# Binary-Code Obfuscations in Prevalent Packer Tools

KEVIN A. ROUNDY, University of Wisconsin and Symantec Research Labs
BARTON P. MILLER, University of Wisconsin

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. While some obfuscations have been reported independently, this survey consolidates the discussion while adding substantial depth and breadth to it. This survey also quantifies the relative prevalence of these obfuscations by using the Dyninst binary analysis and instrumentation tool that was recently extended for defensive malware analysis. The goal of this survey is to encourage analysts to focus on resolving the obfuscations that are most prevalent in real-world malware.

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

**ACM Reference Format:**
Roundy, K. A. and Miller, B. P. 2013. Binary-code obfuscations in prevalent packer tools. ACM Comput. Surv. 46, 1, Article 4 (October 2013), 32 pages.
DOI: http://dx.doi.org/10.1145/2522968.2522972



## 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 broad examination of the obfuscation techniques used by the packer tools that are most popular with malware authors [Bustamante 2008b]. Though previous studies have discussed the current state of binary packing [Yason 2007], antidebugging [Falliere 2007], and antiunpacking [Ferrie 2008a] techniques, there have been no comprehensive studies of the obfuscation techniques that are applied to binary code. Our snapshot of current obfuscation techniques captures the obfuscations that we have seen to date, and will need to be periodically refreshed as obfuscation techniques continue to evolve. While some of the individual obfuscations that we discuss have been reported independently, this article 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

Authors' addresses: K. A. Roundy (corresponding author), Department of Computer Science, University of Wisconsin and Symantec Research Labs, 900 Corporate Pointe, Culver City, CA 90230; email: roundy@cs.wisc.edu; B. P. Miller, Department of Computer Science, University of Wisconsin.

DOI: http://dx.doi.org/10.1145/2522968.2522972

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. 2008b] 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 then present a taxonomy of the obfuscation techniques, along with current approaches to dealing with these techniques (Section 3). As they read this section, readers may wish to reference Section 4, where we provide a summary table of the obfuscation techniques that shows their relative prevalence in real-world malware. Section 4 also provides brief descriptions of the packer tools that are most often used by malware authors. We conclude in Section 5.

## 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 2008b] and applying them to program binaries. The packer tools transformed the binaries by applying obfuscations to their code and compressing or encrypting their code and data bytes. At runtime the packed binaries unroll the code and data bytes of their payload into the program's address space and then execute the payload code. We carefully analyzed the obfuscated *metacode* that packer tools incorporate into such binaries; packer metacode is software that consists of the obfuscated bootstrap code that unrolls the original binary payload into memory, and the modifications that the packer tool makes to the payload code, but does not include the payload code.

We obtained most of our observations about these obfuscated binaries by adapting the Dyninst binary code analysis and instrumentation tool to analyze highly defensive program binaries, and by then using it for that purpose. 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. Prior to our adaptations, Dyninst discovered and analyzed code through a one-time static analysis, and monitored and modified code by using patch-based instrumentation techniques. We extended Dyninst in two phases, creating the SD-Dyninst and SR-Dyninst prototypes. Where relevant, Section 3 provides details of how our tools deal with the obfuscations presented in this article, together with a discussion of antiobfuscation techniques that have been developed by other researchers. Here we present a high-level overview of the techniques used by our prototypes and refer readers to our SD-Dyninst [Roundy and Miller 2010] and SR-Dyninst [Bernat et al. 2011] publications for additional details.

*SD-Dyninst.* Our first step towards getting Dyninst to work on malware was to adapt its static analysis and add dynamic analysis techniques [Roundy and Miller 2010]. We named our prototype implementation of this technique SD-Dyninst, after its hybrid combination of static and dynamic analysis techniques. This work required modifying Dyninst's static code identification to eliminate dependencies on binary code conventions that are followed by compilers, but not by malware authors (see Sections 3.1

and 3.2). By eliminating these assumptions, our static analysis improves its ability to accurately identify obfuscated code, but it does so with diminished coverage. We compensate for this diminished coverage through a combination of static and dynamic analysis techniques. We statically resolve many forms of control-flow obfuscations by using Dyninst's binary slicing capabilities to identify obfuscated control transfer targets. We use dynamic techniques to resolve additional control-flow obfuscations and to analyze code that is not statically present, due to code packing and code overwriting (see Section 3.1). These dynamic analysis techniques leverage Dyninst's instrumentation capabilities. We instrument all obfuscated control transfers that could transition from code we have analyzed to code that is hidden from static analysis. These instrumentation-based techniques, together with our dynamic techniques for identifying code overwrites, are described in Section 3, and at greater length in the paper that introduced SD-Dyninst [Roundy and Miller 2010].

*SR-Dyninst.* After building SD-Dyninst we adapted Dyninst's patch-based instrumentation techniques so that we could instrument malware programs more stealthily. Our primary motivation was to transparently instrument binaries that explicitly attempt to detect code patching and instrumentation by applying self-checksumming and other related anti-patching techniques (see Section 3.5). We solved this problem in a general way, leveraging the observation that most programs that are sensitive to patch-based instrumentation read from the program's code as if it were data. Thus, SR-Dyninst hides code patches from the instrumented program by redirecting the program's attempts to read from patched code to a region of shadow memory that preserves the unpatched data, as described at length in previous publications on SR-Dyninst [Bernat et al. 2011; Roundy 2012]. This strategy has the additional benefit of compensating for instances in which Dyninst's static analysis mistakenly parses data bytes as if they were code, which could lead Dyninst patch over those data bytes. Our strategy compensates for this scenario by causing the instrumented program to read the original data bytes rather than the patched data bytes.

An important limitation of our SD-Dyninst prototype was that it lacked SR-Dyninst's aforementioned protection against errors in its static analysis. This meant that nearly every time an obfuscation caused SD-Dyninst to parse non-code bytes, our instrumentation would patch over those bytes, causing the instrumented program to fail to execute in the same way as the original binary. Unfortunately for us at the time, there are many obfuscations that can introduce these kinds of parsing errors (see Section 3.2), which forced us to perform a careful case-by-case study of each instance of these obfuscations, 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. Ultimately, this labor was not wasted, as it allowed us to discover and study some of the obfuscations that we describe in this article, and it forced us to carefully evaluate and debug our parsing code to eliminate mistaken identifications of data bytes as code.

In this article, we use SR-Dyninst to automatically generate statistical reports of the defensive techniques employed by these packer tools and we present those results in this study. The statistics are based on the disassembly, Control-Flow Graphs (CFGs), data-flow analyses, and instrumentation output generated by our obfuscation-resistant version of Dyninst. We also report on results gained from manual observations of these programs gleaned from methodical perusals of their disassembled code and data.

## 3. THE OBFUSCATION TECHNIQUES

We structure this discussion around foundational binary analysis tasks. For each task, we describe solutions to these tasks, present defensive techniques that malware

authors use to complicate the tasks, and survey any countermeasures 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.

### 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 nondefensive binaries that do not generate code at runtime, 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 nondefensive binaries can also be extracted from the program's memory image at any point during its execution, as the code does not change at runtime 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 runtime. 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 techniques for analyzing dynamic code after our description of code packing and code overwriting.

*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 runtime. 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.

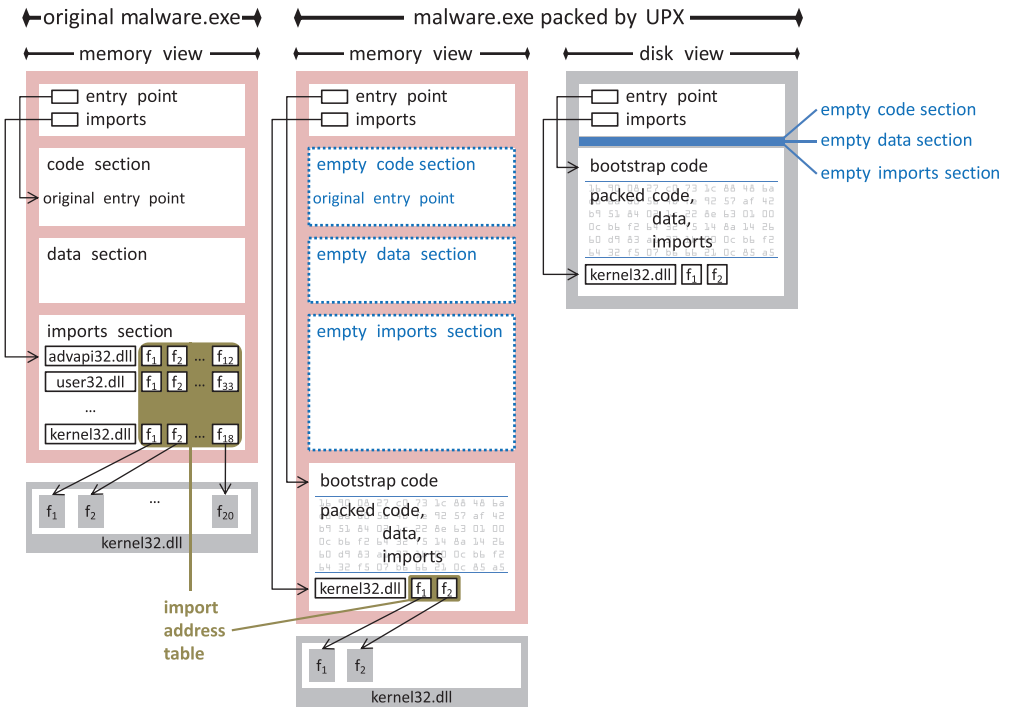


Fig. 1. Abstract view of a representative packing transformation as performed by versions of the UPX packer up through Version 3.08. UPX compresses malware.exe's code and data, setting the packed binary's entry point to its bootstrap code, which will unpack the code and data into memory at runtime. UPX 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 runtime.

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% 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 yet 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 runtime. 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 antivirus 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 runtime. Dyninst differs from Bird by capturing dynamically unpacked code and reapplying 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 runtime. 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 runtime to push the original entry point 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 article only overwrite their own metacode. More complex code overwriting scenarios are possible, for example, the MoleBox packer tool and DarkParanoid virus [Dark Paranoid 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. 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 these 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 a 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. 2008b].

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

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

(a)

| Misused Call |
|---|
| `call <target>` |
| `.target` |
| `pop <register-name>` |

(b)

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

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 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 2(a). 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 2(b)). This easily implemented obfuscation attacks analysis tools 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 nonstandard 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. [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

```
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.

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 runtime. 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 nonstandard uses of the `call` instruction, call-stack tampering adds nonstandard uses of the `ret` instruction as well. Figure 3 illustrates three call-stack tampering tricks used by the ASProtect packer. Figure 3(a) shows an obfuscated `push-ret` instruction sequence that is used as an equivalent to a `jmp ADDR` instruction. Figure 3(b) is a somewhat more complex instruction sequence that is also a `jmp ADDR` equivalent. Figure 3(c) 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.

*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 runtime.

*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 runtime. 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 unexecuted and usually will not discover all targets of multiway 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 runtime 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 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 reregistering 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, prevents 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 runtime. 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 runtime 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 runtime. 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 arranged 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 runtime (e.g., for dynamic instrumentation), does not generalize to junk code detection at opaque branch targets, and does not capture overwritten code. Not capturing overwritten code is significant because many packers overwrite or deallocate code buffers immediately after use.

Code-parsing techniques can cope with ambiguity by leveraging the fact that, though random bytes usually disassemble into valid $\times 86$ 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 misparse 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 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 misidentifying 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 tools are available (e.g., Rose [Quinlan 2000], Qemu [Bellard 2005], TSL [Lim and Reps 2008]).

### 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 A makes a tail call to function B, B's `ret` instructions will bypass A, transferring control to the return address of the function that called A. A naïve code parser confronted with this optimization would confuse A's call to B 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, namely 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 among the code sequences that we have seen tampering with the call stack, most of these sequences are elaborations on the basic examples that we presented in Figure 3.

*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 lookahead 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 and 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.

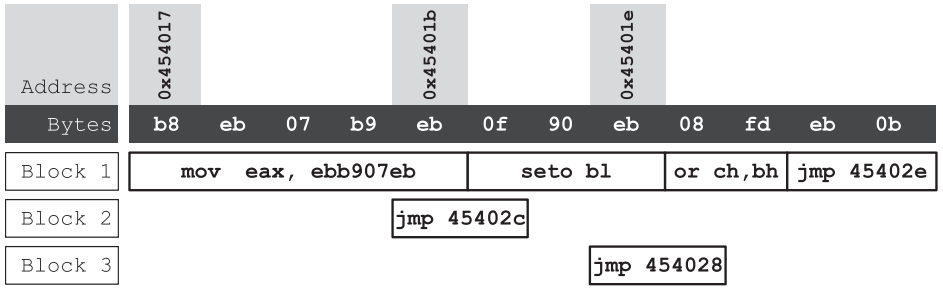


Fig. 4. An example of overlapping instructions and basic blocks taken from the obfuscated bootstrap code of a binary packed by Armadillo. All three basic blocks actually execute.

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 those 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 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 runtime, 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 zero or one 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 browsing through 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 view the program's disassembly and interact with the program's execution, either by single-stepping the program's execution, patching in breakpoints, modifying sequences of code or data, 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. In previous sections, we presented several obfuscations that impact code comprehension, but focused on their other anti-analysis effects; we proceed by summarizing their effects on code comprehension before moving on to new techniques. First, packed and self-modifying binaries contain dynamic code that is difficult for analysts to think about, as most programs do not change over time. Furthermore, since the dynamic code in these binaries is not amenable to static analysis, structural analyses of their code and data may not be available. Second, obfuscations that target disassembler techniques introduce gaps and errors in the disassembly, frequently forcing the analyst to correct these errors through tedious interactions with a disassembler tool. Furthermore, fuzz-testing techniques that confuse weak disassemblers are even more likely to confuse analysts that have to make sense of the unusual instructions. Finally, unusual function layouts and obfuscated function boundaries make it hard to identify functions and their relationships to one another, which is bad for the analyst, as code is easiest to understand when it is structured the way it is written, namely in functions with well-defined interactions.

The remainder of this section presents additional obfuscations that impact code comprehension: machine-language instructions whose constant operands are obfuscated, violations of calling conventions, and do-nothing code in obfuscated programs, and metacode that is decoded by an embedded virtual machine.

*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 constants extensively. The constant that packers most frequently obfuscate is the address of the original entry point of the payload binary. Obfuscated uses of the OEP address make it harder for analysts to reverse-engineer packer bootstrap code 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 try to make sense of the program's instructions. Fortunately, decompiler tools resolve many constant obfuscations while aiding code comprehension generally [Van Emmerik and Waddington 2004; Hex-Rays Decompiler 2011]. In the process of creating programming-language statements from machine-language insructions, decompilers translate the instructions into an intermediate representation on which they apply basic arithmetic rules (a restricted form of symbolic evaluation) to reduce complex operations into simpler forms [Bruschi et al. 2007]. This approach produces programming-language statements from which constant

| Not obfuscated | Obfuscated |
|---|---|
| `mov ecx, 0x294a` | `mov ecx, 0x410c4b`<br>`sub ecx, 0x40e301` |

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

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

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 and constant operand with a sequence of instructions that obfuscate the constant, thereby slowing down the analyst.

obfuscations that do not involve memory accesses are eliminated. Unfortunately, current decompilers do not work on code that is dynamically unpacked at runtime, so the analyst must first reverse-engineer the program such that dynamically unpacked code is statically present in the rewritten binary. For programs that do not employ the antiunpacking techniques that we will discuss in Section 3.6, there are general-purpose unpacking tools that may be able to automate the reverse-engineering task [Böhne and Holz 2008; Yegneswaran et al. 2008]. However, for binaries that do employ antiunpacking, the analyst must reverse-engineer the binary by hand and must therefore manually deobfuscate uses of the OEP address. In these situations, analysts deobfuscate constants using techniques that are more manual-labor intensive, such as using a calculator program to do arithmetic or an interactive debugger to force the obfuscated instructions to execute.

*3.4.2. Calling-Convention Violations.* Calling conventions standardize the contract between caller functions and their callees by specifying such things as 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 a called function. Though there are many calling conventions for the $\times 86$ platform [Fog 2011], program binaries usually adhere to a single set of conventions and analysts can quickly adapt to new conventions. This is not as true of aggressively optimized binaries, which ignore many standardized conventions and can be nearly as hard to analyze as deliberately obfuscated code.

There is a great deal of stylistic variety in the metacode used by packer tools to bootstrap payload code into memory and obfuscate the binaries that they produce. Thus, while packed programs may contain some compiler-generated metacode that adheres to standard calling conventions, nearly all of them contain code that does not. For instance, metacode functions frequently operate directly on register values rather than adhering to convention-defined locations for parameter lists and return values. Though this is also true of some optimized code, packer metacode takes things a step further by sometimes making branching decisions based on status-register flags set by comparisons made in a called function. In this instance, the called function is effectively storing a return value in the $\times 86$ status register; we do not believe there to be any standard calling conventions or compiler optimizations that do this.

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 certain register contents to be unmodified across function-call boundaries. Similarly, for the sake of efficiency, binary instrumentation tools may modify a function in a way that flips status-register flags based on the assumption that those flags will not be read by callers to that function. This assumption is safe for compiler-generated code, but not for packer

metacode, where instrumenting based on this assumption may unintenionally alter the program's behavior.

*Approaches.* Though assumptions based on calling conventions allow tools like disassemblers and instrumenters to make significant improvements in such metrics as code coverage [Harris and Miller 2005; Kruegel et al. 2004] and instrumentation efficiency [Hollingsworth et al. 1994], these assumptions limit their tools' applicability to obfuscated code. 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 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]). To maintain good instrumentation efficiency, Dyninst now assumes registers to be dead only if they are proved to be so by a binary slice [Bernat et al. 2011]).

*3.4.3. Do-Nothing Code.* Most obfuscation techniques protect sensitive code from analysis and reverse-engineering by making the code hard to access or understand. Do-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. 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 and do-little code are most useful in packers with small bootstrap code sizes, as the larger "protector" tools already contain so much code that is solely 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. PolyEnE and most other packers that employ this strategy usually make the inclusion of do-nothing code optional, since many users of packer tools wish their packed binaries to be small.

*Approaches.* Christodorescu et al. developed techniques to detect semantic-nops in malware programs and remove them by rewriting the binary into a "normalized" form [Christodorescu et al. 2005]. Their techniques also normalize control-flow graphs that have been chunked into tiny basic blocks and they account for a single layer of code packing, but do not account for common defensive techniques such as code overwriting, multilayer packing, and anti-patching (see Section 3.5).

Do-nothing code that never executes can sometimes be eliminated by the malware normalization techniques that Bruschi et al. built into the Boomerang decompiler [Bruschi et al. 2007]. They operate on Boomerang's intermediate representation of the code, to which they apply arithmetic rules that reduce the conditional predicates of some branch instructions to "true" or "false". Having identified branches that always take one path, they eliminate the other path from the program. Unfortunately, decompilers do not work on packed code, so this technique requires that packed binaries first be reverse-engineered into an unpacked state, as described in Section 3.6.

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 [Russinovich and Cogswell 1997]) and then using this summary 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.

*3.4.4. Embedded Virtual Machines.* Some obfuscation tools protect sensitive code by creating binaries that contain an embedded virtual machine that can emulate an obfuscated bytecode language, and translating the sensitive code into bytecode. Thus, when

the obfuscated binary executes and control flow passes to the program's sensitive code, the binary invokes its virtual machine to translate this bytecode into native instructions. This layer of indirection makes it difficult to understand the behavior of the obfuscated bytecode, as the virtual machine treats the bytecode as data rather than code. Though a disassembler may successfully analyze the native machine-language instructions of the embedded virtual machine, it will not disassemble the obfuscated bytecode that drives the virtual machine's behavior. For this reason, providing an analysis of obfuscated bytecode is considered out of scope for many analysis tools, including Dyninst. We report briefly on this obfuscation even though we did not study it directly.

To date, the only obfuscation tools we are aware of that have implemented this technique are commercial packing tools such as Code Virtualizer, VMProtect, and Themida. Each time these tools create an obfuscated binary, they construct a unique bytecode and embed a polymorphic virtual machine designed to emulate that particular bytecode. An important feature of these virtual machines is that they emulate a single bytecode instruction at a time, maintaining a virtual program counter that always points to the next bytecode instruction. These virtual machines contain a main loop that fetches the next instruction, decodes it, and dispatches a code routine that emulates the bytecode instruction.

*Approaches.* Despite extensive attempts at obfuscation, the virtual machines employed by current obfuscation tools share characteristics that allow them to be reverse-engineered. Rotalumé is a reverse-engineering tool that is able to find a binary's bytecode, extract its syntax and semantics, and build a control-flow graph for the obfuscated bytecode [Sharif et al. 2009]. Rotalumé works by identifying the fetch, decode, and dispatch phases that are used by the virtual machines in malware emulators. The tool begins by tracing the program's execution and applying data-flow analysis to identify the virtual machine's virtual-program-counter variable. Once they have the virtual PC, they detect the VM's fetch routine by tainting the virtual PC variable to identify uses of the virtual PC to read the opcode of the next instruction. Identifying the fetch routine in this way is important because it allows Rotalumé to identify all opcodes used by the obfuscated bytecode language. Next, they decipher the meaning of those opcodes by identifying the VM's dispatch code, which uses the opcode to invoke the correct code routine that implements that instruction. Dispatch-code identification works by tainting the opcodes obtained in the fetch phase to identify control transfers whose target is determined by opcode values. Having identified the program's dispatch code, Rotalumé then associates each opcode instruction with the code routine that implements it. The code routines that implement the bytecode's jump instructions stand out because they update the virtual PC, and by reverse-engineering these instructions, Rotalumé is able to reconstruct the obfuscated bytecode's control-flow graph. Their approach is general to any embedded virtual machines that emulate a single bytecode instruction at a time with a fetch-decode-dispatch loop.

### 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 often 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]. Fine-grained studies of the program's execution are no less common, and frequently involve the use of code patching to implement software breakpoints in interactive debugger programs like Ida Pro.

On the ×86 platform, the software-breakpoint mechanism works by overwriting the instruction at the breakpoint address with a single-byte `int3` breakpoint instruction. When the `int3` 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 exception handler) then performs 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 the code with jump instructions instead of `int3` 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 and cannot always be used, as ×86 long-jump instructions are 5 bytes long and may overwrite multiple original instructions and basic blocks.

Code-patching techniques can be applied statically, to the binary file, or dynamically, to the program binary's image in memory. Statically applying code patches to malware binaries is difficult because they are usually packed, meaning that the only code in the binary file that is not compressed or encrypted is the packer bootstrap code. Though dynamic patch-based techniques are a better fit for most malware, even dynamic techniques are not readily applied to programs that resist code patching with the stolen-bytes and self-checksumming techniques that we describe in this section.

*3.5.1. Stolen Bytes.* Figure 6(b) illustrates the stolen-bytes technique pioneered by the ASProtect packer, which circumvents patch-based tracing of shared library functions. Patch-based tracing of binary functions typically involves replacing the first instruction of the function with an `int3` or `jmp` instruction that transfers control to instrumentation code [Hollingsworth et al. 1994; Hunt and Brubacher 1999; Yegneswaran et al. 2008]. The stolen-bytes technique bypasses the patched entry point by creating a copy of the library function's first basic block and routing the program's control flow through the copy instead of the 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 runtime, 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 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 get their targets from IAT entries 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, thereby allowing 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 library function.

*Approaches.* Tools that instrument Windows API functions with patch-based techniques (e.g., CWSandbox [Willems et al. 2007], Detours [Hunt and Brubacher 1999], TTAnalyze [Bayer et al. 2006]) face one of two problems when instrumenting malware that use the stolen-bytes technique, depending on whether the byte stealing happens before or after the instrumenter patches the API functions. If the byte stealing occurs first, the instrumenter's code patches will have no effect on the program's execution, as the Windows API function's patched first block will not execute. If the code patching occurs first, however, the packer's byte-stealing code will steal the patched code block instead of the original code block. In this case, if the tracing techniques have used an `int3` instruction, the packer will steal the `int3` instruction and copy it to a new address, resulting in a trap at an unexpected address that current patch-based techniques would not handle correctly. Our Dyninst tool avoids this scenario by ensuring that when the packed program is stealing bytes from Windows API functions it steals original code bytes and not the patched code that Dyninst places at the function's entry points. To

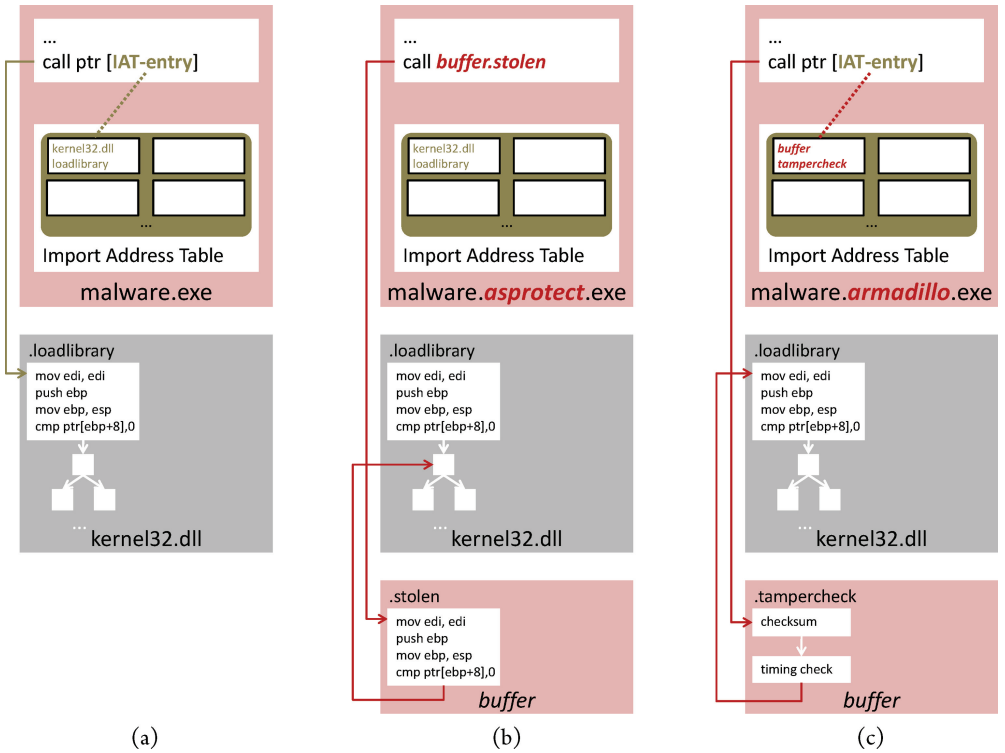


Fig. 6. Packed programs may interpose on inter-library calls originating from their packed payload to resist reverse-engineering and code-patching techniques. In a nonpacked program, as in part (a), calls to functions in shared libraries use indirect call instructions to read their targets from Import 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 antitampering techniques before forwarding the call on to its intended destination.

reliably detect API-function invocations in the presence of the stolen-bytes technique, Dyninst instruments all control transfers that could enter any part the function.

There are alternative instrumentation and program-monitoring techniques that do not rely on code patching. 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 still 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 on tools that apply binary translation techniques to the entire monitored system (e.g., Qemu [Bellard 2005] provides the foundation for several malware analysis tools [Bayer et al. 2006; Chow et al. 2008; Moser et al. 2007]). Whole-system monitoring has also been achieved with 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 close to this goal, with the added benefit that executing the malware in a virtual machine allows the infected guest system to be rolled back to a clean

checkpoint. The primary drawback of whole-system monitoring is that the monitoring tool sits outside of the guest system and must employ virtual machine introspection techniques to make sense of what is happening inside the guest. Emerging introspection tools such as LibVMI [Payne 2011], which originated as the XenAccess project [Payne et al. 2007], perform the necessary introspection techniques for 32-bit Windows and Linux 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 [Aucsmith 1996]. These packers take a checksum over the program's code bytes and then recalculate that checksum at runtime to detect modifications to portions of the program's code or data. In most self-checksumming attempts, the packer checksums its own packer bootstrap code to detect attempts to reverse-engineer the binary unpacking process (e.g., ASProtect). Some packers also checksum the program's payload once it has been unpacked, to protect it from tampering and reverse-engineering (e.g., PECompact). Packers may also checksum the program binary file to protect the integrity of both the packer metacode and the packed payload code (e.g., Yoda's Protector). Finally, packers often read from their own code without explicitly intending to perform self-checksumming. For example, the stolen-bytes technique reads binary code to "steal" code bytes at function entry points. In other cases, code overwrites work by modifying instructions relative to their current value. Finally, even conventional optimizing compilers read from code when they generate instructions that grab constants from nearby code bytes that happen to match a needed value, thereby obtaining some modest savings in overall code size.

*Approaches.* The standard method of defeating self-checksumming techniques (pioneered by Wurster et al. [2005]) is to redirect all memory access instructions at an unmodified shadow copy of the program's code bytes, while executing patched code bytes. Wurster et al. accomplished this by modifying the operating system's virtual memory management so that the instruction TLB caches patched code pages while the data TLB caches shadow code pages. Rosenblum et al. showed that the same technique can be achieved with a modified virtual machine monitor [Rosenblum et al. 2008a]. This shadow-memory-based approach becomes more difficult for programs that unpack or overwrite their code at runtime, however, as the dynamically written code bytes must be present in execution-context memory so that they can execute [Giffin et al. 2005]. Our Dyninst tool has to deal with this problem as it implements a similar self-checksumming defense that uses binary instrumentation to redirect memory access instructions at shadow memory. We allow dynamically unpacked and overwritten code to execute by copying changes in the shadow memory to the patched memory after every code overwrite and before every control transfer into dynamically unpacked code.

### 3.6. Unpacking

Analysts often engage in binary unpacking to subject packed code to static code patching and analysis techniques. Binary unpacking is a reverse-engineering task with two major subparts: reconstructing an executable file that captures dynamically unpacked code and data bytes, and bypassing the metacode that the packer tool bundles into the packed payload binary. The difficulty of accomplishing these unpacking tasks ranges widely from packer to packer; some binaries can be readily unpacked by automated tools [Böhne and Holz 2008; Yegneswaran et al. 2008], while others are extremely difficult to unpack, even for expert analysts. In simple cases, binary unpacking involves identifying the payload code's original entry point and executing the program until it jumps to the OEP, by which time the packer's bootstrap code is done executing and the binary has been unpacked; at this point the analyst can copy the program's payload of unpacked code and data bytes from the program's memory

image into a reconstructed program binary. For simple packers, the packer metacode consists entirely of bootstrap code and can be bypassed by setting the reconstructed binary's entry point to the OEP. The last step in constructing a working executable is to rebuild the payload binary's imported function data structures so that the Windows loader can link up calls to imported functions.

Since many customers of packer tools wish to prevent these reverse-engineering efforts, most binary packers take steps to counter them. In Section 3.1 we discussed defensive techniques that attack the code extraction task and in this section will focus on techniques that make packer metacode difficult to bypass. We describe the antiunpacking techniques used by the 12 tools of this study; for a broader discussion of possible antiunpacking 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 and obfuscating the control transfer to the OEP. Common techniques for obfuscating this control transfer include indirect control flow, call-stack tampering, exception-based control flow, and self-modifying code. The control transfer is often unpacked at runtime, 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 the 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 often resorted to manual-labor-intensive techniques to generate custom unpacker tools for specific packers [BitDefender 2007; Szappanos 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 is instrumental in creating unpacker tools for other binaries created by the same packer. However, recent developments have made this custom unpacking approach increasingly untenable. The first problem is 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 problematic for packer tools that place a polymorphic code generator in the packed binary itself, because 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, but the biggest obstacle for this approach came in 2008, when the percentage of packed malware binaries that used customized and private packing techniques rose to 33% of the total [Bustamante 2008b, 2008a]. Since then, work by Coogan et al. and Debray and Patel has made strides towards automating custom-unpacker creation by extracting packer routines from packed programs [Coogan et al. 2009; Debray and Patel 2010]. The sheer number of unique packing approaches means that their techniques are probably not well-suited for antivirus tools, but they could be used for offline forensic analysis once the techniques have been shown to work on broad collections of real-world malware.

Malware analysis tools have largely switched to employing a variety of general-purpose unpacking techniques that find the OEP of packed programs with principled heuristics. The most reliable heuristics select the OEP transfer from among the set of control transfers into dynamically written memory regions. 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 packers that place code and writeable data on the same memory pages. Pandora's Bochs [Böhne and Holz 2008] selects among control transfers to written memory by assuming that the last of these control transfers is the control transfer to the OEP. The Justin unpacker instead 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 (this heuristic fails on Yoda's Protector), and that command-line arguments supplied to the packed program are moved to the stack prior to the OEP control transfer. Though these heuristics are based on sound principles, it's worth noting that many packer programs have adapted to evade similar heuristics in the past.

*3.6.2. Payload-Code Modification.* The second challenge of antireverse-engineering is preventing the analyst from bypassing the defensive metacode that the packer tool places in the binary. The metacode of most packed binaries is easily bypassed because it only serves to bootstrap the packed payload into memory and never executes again after transferring 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 that transfer control to their metacode so that the metacode executes even after the control transfer to the OEP. The easiest place to insert hooks to metacode callback routines is at inter-library calls. This hooking technique involves replacing imported function addresses in the Import Address Table with the callback addresses, as shown in Figure 6(c). The callback metacode may perform timing checks or probe the binary to detect tampering prior to transferring control to the imported function corresponding to the IAT entry (e.g., Armadillo).

The IAT-hooking technique is attractive because it requires no modification to the payload code itself, though some packer tools do disassemble the payload code and modify it. For example, rather than modifying IAT entries, ASProtect replaces the indirect `call ptr[<IAT-entry>]` instructions by which compilers reference the IAT with direct calls to ASProtect metacode callbacks, as seen in Figure 6(b). To make matters worse, ASProtect and Armadillo place these callbacks in memory buffers allocated by the VirtualAlloc Windows API function that are outside of the binary's memory image. 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). These compilers add metacode to the payload program's source code with source-code instrumentation techniques that are far easier to implement than their binary-instrumentation counterparts.

*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 originally targeted function address. This technique may be possible to automate, but would need 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 is not easily automated.

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 [Dinaburg et al. 2008; Guilfanov 2005; Kang et al. 2007], create unpacked binaries that are amenable to static analysis but cannot actually execute [Christodorescu et al. 2005; Yegneswaran et al. 2008], or work only on a subset of packed binaries [Böhne and Holz 2008; Coogan et al. 2009; Debray and Patel 2010].

## 4. OBFUSCATION STATISTICS

We now proceed to quantify the occurrence of code obfuscations to show which of the obfuscations are most prevalent in real-world malware. To this end, we study 12 of the 15 code-packing tools that, according to a survey performed by Panda Research [Bustamante 2008b], are most often used to obfuscate real-world malware. We used 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 three packers that we have not yet been able to study with Dyninst are Execryptor (4.0% market share), Themida (3.0%), and Armadillo (0.4%). We also omit the Nullsoft Scriptable Install System (3.6%) from this study, which is a toolbox for building custom installer programs that may or may not include code-packing and obfuscation techniques. Though Dyninst can analyze Nullsoft-packed binaries, these binaries do not adhere to a standard set of obfuscations or packing techniques that we can claim to be representative of the Nullsoft system.

The results of our study are presented in Table I. We sort the packers based on the market share that they obtained on malware binaries, as reported by Panda Research. 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.

*Dynamic Code.* All of the tools in Table I pack the code and data bytes of the binary that they obfuscate through compression or encryption. As seen in the row A1 of the table, most of these binaries unpack in multiple stages. We estimate the number of unpacking stages by considering each control transfer to code bytes that have been modified since the previous stage to be a new stage. We categorize events of unpacked code overwriting existing code as code overwrites rather than instances of code unpacking, and list counts of these overwrites in row A2. We observed overwrite sizes 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. As shown in the row A3, some 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 update the PC and fix any control transfers to dead instrumented code so that the new code will execute instead [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, usually to implement virtual functions or inter-library calls. As seen in row B1 of Table I, indirect control transfers are extremely prevalent in some packers, but not in others. In row B2 we see that indirect jumps are not used by most packers but that ASProtect and Yoda's Protector use them extensively. The `ret` instruction is a form of indirect jump that several packers use in nonstandard ways to obfuscate their control flow (row B3). `Call` instructions are misused even more frequently, in which case they are usually paired

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 | 5.2 | 2.6 | 3.3 | 0.2 | 1.1 | 28.0 | 10.2 | 1.5 |
| | % obfuscated size of 48KB payload | 46% | 63% | 42% | 50% | 73% | 56% | 54% | 46% | 46% | 350% | 100% | 46% |
| Dynamic Code | A1 Unpack instances | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 1 | 0 | 4 | 5 | 3 |
| | A2 Overwrite instances | 0 | 0 | 1 | 2 | 0 | 0 | 4 | 0 | 1 | 205 | 10 | 0 |
| | A3 Overwrite byte count | 0 | 0 | 64 | 167 | 0 | 0 | 179 | 0 | 194 | 51 | 9889 | 0 |
| | A4 Overwrite of executing function | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 4 | 0 |
| Instruction Obfuscations | B1 % of calls that are indirect | 83% | 75% | 95% | 44% | 11% | 28% | 30% | 92% | 16% | 7% | 5% | 28% |
| | B2 Count of jumps that are indirect | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 69 | 45 | 0 |
| | B3 Non-standard uses of ret instruction | 0 | 0 | 1 | 1 | 0 | 0 | 4 | 0 | 0 | 83 | 25 | 0 |
| | B4 Non-standard uses of call instruction | 1 | 4 | 0 | 24 | 0 | 2 | 2 | 1 | 0 | 60 | 85 | 0 |
| | B5 Obfuscated constants | ✓ | ✓ | × | ✓ | ✓ | ✓ | × | ✓ | × | ✓ | ✓ | × |
| | B6 Fuzz-Test: unusual instructions | × | ✓ | × | ✓ | ✓ | × | × | × | × | ✓ | ✓ | × |
| Code Layout | C1 Functions share blocks | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 2 | 43 | 0 | 0 |
| | C2 Blocks share code bytes | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 1 | 0 |
| | C3 Interleaved blocks from different funcs | 0 | 0 | 1 | 14 | 0 | 4 | 1 | 4 | 3 | 52 | 2 | 1 |
| | C4 Function entry block not first block | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 10 | 3 | 1 |
| | C5 Inter-section functions | ✓ | × | ✓ | ✓ | × | × | × | ✓ | ✓ | × | × | ✓ |
| | C6 Inter-object functions | × | × | × | × | × | × | × | × | × | ✓ | × | × |
| | C7 Code chunked into tiny basic blocks | × | ✓ | × | ✓ | × | × | × | × | × | ✓ | ✓ | × |
| | C8 Anti-linear disassembly | × | ✓ | × | × | × | × | ✓ | × | × | ✓ | ✓ | × |
| | C9 Flags used across functions | × | × | ✓ | × | × | × | × | ✓ | ✓ | ✓ | × | × |
| | C10 Writeable data on code pages | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| | C11 Fuzz-Test: fallthrough into non-code | × | ✓ | ✓ | ✓ | × | × | × | × | × | ✓ | ✓ | × |
| Anti-Rewriting | D1 Code bytes in PE header | 0 | 0 | 102 | 0 | 0 | 0 | 0 | 158 | 0 | 0 | 0 | 182 |
| | D2 Code unpacked to VirtualAlloc buffer | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 |
| | D3 Polymorphism engine in packer tool | × | ✓ | × | × | × | × | × | × | × | ✓ | × | × |
| | D4 Polymorphism engine in packed binary | × | × | × | × | × | × | × | × | × | ✓ | × | × |
| | D5 Checksum of binary file | × | × | × | × | × | × | × | × | × | × | ✓ | × |
| | D6 Checksum of malware payload | × | × | × | ✓ | × | × | × | × | × | ✓ | × | × |
| | D7 Checksum of packer bootstrap code | × | × | × | ✓ | × | × | × | × | × | ✓ | ✓ | × |
| Anti-Tracing | E1 Exception-based control transfers | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 8 | 0 |
| | E2 Windows API callbacks to packer code | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 89 | 0 |
| | E3 Metacode uses WinAPI funcs not in IAT | ✓ | ✓ | ✓ | × | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| | E4 Timing check | × | × | × | × | × | × | × | × | × | × | ✓ | × |

with a pop instruction somewhere in the basic block at the call target (row B4). 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 (row B5). In row B6, we show that some of these packers fuzz test analysis tools by including rarely used instructions that malware emulators may not handle properly, such as floating point instructions. ASProtect's fuzz testing is particularly thorough in its extensive use of segment selector prefixes on instructions that do not require them.

*Code Layout.* Though compilers occasionally share code by inlining or outlining code sequences that are used by multiple functions, some packed malware programs do so aggressively, as seen in row C1. On the other hand, compilers do not exploit the dense variable-length $\times 86$ instruction set to construct valid overlapping instruction streams as malware sometimes does (row C2). These sequences are necessarily short, and are not particularly useful, but serve to break the assumptions made by some analysis tools [Kruegel et al. 2004; Nanda et al. 2006]. Code sharing can result in some interleaving of blocks from different functions, but not to the extent by which packers like ASProtect and EXEcryptor interleave function blocks all across large code sections (row C3). These strange function layouts break the common assumption that the function entry block is also the block at the smallest address (row C4). To make matters worse, packer function blocks are frequently spread across code sections (row C5) and even across code objects (row C6), in which case part of the code is often in a 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 (row C7). 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). Linear-sweep disassemblers get confused by the padding bytes that packers put in-between the chunked basic blocks to hide the actual instructions from the disassembler (row C8). Another prevalent characteristic of obfuscated code is that the contracts between calling functions and called functions are highly nonstandard. A good example of this is when a called function sets a status flag that is read by the caller, which is something that $\times 86$ compilers never do, even in highly optimized code (row C9). Similarly, compilers avoid placing writeable data on code pages, since this has dangerous security implications, whereas obfuscated programs do this extensively (row C10). 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 (row C11).

*Antirewriting.* 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. Row D1 of this table section shows that the UPack, FSG, and MEW packers put code in the PE header where it will be overwritten by a binary rewriter if it replaces the PE header. 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 (row D2). Polymorphism is also a problem for binary rewriting because polymorphic packer tools produce different packed binaries each time they pack the same payload executable (row D3). To make

matters worse, 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 (row D4). Finally, some packers use self-checksumming to detect modifications to their code. Checksums are most often calculated over the packed program's bootstrap code (row D5), though they are sometimes also calculated over the unpacked payload code (row D6) and over the packed binary file itself (row D7).

*Antitracing.* 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 resist tracing with exception-based control transfers, which obfuscate the program's control flow and cause errors in emulators that do not detect fault-raising instructions and handle exceptions properly (row E1). Some packers also use library functions that take function pointers as parameters and call back into the binary, as a way of circumventing tracers that only patch the malware binary itself. All of the callbacks we have observed leverage the For normal binaries, analysts look at the Import Address Table data structure to determine which Windows API functions are used by the program [Hunt and Brubacher 1999], but most malware binaries circumvent the IAT for many of their Windows API calls (row E3), limiting the effectiveness of this strategy. Timing checks are used by such packers as Yoda's Protector, Armadillo, and Themida, to detect significant slowdowns that are indicative of single-step tracing or interactive debugging. Finally, in the last table row we list packed programs that use Windows API calls to detect that a debugger process is attached to their running process.

*Packer Personalities.* We now supplement the statistical summary of each packer tool in Table I with a description of the prominent characteristics of each 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 degree of obfuscation present in the packer metacode.

*FSG (158 bytes).* The Fast, Small, Good EXE packer's goals are stated in its name. Its metacode code is fast and small (its 158 bytes of metacode 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. A good example of this is a 7-byte code sequence in which FSG initializes its registers by pointing the stack pointer at the binary and executing a popad instruction. These techniques make FSG's bootstrap code small enough 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 among the few packer tools that are open sourced. Somewhat surprisingly, 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 custom packers can be derived from its open-source code, like the UPX-based packer that was used to pack Conficker A [Porras et al. 2009].

*UPack (629 bytes).* The UPack packer manages to consistently generate the smallest packed binaries of any tool we have studied, while including several nasty obfuscation techniques. The UPack authors repurpose nonessential fields of the binary's PE and DOS headers with extraordinary effectiveness, stuffing them with code, data, function

pointers, and the import table, and even causing the DOS and PE headers to overlap. These PE header tricks cause robust binary analysis tools such as IdaPro and OllyDbg to misparse some of the PE's data structures. Meanwhile, its strange function layouts and code overwrite techniques cause problems for other types of 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. PolyEnE also varies the address of the Import Table and the memory size of the section that contains the bootstrap code.

*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. NSPack employs two notable defenses against binary rewriting in that it only unpacks if its symbol table is empty and it places the bulk of its bootstrap into a memory buffer that it deallocates after it has unpacked the payload code.

*nPack (2719 bytes).* The bootstrap code of nPack-generated binaries appears to be mostly compiler generated, without much emphasis on compactness. Other reasons for its large bootstrap code are its single unpacking phase and its support for packing binaries that use 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 obfuscating its control flow. ASPack tampers with the call stack, uses `call` and `ret` instructions for nonstandard purposes, and overwrites an instruction operand at runtime to modify a control-flow target.

*PEtite (5197 bytes).* PEtite's binary code is not heavily obfuscated, but it does contain several anti-analysis techniques. Like FSG, PEtite redirects the stack pointer to a nonstandard location (to a memory buffer in version 2.4 and at the binary in version 1.3), but PEtite does not restore the stack pointer until just before jumping to the program's OEP. This appears to be an antitracing technique designed to thwart tools that track the packed program's use of the call stack. The large size of PEtite's metacode is owing to its use of only one unpacking loop, which appears to consist mostly of compiler-generated code.

*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 antidebugging techniques. The author of Yoda's Protector wrote the first paper on the use of exception-based control flow for the purposes of binary obfuscation [Danehkar 2005], and accordingly, Yoda's Protector uses exception-based control transfers more extensively than any other packer. Of the packer tools we studied with Dyninst, Yoda's Protector is the only tool to checksum its binary file or perform timing checks.

*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 antidisassembler obfuscations. For example, ASProtect causes the program to parse into ASProtect-packed binaries that carry a polymorphic code generation engine which adds considerably to the difficulty of automatically reverse-engineering binaries

that have been packed by ASProtect. Reverse-engineering is also made particularly difficult by ASProtect's use of the stolen-bytes techniques that it pioneered.

## 5. CONCLUSION

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 malware binaries. In this article, we have described the obfuscations that we have seen in malware programs and presented the techniques by which researchers have dealt with these obfuscations. We have also quantified the prevalence of these obfuscation techniques by reporting on the obfuscations that widely used packer tools apply to malware binaries. By doing so, we hope to help future malware analysis tools know which obfuscations their tools must account for. We also hope to guide future work in deobfuscation techniques so that researchers will build solutions for the most serious problems that malware analysts must face.

## REFERENCES


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

Aucsmith, D. 1996. Tamper resistant software: An implementation. In *Proceedings of the Workshop on Information Hiding*. 317–333.

Babic, D., Martignoni, L., McCamant, S., and Song, D. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11).*

Balakrishnan, G. and Reps, T. 2004. Analyzing memory accesses in ×86 executables. In *Proceedings of the Conference on Compiler Construction (CC'04)*. 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 *Proceedings of the Symposium on Operating Systems Principles.*

Bayer, U., Moser, A., Kruegel, C., and Kirda, E. 2006. Dynamic analysis of malicious code. *J. Comput. Virology 2*, 1, 66–77.

Bellard, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference.*

Bernat, A. R. and Miller, B. P. 2011. Anywhere, any time binary instrumentation. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'11).*

Bernat, A. R., Roundy, K. A., and Miller, B. P. 2011. Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11).*

BitDefender. 2007. BitDefender anti-virus technology. White Paper. http://www.bitdefender.com/news/?f_year=2007.

Bohne, L. and Holz, T. 2008. Pandora's bochs: Automated malware unpacking. M.S. thesis, University of Mannheim. http://archive.hack.lu/2009/PandorasBochs.pdf.

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

Bruschi, D., Martignoni, L., and Monga, M. 2007. Code normalization for self-mutating malware. *IEEE Secur. Privacy 5*, 2.

Bustamante, P. 2008a. Malware prevalence. Panda Research web article. http://research.pandasecurity.com/malware-prevalence-august-2008/.

Bustamante, P. 2008b. Packer (r)evolution. Panda Research web article. http://research.pandasecurity.com/packer-revolution/.

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

Chow, J., Garfinkel, T., and Chen, P. M. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference.* 24.

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

Cifuentes, C. and Fraboulet, A. 1997. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance (ICSM'97).*

Cifuentes, C. and Van Emmerik, M. 1999. Recovery of jump table case statements from binary code. In *Proceedings of the International Workshop on Program Comprehension (ICPC'99)*.

Collberg, C., Thomborson, C., and Low, D. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'98)*.

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

Danehkar, A. 2005. Inject your code into a portable executable file. http://www.codeproject.com/KB/system/inject2exe.aspx.

Dark Paranoid. 1998. Engine of eternal encryption. Moon bug 7. http://www.eset.com/us/threat-center/encyclopedia/threats/darkparanoid/.

Debray, S. and Evans, W. 2002. Profile-guided code compression. In *Proceedings Conference on Programming Language Design and Implementation (PLDI'02)*.

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

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

Falliere, N. 2007. Windows anti-debug reference. Infocus web article. http://www.securityfocus.com/infocus/1893.

Ferrie, P. 2008a. Anti-unpacker tricks. In *Proceedings of the International CARO Workshop*.

Ferrie, P. 2008b. Anti-unpacker tricks - part one. Virus Bulletin. www.virusbtn.com/pdf/magazine/2008/200812.pdf.

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 *Proceedings of the Annual Computer Security Applications Conference (ACSAC'05)*.

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 *Proceedings of the Virus Bulletin Conference*.

Guilfanov, I. 2005. Using the universal pe unpacker plug-in included in ida pro 4.9 to unpack compressed executables. 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 *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID'08)*.

Harris, L. C. and Miller, B. P. 2005. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News 33*, 5.

Hex-Rays Decompiler. 2011. Version 1.6. http://hex-rays.com.

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

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

Jacobson, E. R., Rosenblum, N. E., and Miller, B. P. 2011. Labeling library functions in stripped binaries. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'11)*.

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

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

Lakhotia, A., Kumar, E. U., and Venable, M. 2005. A method for detecting obfuscated calls in malicious binaries. *Trans. Softw. Engin. 31*, 11.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *Proceedings of the International Conference on Compiler Construction (CC'08)*.

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

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

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 *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05).*

MADOU, M., ANCKAERT, B., DE SUTTER, B., AND DE BOSSCHERE, K. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the ACM Workshop on Digital Rights Management*. 25

MAEBE, J. AND DE BOSSCHERE, K. 2003. Instrumenting self-modifying code. In *Proceedings of the Workshop on Automated and Algorithmic Debugging.*

MAEBE, J., RONSSE, M., AND DE BOSSCHERE, K. 2002. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the Workshop on Binary Translation held in conjunction with the Conference on Parallel Architectures and Compilation Techniques (PACT'02).*

MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'07).*

MILLER, B. P., FREDRIKSEN, L., AND SO, B. 1990. An empirical study of the reliability of unix utilities. *Comm. ACM 33*, 12.

MOSER, A., KRUEGEL, C., AND KIRDA, E. 2007. Exploring multiple execution paths for malware analysis. In *Proceedings of the Symposium on Security and Privacy.*

MUTH, R. AND DEBRAY, S. 2000. On the complexity of ow-sensitive dataow analyses. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'00).*

NANDA, S., LI, W., LAM, L.-C., AND CHIUEH, T.-C. 2006. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the Symposium on Code Generation and Optimization (CGO'06).*

PARADYN TOOLS PROJECT. 2011. ParseAPI programmer's guide. Version 7.0.1. http://www.paradyn.org/html/manuals.html.

PAYNE, B. D. 2011. LibVMI, version 0.6. http://vmitools.sandia.gov/.

PAYNE, B. D., CARBONE, M., AND LEE, W. 2007. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'07).*

PERRIOT, F. AND FERRIE, P. 2004. Principles and practise of x-raying. In *Proceedings of the Virus Bulletin Conference.*

POPOV, I., DEBRAY, S., AND ANDREWS, G. 2007. Binary obfuscation using signals. In *Proceedings of the USENIX Security Symposium.*

PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. 2009. A foray into conficker's logic and rendezvous points. Tech. rep., SRI International. https://www.usenix.org/legacy/event/leet09/tech/full_papers/porras/porras.pdf.

PRAKASH, C. 2007. Design of $\times$86 emulator for generic unpacking. In *Proceedings of the Association of Anti-Virus Asia Researchers International Conference.*

QUINLAN, D. 2000. Rose: Compiler support for object-oriented frameworks. In *Proceedings of the Conference on Parallel Compilers (CPC'00).*

QUIST, D. AND SMITH, V. 2007. Covert debugging: Circumventing software armoring techniques. Blackhat USA. http://www.blackhat.com/presentations/bh-usa-07/Quist_and_Valsmith/Whitepaper/bh-usa-07-quist_and_valsmith-WP.pdf.

ROSENBLUM, N. E., COOKSEY, G., AND MILLER, B. P. 2008a. Virtual machine-provided context sensitive page mappings. In *Proceedings of the Conference on Virtual Execution Environments (VEE'08).*

ROSENBLUM, N. E., MILLER, B. P., AND ZHU, X. 2010. Extracting compiler provenance from program binaries. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'10).*

ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. 2008b. Learning to analyze binary computer code. In *Proceedings of the Conference on Artificial Intelligence (AAAI'08).*

ROUNDY, K. A. 2012. Hybrid analysis and control of malicious code. Ph.D. thesis, Department of Computer Science, University of Wisconsin. http://pages.cs.wisc.edu/~roundy/dissertation.pdf.

ROUNDY, K. A. AND MILLER, B. P. 2010. Hybrid analysis and control of malware. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID'10).*

ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. 2006. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'06).*

RUSSINOVICH, M. AND COGSWELL, B. 1997. Windows NT system call hooking. *Dr. Dobb's J. 22*, 1.

SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. 2002. Disassembly of executable code revisited. In *Proceedings of the 9$^{th}$ Working Conference on Reverse Engineering.* 45.

SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. 2008. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08).*

Sharif, M., Lanzi, A., Giffin, J., and Lee, W. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of the Symposium on Security and Privacy*.

Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. 1993. Binary translation. *Comm. ACM 36*, 2.

Stepan, A. E. 2005. Defeating polymorphism: Beyond emulation. In *Proceedings of the Virus Bulletin Conference.*

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

Szappanos, G. 2007. Exepacker blacklisting. Virus Bulletin.

Theiling, H. 2000. Extracting safe and precise control flow from binaries. In *Proceedings of the Conference on Real-Time Computing Systems and Applications.* 23.

Trilling, S. 2008. Project green bay-calling a blitz on packers. CIO digest: Strategies and analysis from symantec. http://eval.symantec.com/mktginfo/enterprise/articles/biodigest_october08_magazine.pdf.

Van Emmerik, M. and Waddington, T. 2004. Using a decompiler for real-world source recovery. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'04).* 26.

Vigna, G. 2007. Static disassembly and code analysis. In *Malware Detection. Advances in Information Security*. Vol. 27, Springer, 19–41.

Willems, C., Holz, T., and Freiling, F. 2007. Toward automated dynamic malware analysis using cwsandbox. In *Proceedings of the Symposium on Security and Privacy.*

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

Yason, M. V. 2007. The art of unpacking. Blackhat USA. http://www.blackhat.com/presentations/bh-usa-07/Yason/Presentation/bh-usa-07-yason.pdf.

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.

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

Received March 2012; revised August 2012; accepted October 2012