually perusing binary code. Building on the analysis tasks of previous sections, the analyst recovers the program's machine-language instructions and views them as assembly-language instructions or decompiled programming-language statements. The visual representation of the code itself is often supplemented by views of reconstructed programming-language constructs such as functions, loops, structs, and variables. By using an interactive debugging tool like IdaPro [Guilfanov 2011] or OllyDbg [Yuschuk 2000], the analyst can interact with the program's execution, either by single-stepping the program's execution, patching in breakpoints, modifying sequences of code or data, and/or tracing its behavior at single-instruction granularity.

In this section we discuss obfuscations that directly affect a human analyst's ability to understand binary code. This list will be brief because we have already presented many of the obfuscations that impact code comprehension in previous sections. For example, even after the analyst successfully extracts the code from a packed, self-modifying binary, the dynamic nature of the code comprehend makes it difficult to comprehend. Similarly, obfuscations that disassembly gaps and errors make the code difficult to understand,

Not obfuscated	Obfuscated
`mov ecx, 0x294a`	`mov ecx, 0x410c4b` `sub ecx, 0x40e301`

Not obfuscated	Obfuscated
`call <next>`	
`pop ebp`	
`sub ebp, 0x40e207`	`mov ebp, -0xab7`

Not obfuscated	Obfuscated
`mov edi, ptr[eax+a4]`	
`rol edi, 7`	`mov edi, <OEP>`

Fig. 5. Simple examples of program-constant obfuscations taken from metacode produced by the Yoda's Protector packer. In each case, the packer simulates the behavior of a simple instruction with a constant operand with a sequence of instructions that hide the constant, thereby slowing down the analyst.

and obfuscations that confuse weak disassemblers confuse non-expert analysts as well. Finally, unusual function layouts and obfuscated function boundaries make it hard to identify functions and their relationships to one another, code is easiest to understand when it is structured the way it is written, viz., as functions with well-defined interactions. We begin by discussing obfuscated constant operands in machine-language instructions followed by a discussion of calling-convention violations. We then describe uses of do-nothing code in obfuscated programs.

3.4.1 *Obfuscated constants.* Half of the packers we have studied obfuscate some constants in machine language instructions, and a third of the packers obfuscate program extensively. The constant that packers most-frequently obfuscate is the address of the original entry point of the payload code. Obfuscating uses of the OEP address prevents analysts from discovering the workings of a packer tool by packing a binary of their choice for which they know the OEP address beforehand, and then searching for the known OEP address in the program's code and data. The degree of obfuscation applied to the OEP and other program constants ranges from simple examples like those of Figure 5 to more elaborate encryption algorithms.

Approaches. Constant obfuscations make the code harder to understand, with the goal of slowing down analysts that are trying make sense of the program's instructions. Fortunately, decompilers offer an automated solution to dealing with this problem. In the process of replacing machine-language insructions with programming language statements, decompilers translate the instructions into an intermediate representation on which they apply basic arithmetic rules to reduce complex operations into simpler forms [**?**; **?**; **?**]. This approach produces programming language statements from which most constant-based obfuscations have eliminated. Unfortunately, current decompilers do not work on code that is dynamically unpacked at runtime, so the analyst must first reverse engineer the program so that dynamically unpacked code is statically present in the rewritten binary. For programs that do not employ the anti-unpacking techniques that we will discuss in Section 0**??** there are general-purpose unpacking tools that may be able to automate the reverse-engineering task for the binaries [**?**; Yegneswaran et al. 2008]. However, for the remaining packed programs, constant obfuscations lead to a chicken and egg problem, since most the most heavily obfuscated constant is the OEP address, which the reverse-engineer needs to find to reverse-engineer the binary and decompile the code. In these cases, analysts must turn to techniques that are more manual-labor intensive. For example, analysts may open up a calculator program and do arithmetic by hand or they may force the obfuscated instructions to execute, which they often do if they are already using an interactive debugger program such as Ida Pro to view the obfuscated disassembly.

3.4.2 *Calling-convention violations.* Calling conventions standardize the contract between caller functions and their callees by specifying such things as where where to store function parameters and return values, which of the caller/callee is responsible for clearing parameter values from the call stack, and which of the architecture's general-purpose registers can be overwritten by the called function. Despite the large number and variety in x86 calling-conventions [Fog 2011], all of these conventions agree on certain points, notably, that status register flags are volatile across function-call boundaries and that they do not store function return values.

Though many packed programs contain some compiler-generated metacode that adheres to standardized calling conventions, nearly all of them contain packer metacode that does not. Even across the metacode of

a single packed binary there are usually no calling conventions that hold, as much the code is hand-written and has high variability. Rather than having well-defined parameters and return values, the called functions frequently operate directly on register values. Furthermore, caller functions sometimes make branching decisions based on status-register flags set by comparisons made in the called function The lack of standardized conventions causes problems for analysis tools and human analysts that have built up assumptions about calling conventions based on compiler-generated code. For example, a human analyst may incorrectly expect register values to be unmodified across function call boundaries. Similarly, for the sake of efficiency, binary instrumentation tools may modify binary code in a way that modifies status-register flags in a called function, based on the assumption the flags will not be read by the caller function. This is a safe assumption in most compiler-generated code, but on obfuscated code the instrumenter may unintenionally alter the program's behavior by modifying the flags in the status register (Dyninst used to have this problem).

Approaches. Though analysis tools such as disassemblers and instrumenters can make significant improvements in such metrics as code coverage [**?**; Kruegel et al. 2004] and instrumentation efficiency [Hollingsworth et al. 1994] by making assumptions based on calling conventions, they may limit their applicability to obfuscated code by doing so. Analysis tools that work correctly on obfuscated code usually deal with calling-convention violations by using pessimistic assumptions at function boundaries. For example, many binary instrumenters either make the pessimistic assumption that no register values are dead across function boundaries (e.g., DIOTA [Maebe et al. 2002], DynamoRIO [Bruening 2004], PIN [Luk et al. 2005]), or assume them to be dead only if they are proved to be so by a binary slice on the given register (Dyninst [Bernat et al. 2011]).

3.4.3 *Do-nothing code.* Most obfuscation techniques protect sensitive code from analysis and reverse engineering by making the hcode hard to access or understand. Don-nothing code is an alternative strategy that hides sensitive code by diluting the program with semantic no-ops. As this do-nothing code must appear to do useful work to attract the analyst's attention, it is usually heavily obfuscated. An alternative method of distracting the analyst from important code is the use of "do-little" rather than actual do-nothing code. This may include calculating a value that is used for some later computation in a roundabout way, in the spirit of Ebringer's time-lock puzzles [**?**]. Do-little code has the benefit of preventing the analyst from ignoring or even eliding the obfuscated code from the program's execution. Do-nothing code is really only needed in packers with small bootstrap code sizes, as the larger "protector" tools contain so much code that is designed to cause problems for analysis tools that the code that is actually responsible for unpacking is already a very small fraction of the packer metacode. As such, only PolyEnE and a few other packers employ this strategy, and usually make the inclusion of do-nothing code optional, since users of packer tools often wish the packed binaries to be small.

Approaches. Christodorescu et al. developed techniques to detect and remove semantic-nops in a rewritten version of the malware binary. Their techniques also normalize control-flow graphs that have been chunked into tiny basic blocks and account for a single layer of code packing, but they did not account for common defensive techniques such as code overwriting and anti-patching (see Section 0**??**) and they did not build a robust implementation of their techniques [Christodorescu et al. 2005]. Do-nothing code that never executes can be eliminated by the malware normalization techniques that Bruschi et al. built into the Boomerang decompiler [**?**]. They apply arithmetic rules to reduce the conditional predicates of branch instructions to identify branches that always take one path, and eliminate the other path from the CFG and the decompiled code. Unfortunately, their techniques do not apply to packed binaries.

Malware analysts can avoid studying do-nothing code by starting from a high-level summary of program behavior (e.g., a log of Windows API [Yegneswaran et al. 2008] or system calls [**?**]) and using this to focus in on the interesting parts of the code. This technique is successful when do-nothing code and do-little code are not embedded into code that is of interest to the analyst.

3.5 Code Patching

Code-patching techniques support a variety of dynamic analysis tasks by modifying and adding to the program's code and data bytes. In particular, malware analysts frequently use code-patching techniques to monitor malware binaries at a coarse granularity, most often by modifying the entry points of system

libraries to log the malware's use of Windows API functions [Bayer et al. 2006; Hunt and Brubacher 1999; Willems et al. 2007; Yegneswaran et al. 2008]. As mentioned in the previous section, the Windows API traces obtained by this technique give analysts a high-level view of the program's behavior. These high-level views of malware behavior are usually supplemented with detailed studies of the code that are most-often obtained by loading the malware in an interactive debugger program such as Ida Pro and closely monitoring its execution through the use of software breakpoints.

Software breakpoints rely on code patching techniques and are used both in tracing tools and interactive debuggers. The actual software-breakpoint mechanism overwrites the instruction at the breakpoint address with the single-byte `int 3` breakpoint instruction. When an `int 3` breakpoint executes, it interrupts the program's execution and alerts an attached debugger process (or exception handler) of the event. The debugger process (or the analyst's signal handler) then the desired analysis task and removes the breakpoint so that the program can continue its execution. Because software breakpoints have a large execution-time cost, analysts frequently modify binary code by patching it with jump instructions instead of trap instructions [Hollingsworth et al. 1994; Hunt and Brubacher 1999]. This technique allows tools to jump to analysis code and back again with little execution-time overhead, but is harder to implement correctly because x86 long-jump instructions are 6 bytes long and may overwrite multiple original instructions.

Code patching techniques can be applied statically, to the binary file, or dynamically, modifying the program binary's image in memory. Code-patching of malware binaries usually is usually dynamic, since the binaries are usually packed and most of the code in the binary file is in a compressed or encrypted form, making code patches difficult to apply statically. However, even dynamic patch-based techniques are not readily applied to malware, since these binaries employ techniques that are specifically designed to detect code patching.

3.5.1 *Stolen bytes.* Figure 6b illustrates the stolen-bytes technique pioneered by the ASProtect packer. This technique obfuscates calls to shared-library functions and circumvents patch-based tracing of functions in shared libraries, including Windows API functions. Patch-based tracing of Windows API functions typically replaces the first instruction of the function with an `int3` or `jmp` instruction that transfers control to the instrumentation code [Hunt and Brubacher 1999; Yegneswaran et al. 2008]. The stolen-bytes technique bypasses the patched entry point by creating a copy of the imported function's first basic block and routing the program's control flow through the copy instead of the patched original block. Since this technique must read from the shared library to "steal" the first basic block from the imported function, the byte stealing occurs at run-time, after the shared library has been loaded and calls to the imported function have been linked through the Import Address Table. This technique must therefore modify indirect calls to the imported function so that they target the "stolen" copy of the function's entry block. As shown in the figure, ASProtect achieves this by overwriting calls that access the function's IAT entry with calls that directly target the stolen block; other packers achieve the same effect by writing the stolen block's address into the imported function's IAT entry, which allows them to leave the original call instructions intact. The packer completes the detour by pointing the control transfer that ends the stolen block at the subsequent blocks of the original function.

Approaches. Tools that instrument Windows API and other shared library functions with patch-based techniques (e.g., CWSandbox [Willems et al. 2007], Detours [Hunt and Brubacher 1999], TTAnalyze [Bayer et al. 2006]) have no effect on the program's execution, as the Windows API function's patched first block will not execute. Our Dyninst tool performs similar patch-based instrumentation on some Windows API functions, but adds techniques to keep its instrumentation of those functions from being bypassed. To detect API-function invocations, Dyninst instruments the control transfers that enter the function, detecting transfers into any basic block belonging to the API function. Dyninst also ensures that when the packed program is stealing bytes from Windows API functions it steals original code bytes and not any patches that Dyninst might place at the function's entry points. Dyninst hides these patches by emulating all memory accesses that read from API functions so that they read from an unmodified shadow copy of the function's code bytes.

There are multiple alternative instrumentation and program-monitoring techniques that do not rely on code patching at all. For example, analysts can avoid triggering anti-patching techniques by tracing or instrumenting the code with tools that do not patch the binary code and instead use just-in-time binary

translation techniques to execute and instrument the monitored programs (e.g., DIOTA [Maebe et al. 2002], DynamoRIO [Bruening 2004]). However, these techniques can be detected because they allocate extra space for instrumentation code in the program's address space that would not normally be there [Bernat et al. 2011]. One avenue through which analysts have avoided this problem is by building tools that apply binary translation techniques to the entire monitored system (e.g., Qemu [Bellard 2005] has provided the foundation for several malware analysis tools [Bayer et al. 2006; **?**; Moser et al. 2007]). Whole-system monitoring has also been achieved by tools that extend virtual machine hypervisors [Dinaburg et al. 2008]. Though it is not possible to monitor malware's execution in a provably undetectable way [Dinaburg et al. 2008], whole-system monitoring comes closest to this goal, with the added benefit that an infected guest system can be rolled back to a clean checkpoint. Its primary drawback is that monitoring tools that sit outside of the monitored system must employ virtual machine introspection techniques to make sense of what is happening inside the monitored system. Emerging introspection tools such as LibVMI [**?**], which originated as teh XenAccess project [**?**], perform the necessary introspection techniques for certain guest systems, but may not be compatible with other monitoring tools.

3.5.2 *Self-checksumming.* Many packed binaries verify that their code has not been tampered with by applying self-checksumming techniques. These packers take a checksum over the program's code bytes and then recalculate that checksum at run-time to detect modifications to the program's code and data. Packers may self-checksum their own packer bootstrap code (e.g., ASProtect), the program's payload once it has been unpacked (e.g., PECompact), and the program binary file (e.g., Yoda's Protector). Checksums over packer bootstrap code help to detect attempts to reverse-engineer the binary unpacking process, while checksums of the program's payload code protect the payload from tampering and reverse-engineering. Finally, checksums of the binary file protect both the packer metacode and the payload code. Frequently, the packers read from their own code without explicitly intending to perform self-checksumming. For example, the stolen-bytes technique reads binary code to "steals" code bytes at function entry points, which can cause strange behavior if those functions have been patched.

Approaches. The standard method of defeating self-checksumming techniques (pioneered by Wurster et al. [Wurster et al. 2005]) is to redirect all memory accesses at an unmodified shadow copy of the program's code bytes, while executing the patched code bytes. Wurster et al. accomplished this by modifying the operating system's virtual memory management so that the ITLB caches patched code pages while the DTLB caches unmodified code pages. Rosenblum et al. showed that the same techniques can be achieved with a modified virtual machine monitor [Rosenblum et al. 2008]. This TLB-based approach becomes more difficult for programs that unpack or overwrite their code at run-time however [Giffin et al. 2005], as the dynamically written code bytes must be present in the execution-context memory pages as well so that they can execute. Dyninst copies changes in the shadow memory to the patched memory before any control transfer into dynamically unpacked code, and whenever existing code is overwritten. Dyninst redirects the program's memory accesses by emulating memory-access instructions to point them at the shadow memory.

3.6 Unpacking

The analyst's goal is to subject the binary to static analysis and code patching by rewriting the packed program binary such that its dynamically unpacked code is present in the binary file. The three subtasks involved in creating an unpacked binary are: reconstructing an executable file that contains the unpacked code, modifying the packed binary to bypass the packer's bootstrap code, and reconstructing the payload binary's imported function data structures so that the Windows loader can link up calls to imported functions. Solving the first task requires identifying the payload code's Original Entry Point and executing the program until it jumps to the OEP, by which time the bootstrap code is done executing and the binary has been unpacked; at this point the analyst can copy the program's code and data bytes from the program's memory into a reconstructed program binary. The analyst accomplishes the task of bypassing the packer's bootstrap code by setting the reconstructed binary's entry point to the OEP. Finally, the analyst reconstructs the original import function data structures from the Import Address Table entries used by the unpacked payload code.

Since many customers of packer tools wish to prevent these reverse-engineering efforts, many binary packers take steps to counter them. Here we describe the anti-unpacking techniques used by the 12 tools of this

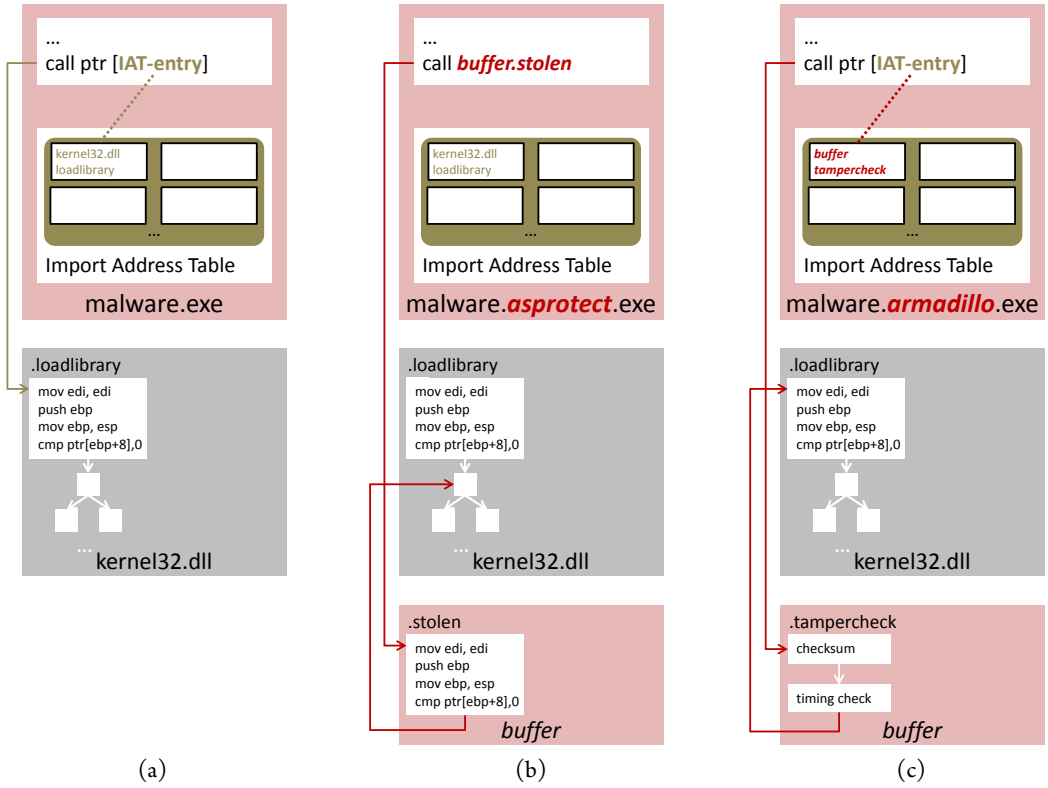

Fig. 6. Packed programs may interpose on inter-library calls originating from their packed payload to resist reverse-engineering and code-patching techniques. In a non-packed program, as in part (a), calls to functions in shared libraries use indirect call instructions to read their targets in from Import Address Table entries. Part (b) illustrates ASProtect's implementation of the stolen-bytes technique, which evades binary instrumentation at the entry points of functions in shared libraries by making a copy of the first block of an imported function and redirecting the call instruction to point at the "stolen" block. Part (c) illustrates Armadillo's method of hooking inter-library calls by replacing IAT addresses with the address of an Armadillo function that performs anti-tampering techniques before forwarding the call on to its intended destination.

study; for a broader discussion of possible anti-unpacking techniques we refer the reader to Peter Ferrie's surveys on the topic [Ferrie 2008a; 2008b].

3.6.1 *Anti-OEP finding.* Since finding the original entry point of the packed binary is such a crucial step in the reverse-engineering process, most binary packers thoroughly obfuscate their bootstrap code, with special emphasis on hiding the control transfer to the OEP. Indirect control flow, call-stack tampering, exception-based control flow, and self-modifying code are all common techniques for hiding and obfuscating the control transfer to the OEP. The control transfer is often unpacked at run-time, and in some cases (e.g., ASProtect) the code leading up to the control transfer is polymorphic and unpacked to a different address on each execution of the same packed binary. To counter known cipher-text attacks based on teh first instructions at the program's OEP, the packer may scramble the code at the OEP so that it is unrecognizable yet functionally equivalent to the original (e.g., EXEcryptor).

Approaches. In the early days of the code-packing technique there were a few dozen packers that security companies had to be able to unpack, and they could resort to manual-labor intensive techniques to generate custom unpacker tools for specific packers [BitDefender 2007; **?**; Yason 2007]. For many packer tools, the control transfer to the OEP is the same for all packed binaries that the tool creates, and finding the control transfer to the OEP greatly facilitates the reverse-engineering of the other binaries created by that packer. In time, however, this approach became increasingly untenable. The first problem was the emergence of polymorphic packer tools that generate a different control transfer to the OEP each time they pack a binary.

Things get even more interesting for packer tools that place a polymorphic code generator in the packed binary itself, since different executions of the same binary unpack different control transfers to the OEP. Both of these polymorphic techniques make custom unpacker tools hard to generate. The real death-knell for this approach, however, came in 2008, when the percentage of packed malware binaries that used customized and private packing techniques rose to 33% of the total [Bustamante 2008; **?**].

Malware analysis tools have largely switched to employing a variety of general-purpose unpacking techniques that rely on finding the OEP of a packed program with principled heuristics. The most reliable heuristics select the OEP transfer from among the set of control transfers into memory regions that have previously been written to. This set of control transfers can be large for packed binaries that unpack in multiple stages, especially if the unpacker tool tracks written regions at a coarse granularity. For example, the Justin [Guo et al. 2008] and OmniPack unpackers [Martignoni et al. 2007] track writes at a memory-page granularity, and detect thousands of false-positive unpacking instances for some packers [Martignoni et al. 2007] that place code and writeable data on the same memory pages. Pandora's Bochs [**?**], selects among these control transfers by assuming that the last of these control transfers is the control transfer to the OEP. The Justin unpacker filters out false OEP transfers by checking that the stack pointer is the same as it was at the start of the packer's run (Yoda's Protector modifies the stack pointer), and that command-line arguments supplied to the packed program are moved to the stack prior to the OEP control transfer.

3.6.2 *Payload-code modification.* The second challenge of anti-reverse-engineering is preventing the analyst from bypassing the defensive metacode that the packer tool places in the packed binary. The metacode in obfuscated binaries created by many packer tools is easily bypassed because the metacode only serves to bootstrap the packed payload into memory and never executes again after transfering to the payload's OEP. For these packed binaries the metacode is redundant once the binary has been reverse-engineered such that the unpacked code is statically present and the original Import Address Table has been reconstructed, so the analyst can set the modified binary's entry point to the OEP and the binary will only execute payload code.

To counter this reverse-engineering technique for bypassing packer metacode, most members of the "protector" class of packer tools modify the packed payload with hooks into the metacode so that metacode executes even after the control transfer to the OEP. The easiest method of ensuring that post-OEP metacode will execute is by hooking inter-library calls with callbacks into packer metacode routines. This technique involves replacing imported function addresses in the Import Address Table with the callback addresses, as shown in Figure 6c. The callback metacode may perform timing checks or probe the binary to detect tampering prior to transfering control to the imported function corresponding to the IAT entry (e.g., Armadillo). Furthermore, the callbacks are typically located in run-time-allocated memory buffers outside of the memory that is mapped to the program binary so that a memory dump of the binary does not capture the callback functions.

The IAT-hooking technique is attractive because it requires no modification to the payload code itself, though some packer tools do use binary analysis techniques to disassemble the payload code and modify it. For example, rather than modifying IAT entries, ASProtect replaces the standard indirect `call ptr[<IAT-entry>]` instructions that reference the IAT with direct calls to its metacode callbacks, as seen in Figure 6b. To make matters worse, ASProtect and Armadillo place these callbacks in memory buffers outside of the memory image that they allocate with the VirtualAlloc Windows API call. By so doing, they ensure that reverse-engineering techniques that only capture code in the binary's memory image will miss the metacode callbacks and any stolen code bytes that they contain.

Obfuscating compilers provide yet another method of inserting post-OEP metacode (e.g., Armadillo and Themida provide plugins to the Visual Studio compiler). Obfuscating compilers can add post-OEP metacode by modifying the payload program's source code rather than having to modify its machine-language instructions in the program binary, which is much harder to do.

Approaches. We are unaware of any generally applicable techniques that automate a solution to the problem of removing payload-code modifications. Reverse-engineers manually reconstruct IAT-entries that have been replaced by pointers to metacode wrapper routines by tracing through the wrapper to discover the external function address. This technique may be possible to automate, but needs to account for timing checks and self-checksumming techniques in the metacode wrapper routines. An alternative technique is to

modify the packer bootstrap code that fills IAT entries with the addresses of metacode-wrapper routines instead of the addresses of imported functions. This technique requires that the analyst identify and patch over the wrapper-insertion code with code that fills in legitimate IAT entries. While some analysts do adopt this technique, it requires significant reverse-engineering expertise and could not easily be automated. As a side note, the stolen-bytes technique presents similar problems to those of payload-code modification in the context of binary unpacking, and reverse-engineers apply similar ad-hoc techniques to deal with that technique.

Because of these difficulties in reverse-engineering packed binaries in the general case, most unpacking tools either automate only some parts of the unpacking process [Guilfanov 2005; Kang et al. 2007], create unpacked binaries that cannot actually execute [Christodorescu et al. 2005; Yegneswaran et al. 2008], or work only on a subset of packed binaries [**?**; Coogan et al. 2009; Debray and Patel 2010].

4. OBFUSCATION STATISTICS

We now proceed to quantify the prevalence of code obfuscations to show which of the code obfuscations are most prevalent in real-world malware. To this end, we study twelve of the fifteen binary obfuscation tools that, according to a 2008 survey performed by Panda Research [Bustamante 2008], are most-often used to obfuscate real-world malware. We then applied Dyninst, our binary analysis and instrumentation tool [Hollingsworth et al. 1994; Roundy and Miller 2010], to study the obfuscated code that these tools bundle with the malware binaries that they protect from analysis.

The results of our study are presented in Table I. We sort the packers based on the market share numbers reported by Panda Research for these packers during the months of February and March 2008. We structure our discussion of these results by talking about each of the categories of obfuscation techniques in turn, and conclude with a description of each packer tool, including a discussion of the three remaining packer tools that have significant market share, but that we have not yet been able to study with Dyninst: Execryptor, Armadillo, and Themida.

Dynamic Code: All of these obfuscation tools pack the code and data bytes of the binary that they obfuscate through compression or encryption, and most of them unpack in multiple stages. We estimate the number of unpacking loops by counting control transfers to code bytes that have been modified since the previous transfer into unpacked code. When the unpacked code overwrites existing code, we categorize the event as a code overwrite rather than an instance of code unpacking. We observed overwrites that ranged from a single instruction opcode byte, to an overwrite of an 8844 byte range that consisted almost entirely of code. In the latter case, Yoda's Protector was hiding its bootstrap code from analysis by erasing it after its use. In some instances, obfuscated programs overwrite the function that is currently executing, or overwrite a function for which there is a return address on the call stack. Both cases are tricky for instrumenters, which must redirect control flow updating the PC and fixing any control transfers to dead code [Maebe and De Bosschere 2003; Roundy and Miller 2010].

Instruction Obfuscations: Indirect calls are fairly common in compiler-generated code, but most compilers use them only for function pointers, and in particular for virtual functions and inter-module calls. Indirect jumps are not particularly common, but the ret instruction is a form of indirect jump that some packers use extensively to hide obfuscated control flow. Similarly, call instructions are frequently misused, in which case they are usually paired with a pop instruction somewhere in the basic block at the call target. Another common obfuscation is the obfuscation of instruction constants, especially for constants that reveal critical information about the packing transformation, for example, the address of the program's original entry point. Some of these packers fuzz test analysis tools, either by causing parsing algorithms to parse random bytes, or by including rarely used instructions that malware emulators may not handle properly, such as floating point instructions. ASProtect's fuzz-testing is particularly thorough, especially with regards to its due to its extensive use of segment selector prefixes on instructions for which they should have no effect.

Code Layout: Code sharing actually occurs in some compiler-generated code, usually due to compiler optimizations involving in-lining or out-lining of code sequences that are used by multiple functions. This is also true of most packed programs malware, which aggressively share assembly code routines. On the other hand, compilers do not exploit the dense variable-length x86 instruction set to construct valid overlapping instruction streams as malware sometimes does. These sequences are necessarily short, and are never very

Table I. Packer statistics

		UPX	polyEnE	UPack	PECompact	PEtite	nPack	ASPack	FSG	Nspack	ASProtect	Yoda's Protector	MEW
	Malware market share *	9.5%	6.2%	2.3%	2.6%	2.5%	1.7%	1.3%	1.3%	0.9%	0.4%	0.3%	0.1%
	Bootstrap code size in kilobytes	0.4	1.1	1.0	5.8	0.0	2.6	3.3	0.2	1.1	28.0	10.2	1.5
Dynamic Code	Unpack instances	1	2	2	2	0	2	2	1	0	4	5	3
	Overwrite instances	0	0	1	2	0	0	4	0	1	205	10	0
	Overwrite byte count	0	0	64	167	0	0	179	0	194	51	9889	0
	Overwrite of executing function	0	0	1	0	0	0	0	0	0	12	4	0
Instruction Obfuscations	% of calls that are indirect	83%	75%	95%	44%	0%	28%	30%	92%	16%	7%	5%	28%
	Count of jumps that are indirect	0	0	0	0	0	0	0	1	0	69	45	0
	Non-standard uses of ret instruction	0	0	0	1	0	0	0	0	0	83	25	0
	Non-standard uses of call instruction	1	4	0	24	0	2	2	1	0	60	85	0
	Obfuscated constants	✓	✓	×	✓	×	✓	×	✓	×	✓	✓	×
	Fuzz-Test: unusual instructions	×	✓	×	✓	×	×	×	×	×	✓	✓	×
Code Layout	Functions share blocks	0	0	4	0	0	0	0	0	2	43	0	0
	Blocks share code bytes	0	10	0	0	0	0	0	0	0	6	1	0
	Interleaved blocks from different funcs	0	0	1	14	0	4	1	4	3	52	2	1
	Function entry block not first block	0	0	0	0	0	0	1	0	0	10	3	1
	Inter-section functions	✓	×	✓	✓	×	×	×	✓	✓	×	×	✓
	Inter-object functions	×	×	×	×	×	×	×	×	×	✓	×	×
	Code chunked into tiny basic blocks	×	✓	×	✓	×	×	×	×	×	✓	✓	×
	Anti-linear disassembly	×	✓	×	×	×	×	✓	×	×	✓	✓	×
	Flags used across functions	×	×	✓	×	×	×	×	✓	✓	✓	×	×
	Writeable data on code pages	✓	×	✓	✓	×	✓	✓	×	✓	✓	✓	✓
	Fuzz-Test: fallthrough into non-code	×	✓	✓	✓	×	×	×	×	×	✓	✓	×
Anti-Rewriting	Code bytes in PE header	0	0	102	0	0	0	0	158	0	0	0	182
	Code unpacked to VirtualAlloc buffer	0	0	0	1	0	0	0	0	1	3	0	0
	Polymorphism engine in packer tool	×	✓	×	×	×	×	×	×	×	✓	×	×
	Polymorphism engine in packed binary	×	×	×	×	×	×	×	×	×	✓	×	×
	Checksum of binary file	×	×	×	×	×	×	×	×	×	×	✓	×
	Checksum of malware payload	×	×	×	✓	×	×	×	×	×	✓	×	×
	Checksum of packer bootstrap code	×	×	×	✓	×	×	×	×	×	✓	✓	×
Anti-Tracing	Exception-based control transfers	0	0	0	1	0	0	0	0	0	1	8	0
	Metacode uses WinAPI funcs not in IAT	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×
	Timing check	×	×	×	×	×	×	×	×	×	×	✓	×
	Anti-Debug: via WinAPI calls	×	×	×	×	×	×	×	×	×	✓	✓	×

* *As determined by PandaLabs in 2008 study:* http://pandaresearch.wordpress.com/2008/03/19/packer-revolution/

useful, but are used to break the assumptions that are made by many analysis tools [Kruegel et al. 2004; Nanda et al. 2006]. Blocks from different functions are often interleaved as a result of function sharing, but some packers such as ASProtect aggressively interleave function blocks such that the blocks may be spread all across a code section. These strange function layouts often result in function entry blocks that are at higher addresses than other function blocks, breaking a common assumption. To make matters worse, function blocks are frequently spread across code sections and even across code objects, in which case the code is often in memory buffer created by a VirtualAlloc Windows API call. In portions of several obfuscated binaries, basic block lengths are reduced by chunking the blocks into groups of two or three instructions, ending with a jump instruction to the next tiny code chunk. The purpose of this code chunking is sometimes to thwart the linear-sweep disassemblers that are used by many interactive debuggers (e.g., OllyDbg) and disassembly tools (e.g., Objdump); padding bytes in-between the chunked basic blocks confuse the linear-sweep disassembler, hiding the actual instructions from analysis. Another interesting characteristic of obfuscated code is that the contracts between calling functions and called functions are completely nonstandard. A good example of this is the frequence with which a called function sets a status flag that is read by the caller, which is something that x86 compilers never do, even in highly optimized code. Similarly, compilers practically never place writeable data on code pages, since this has dangerous security implications, whereas obfuscated programs do this extensively. Finally, several obfuscators fuzz-test recursive-traversal parsers by causing them to parse into non-code bytes. They do this either by including junk bytes on one edge of a conditional branch that is never taken [Collberg et al. 1998], or more often, by causing control flow to fall through into bytes that will be decrypted in place at runtime.

Anti-Rewriting: Analysts frequently try to rewrite packed binaries to create versions that are fully unpacked and bypass the obfuscated bootstrap code. One way in which packed binaries make this task challenging is by placing code in strange places, such as the Portable Executable file header and in buffers that are created by calls to the VirtualAlloc Windows API function. The UPack, FSG, and MEW packers put code in the PE header where it will be overwritten by the binary rewriter if it adds or renames program sections. The code in VirtualAlloc buffers can easily be missed by the rewriting process, and since ASProtect places payload code in these buffers using the stolen bytes technique, the program will crash if the code in these buffers is not preserved by the rewriting process. Polymorphism is also a problem for binary rewriting. Polymorphic packer tools produce different packed binaries each time they pack the same payload executable. For example, PolyEnE changes the cipher it uses to encrypt the payload each time it unpacks a program, and modifies its own bootstrap code as well. ASProtect and other packer tools (e.g., Themida) include a polymorphic unpacker in the packed binaries themselves, so that the same packed binary unpacks different bootstrap code and adds different obfuscations to the program's payload code. Finally, some packers use self-checksumming to detect modifications to their code. Checksums are most-often calculated over the packed program's bootstrap code, though they are sometimes also calculated over the unpacked payload code and over the packed binary file itself.

Anti-Tracing: Tracing techniques are commonly used to study the behavior of obfuscated programs, and may be collected at different granularities, ranging from instruction-level traces to traces of Windows API calls, and system calls. Some packers use exception-based control transfers, which obfuscate the program's control flow and may cause errors in emulators, which may not handle exceptions properly. Since analysts frequently request traces at the level of Windows API calls [Yegneswaran et al. 2008], obfuscated binaries frequently circumvent the Import Address Table data structure by which the Windows loader links the code to imported library functions, and which many tracing tools use to decide which Windows API functions to instrument [Hunt and Brubacher 1999]. Timing checks are used by some obfuscated binaries to detect significant slowdowns that are indicative of single-step tracing or interactive debugging. Finally, some packed programs invoke Windows API calls that reveal the presence of a debugger process that is attached to the running process.

Packer Personalities. :

We now describe the characteristics of each packer tool, and the features that are unique to that tool. We organize the list of packers by increasing size of their bootstrap code, but as seen in line 3 of Table I, bootstrap size is not necessarily indicative of the size of the binaries that the packers produce, nor is it always indicative of the prevalence of obfuscation techniques in the packer.

FSG (158 bytes): The Fast, Small, Good EXE packer's goals are stated in its name. Its bootstrap code is fast and small (its 158 bytes of bootstrap code make it the smallest packer in our list), and it offers fairly good compression ratios considering its tiny size. FSG achieves a significant degree of obfuscation in its pursuit of compactness, using unusual instructions and idioms that make its bootstrap code much harder to analyze than its 158 bytes would seem to indicate. FSG's bootstrap code is sufficiently small to fit comfortably in the one-page section that contains the Windows Portable Executable (PE) header.

UPX (392 bytes): The Ultimate Packer for eXecutables is notable for offering the broadest coverage of hardware platforms and operating systems, and is one of the few packer tools that are open sourced. UPX is the most popular packer tool with malware authors despite its lack of any obfuscation beyond what is afforded by the code-packing transformation itself. Among the contributing factors for UPX's popularity are its early arrival on the packing scene and the ease with which a custom packer can be derived from its open-source code.

UPack (629 bytes): The UPack packer consistently generates the smallest packed binaries while including several obfuscation techniques. The UPack authors scavenge bytes from the PE header with unparalleled effectiveness, stuffing it with code, data, function pointers, and the import table. Its PE header tricks cause robust binary analysis tools such as IdaPro and OllyDbg to misparse some of the PE's data structures, while its code overwriting and two-phase unpacking cause problems for other analysis tools.

PolyEnE (897 bytes): PolyEnE is widely used by malware for its ability to create polymorphic binaries: each time a binary is packed it uses a different cipher on the encrypted payload bytes, resulting in changes to the bootstrap code itself, and to the encrypted bytes. The address of the bootstrap code's Import Table is also varied, is the memory size of the bootstrap code's section, though the packed binary itself always has the same size.

MEW (1593 bytes): MEW employs several tricks that save space while causing problems for analysis tools: it employs two unpacking stages rather than one, wedges code into unused bytes in the PE Header, shares code between functions, and mixes code and writeable data on the same memory pages.

PECompact (1943 bytes): PECompact provides perhaps the broadest array of defensive techniques for a packer that offers good compression. Notable defensive techniques include its early use of an exception to obfuscate control flow, as well as its optional incorporation of self-checksumming techniques to detect code patching.

NSPack (2357 bytes): NSPack also contains a broad collection of defensive techniques. Two notable examples: it resists binary rewriting by unpacking part of its bootstrap into a memory buffer that is not backed by a file and it only unpacks if its symbol table is empty.

nPack (2719 bytes): The bootstrap code of nPack-generated binaries appears to have been mostly compiler-generated, without much emphasis on compactness. Another reason for its large bootstrap code are its single unpacking phase and its support for packing binaries with thread-local storage. There is little to no code obfuscation in this packer's bootstrap code.

ASPack (3095 bytes): Among the packers whose primary goal is to provide binary compression, ASPack does the most thorough job of frustrating analysis by obfuscating control flow in its bootstrap code. Its obfuscation techniques include call-stack tampering, use of `call` and `ret` instructions for non-standard purposes, and code overwriting to modify a control flow target.

Yoda's Protector (9429 bytes): This is the smallest of the "protector" class of packing tools, most of which are written as for-profit tools designed to present intellectual property from reverse engineering. Despite its moderate size, Yoda's Protector employs an impressive array of control-flow obfuscation and anti-debugging techniques, several of which are unique to this packer.

ASProtect (28856 bytes): ASProtect's large bootstrap code shares some code with ASPack, its sister tool, and its main features are likewise directed towards control-flow and anti-disassembler obfuscations. ASProtect-packed binaries carry a polymorphic code-generation engine which adds considerably to their size and to the difficulty of automatically reverse-engineering binaries that have been packed by ASProtect. Reverse-engineering is also made difficult by ASProtect's modification of call instructions that reference the IAT and because of its use of stolen-bytes techniques.

REFERENCES

16 1998. Darkparanoid virus.

Anckaert, B., Madou, M., and De Bosschere, K. 2007. A model for self-modifying code. In *Workshop on Information Hiding*. Alexandria, VA, 232–248.

Babic, D., Martignoni, L., McCamant, S., and Song, D. 2011. Statically-Directed Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, Canada.

Balakrishnan, G. and Reps, T. 2004. Analyzing memory accesses in x86 executables. In *Conference on Compiler Construction (CC)*. New York, NY, 5–23.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*. Bolton Landing, NY.

Bayer, U., Moser, A., Kruegel, C., and Kirda, E. 2006. Dynamic analysis of malicious code. *Journal in Computer Virology 2,* 1 (August), 66–77.

Bellard, F. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*. Anaheim, CA.

Bernat, A. R. and Miller, B. P. 2011. Anywhere, Any Time Binary Instrumentation. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Szeged, Hungary.

Bernat, A. R., Roundy, K. A., and Miller, B. P. 2011. Efficient, Sensitivity Resistant Binary Instrumentation. In *International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, Canada.

BitDefender. 2007. BitDefender anti-virus technology. White Paper.

Bruening, D. 2004. Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Bustamante, P. 2008. Packer (r)evolution. Panda Research web article.

Charney, M. 2010. Xed2 user guide. `http://www.cs.virginia.edu/kim/publicity/pin/docs/36111/Xed/html/main.html`.

Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., and Veith, H. 2005. Malware normalization. Tech. Rep. 1539, Computer Sciences Department, University of Wisconsin. November.

Cifuentes, C. and Fraboulet, A. 1997. Intraprocedural static slicing of binary executables. In *International Conference on Software Maintenance (ICSM)*. Los Alamitos, CA.

Cifuentes, C. and Van Emmerik, M. 1999. Recovery of jump table case statements from binary code. In *International Workshop on Program Comprehension (ICPC)*. Pittsburgh, PA.

Collberg, C., Thomborson, C., and Low, D. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Symposium on Principles of Programming Languages (POPL)*. San Diego, CA.

Coogan, K., Debray, S., Kaochar, T., and Townsend, G. 2009. Automatic static unpacking of malware binaries. In *Working Conference on Reverse Engineering*. Antwerp, Belgium.

Debray, S. and Evans, W. 2002. Profile-guided code compression. In *Conference on Programming Language Design and Implementation (PLDI)*. Berlin, Germany.

Debray, S. and Patel, J. 2010. Reverse engineering self-modifying code: Unpacker extraction. In *Working Conference on Reverse Engineering*. Boston, MA.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. 2008. Ether: Malware analysis via hardware virtualization extensions. In *Conference on Computer and Communications Security*. Alexandria, VA.

Falliere, N. 2007. Windows anti-debug reference. Infocus web article.

Ferrie, P. 2008a. Anti-unpacker tricks. In *International CARO Workshop*. Amsterdam, The Netherlands.

Ferrie, P. 2008b. Anti-unpacker tricks - part one. *Virus Bulletin*.

Fog, A. 2011. Calling conventions for different c++ compilers and operating systems. `http://www.agner.org/optimize/`.

Giffin, J. T., Christodorescu, M., and Kruger, L. 2005. Strengthening software self-checksumming via self-modifying code. In *Annual Computer Security Applications Conference (ACSAC)*. Tucson, AZ.

GNU Project - Free Software Foundation. 2011. objdump, gnu manuals online. Version 2.22 `http://sourceware.org/binutils/docs/binutils/`.

Graf, T. 2005. Generic unpacking - how to handle modified or unknown PE compression engines? In *Virus Bulletin Conference*. Dublin, Ireland.

Guilfanov, I. 2005. Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables. Online tutorial. `http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf`.

Guilfanov, I. 2011. The IDA Pro disassembler and debugger. DataRescue. Version 6.2 `http://www.hex-rays.com/idapro/`.

Guo, F., Ferrie, P., and Chiueh, T. 2008. A study of the packer problem and its solutions. In *Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer Berlin / Heidelberg, Cambridge, MA.

Hollingsworth, J. K., Miller, B. P., and Cargille, J. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*. Knoxville, TN.

Hunt, G. and Brubacher, D. 1999. Detours: Binary interception of win32 functions. In *USENIX Windows NT Symposium*. Seattle, WA.

Jacobson, E. R., Rosenblum, N. E., and Miller, B. P. 2011. Labeling library functions in stripped binaries. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Szeged, Hungary.

Kang, M. G., Poosankam, P., and Yin, H. 2007. Renovo: A hidden code extractor for packed executables. In *Workshop on Recurring Malcode*. Alexandria, VA.

Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. 2004. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*. San Diego, CA.

Lakhotia, A., Kumar, E. U., and Venable, M. 2005. A method for detecting obfuscated calls in malicious binaries. *Transactions on Software Engineering 31,* 11 (November).

Lindholm, T. and Yellin, F. 1999. *Java Virtual Machine Specification*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Conference on Computer and Communications Security*. Washington, D.C.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL.

Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *ACM workshop on Digital Rights Management*. Alexandria, VA.

Maebe, J. and De Bosschere, K. 2003. Instrumenting self-modifying code. In *Workshop on Automated and Algorithmic Debugging*. Ghent, Belgium.

Maebe, J., Ronsse, M., and De Bosschere, K. 2002. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Workshop on Binary Translation held in conjunction with the Conference on Parallel Architectures and Compilation Techniques (PACT)*. Charlottesville, VA.

Martignoni, L., Christodorescu, M., and Jha, S. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)*. Miami Beach, FL.

Miller, B. P., Fredriksen, L., and So, B. 1990. An empirical study of the reliability of unix utilities. *Communications of the ACM 33,* 12 (December).

Moser, A., Kruegel, C., and Kirda, E. 2007. Exploring multiple execution paths for malware analysis. In *Symposium on Security and Privacy*. Oakland, CA.

Muth, R. and Debray, S. 2000. On the complexity of flow-sensitive dataflow analyses. In *Symposium on Principles of Programming Languages (POPL)*. Boston, MA.

Nanda, S., Li, W., Lam, L.-C., and cker Chiueh, T. 2006. Bird: Binary interpretation using runtime disassembly. In *Symposium on Code Generation and Optimization (CGO)*. New York, NY.

Paradyn Tools Project. 2011. ParseAPI programmer's guide. Version 7.0.1 `http://www.paradyn.org/html/manuals.html`.

Perriot, F. and Ferrie, P. 2004. Principles and practise of x-raying. In *Virus Bulletin Conference*. Chicago, IL.

Popov, I., Debray, S., and Andrews, G. 2007. Binary obfuscation using signals. In *USENIX Security Symposium*. Boston, MA.

Prakash, C. 2007. Design of x86 emulator for generic unpacking. In *Assocation of Anti-Virus Asia Researchers International Conference*. Seoul, South Korea.

Quinlan, D. 2000. Rose: Compiler support for object-oriented frameworks. In *Conference on Parallel Compilers (CPC2000)*. Aussois, France.

Quist, D. and Valsmith. 2007. Covert debugging: Cirvumventing software armoring techniques. In *Blackhat USA*. Las Vegas, NV.

Rosenblum, N. E., Cooksey, G., and Miller, B. P. 2008. Virtual machine-provided context sensitive page mappings. In *Conference on Virtual Execution Environments (VEE)*. Seattle, WA.

Rosenblum, N. E., Miller, B. P., and Zhu, X. 2010. Extracting compiler provenance from program binaries. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Toronto, Canada.

Rosenblum, N. E., Zhu, X., Miller, B. P., and Hunt, K. 2008. Learning to analyze binary computer code. In *Conference on Artificial Intelligence (AAAI)*. Chicago, IL.

Roundy, K. A. and Miller, B. P. 2010. Hybrid analysis and control of malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)*. Ottawa, Canada.

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

Schwarz, B., Debray, S., and Andrews, G. 2002. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*. Richmond, VA.

Sharif, M., Lanzi, A., Giffin, J., and Lee, W. 2008. Impeding malware analysis using conditional code obfuscation. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. 1993. Binary translation. *Communications of the ACM 36,* 2 (February).

Stepan, A. E. 2005. Defeating polymorphism: beyond emulation. In *Virus Bulletin Conference*. Dublin, Ireland.

Stewart, J. 2007. Unpacking with ollybone. Online tutorial. `http://www.joestewart.org/ollybone/tutorial.html`.

Theiling, H. 2000. Extracting safe and precise control flow from binaries. In *Conference on Real-Time Computing Systems and Applications*. Cheju Island, South Korea.

Trilling, S. 2008. Project green bay–calling a blitz on packers. *CIO Digest: Strategies and Analysis from Symantec*.

Vigna, G. 2007. Static disassembly and code analysis. In *Malware Detection*. Advances in Information Security, vol. 35. Springer.

Willems, C., Holz, T., and Freiling, F. 2007. Toward automated dynamic malware analysis using CWSandbox. In *Symposium on Security and Privacy*. Oakland, CA.

Wurster, G., Van Oorschot, P. C., and Somayaji, A. 2005. A generic attack on checksumming-based software tamper resistance. In *Symposium on Security and Privacy*. Oakland, CA.

Yason, M. V. 2007. The art of unpacking. In *Blackhat USA*. Las Vegas, NV.

Yegneswaran, V., Saidi, H., and Porras, P. 2008. Eureka: A framework for enabling static analysis on malware. Tech. Rep. SRI-CSL-08-01, SRI International. April.

Yuschuk, O. 2000. OllyDbg. Version 1.10 `http://www.ollydbg.de`.