HYBRID ANALYSIS AND CONTROL OF MALICIOUS CODE

by

Kevin A. Roundy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 5/1/2012

The dissertation is approved by the following members of the Final Oral Committee:

Barton P. Miller, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Shan Lu, Assistant Professor, Computer Science
Thomas Ristenpart, Assistant Professor, Computer Sciences
Nigel Boston, Professor, Electrical and Computer Engineering, and Mathematics

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

State of the art analysis techniques for malicious executables lag significantly behind their counterparts for compiler-generated executables. This difference exists because 90% of malicious software (also known as malware) actively resists analysis. In particular, most malware resists static attempts to recover structural information from its binary code, and resists dynamic attempts to observe and modify its code.

In this dissertation, we develop static and dynamic techniques and combine them in a hybrid algorithm that preserves the respective strengths of these techniques while mitigating their weaknesses. In particular, we build structural analyses with static parsing techniques that can disassemble arbitrarily obfuscated binary code with high accuracy, and recover the structure of that code in terms of functions, loops, and basic blocks. We develop dynamic techniques to identify transitions into statically unreachable code and respond to malware that overwrites its code. These dynamic techniques remove overwritten and unreachable code from our analysis and trigger additional parsing at entry points into un-analyzed code, before this code executes. Our stealthy instrumentation techniques leverage our structural analysis to stealthily and efficiently instrument binary code that resists modification. These instrumentation techniques hide the modifications they make to the binary code, and the additional space that they allocate in the program's address space to hold instrumentation.

We demonstrate the utility of our techniques by adapting the Dyninst 7.0 binary analysis and instrumentation tool so that its users can analyze defensive malware code in exactly the same way that they analyze non-defensive binaries. We also build customizable malware analysis factories that perform batch-processing of malware binaries in an isolated environment, to help security companies efficiently process the tens of thousands of new malware samples that they receive each day. Finally, we use our

analysis factory to study the most prevalent defensive techniques used by malware binaries. We thereby provide a snapshot of the obfuscation techniques that we have seen to date, and demonstrate that our techniques allow us to analyze and instrument highly defensive binary code.

1 INTRODUCTION

Malicious software infects computer systems at an alarming rate, causing economic damages that are estimated at more than one hundred billion dollars per year [84]. Immediately upon discovering a new threat, analysts begin studying its code to determine damage done and information extracted, and ways to curtail its impact. Analysts also study malware for clues about how to clean infected systems and construct defenses. The time analysts spend analyzing the malware represents a time-window during which the malware remains effective. Thus, a primary goal of malware authors is to make these analysis tasks as difficult and resource intensive as possible. This goal explains why 90% of malware program binary files contain *defensive* techniques that protect their code from analysis [17]. These defenses result in labor intensive analyses that take a long time; companies take an average of 18 days to contain cyber-attacks [59] and 31% of cyber-crimes affecting computer users are never solved [83]. By comparison, friendly program binaries are far easier to analyze because current tools can automatically analyze the binary code, enabling analysts to understand it in terms of familiar concepts like functions and loops. Regrettably, these static analysis techniques have not been able to deal with the defensive techniques employed by malware [77]. The result is that malware analysts have turned instead to run-time analysis techniques and typically try to build up an understanding of the malware by looking at traces of executed instructions [37]. This approach frequently overwhelms inexperienced analysts [99], requires the malicious code to be executed before it is understood, and involves modifying the monitored code in detectable ways [3, 43, 106].

This dissertation shows how to build structural analyses for malicious code based on a *hybrid* of static and dynamic analysis techniques. We have developed static techniques that build accurate structural analyses of

defensive code. Our dynamic instrumentation techniques discover code that is not visible to static analysis, while hiding any code modifications that we need to monitor and control the malware [11]. Our static and dynamic analysis techniques are implemented as independent components for analysts to build on [16, 87]. We also show that we can provide structural analysis and dynamic instrumentation of defensive code prior to its execution by combining our static and dynamic techniques in a hybrid algorithm [103]. With our hybrid approach we have analyzed more code in defensive binaries than is possible through static or dynamic analysis alone. We demonstrate the effectiveness and utility of our hybrid analysis approach by providing foundational techniques in a general-purpose framework to speed up the development of future malware analysis tools. The foundational analysis techniques we provide are the following:

- The ability to *find* the code in the program binary. Even in compiler-generated binaries, accurately identifying the code is challenging because binaries also contain data, padding, and junk bytes that are interspersed with the code. Malware binaries make code identification even harder, both by hiding their code from static analysis through obfuscations, code-packing, and code overwrites, and by reducing the accuracy of code identification techniques through the use of valid control transfers into non-code bytes. Our hybrid analysis techniques allow us to find code both statically and dynamically, before the code executes.

- The ability to structurally *analyze* the code prior to its execution. This involves building a structural analysis of the program's control flow that includes functions and loops. Researchers have developed static techniques to mitigate particular code obfuscations [28, 62, 63], but have not built techniques to parse arbitrarily obfuscated code. We build static code-parsing techniques to analyze the code, while our

dynamic techniques trigger further analysis in response to resolved code obfuscations, code unpacking, and code overwrites.

- The ability to *instrument* the code. Instrumentation serves both as a means for the analyst to dynamically analyze and control malware, and to support our own analyses by detecting places that leave statically understandable code and transition to un-analyzed code. Analysis tools built on our framework control the malware's execution by using structural analysis to choose locations at which to change and add to the malware's code. We find additional code at run-time by instrumenting potential transitions into code that is hidden by defensive techniques like code obfuscations and run-time code unpacking. This instrumentation triggers structural analysis of the new code prior to its execution. Since malware frequently employs defensive techniques to detect instrumentation, we developed compensatory techniques to hide the effects of instrumentation from malware [11].

The current ease with which program analysis tools are developed for non-defensive binaries owes a great debt to the existence of binary analysis frameworks such as Dyninst [57], ROSE [97], and Vulcan [113] that provide these basic analysis primitives. This dissertation establishes techniques by which such frameworks can be adapted to work on defensive malware. We demonstrate the practicality of our techniques by implementing them in Dyninst, and by then using Dyninst to successfully analyze defensive malware.

1.1 Background and Challenges

Both static and dynamic analyses are used on malware, but neither is sufficient by itself. While *static analyses* glean information from source code

or program binary files, *dynamic analyses* execute programs and learn about them by observing their run-time behavior. Static analysis techniques have the ability to analyze the binary as a whole, and thereby recover structural information such as functions and loops that are familiar to the programmer [118]. This familiar view can significantly accelerate the analyst's understanding of the code. However, static techniques cannot always predict the targets taken by control-transfer instructions, even through the application of costly dataflow analyses such as pointer aliasing [5, 56]. Dynamic techniques, on the other hand, resolve the targets of all obfuscated instructions that execute. However, their corresponding downside is that they do not find or analyze code that does not execute.

Defensive malware binaries employ a wide variety of techniques to further exploit the weaknesses of static and dynamic analyses. To hide code from static analysis, malware authors employ code obfuscation, code packing, and code overwriting techniques. The goal of *code obfuscation* techniques is to make the program's code difficult to understand. Malware authors apply obfuscations both at the level of individual machine-language instructions and at the function and program level. For example, malware authors frequently obfuscate individual control transfers by using instructions that determine their targets based on register contents or memory values, rather than by using instructions whose targets are explicitly defined by instruction operands. An example of function-level obfuscation is the practice of placing a function's basic blocks in non-contiguous regions of the program's address space. *Code packing* techniques compress or encrypt the binary's malicious code and package the binary with bootstrap code that decompresses the malicious payload into the program's address space at run-time, often doing so incrementally or in multiple stages. Code packing is present in at least 75% of all malware binaries [12, 119]. Taken together, code obfuscations make what little code is present in a packed program binary file hard to analyze, while code packing makes the rest

of the code inaccessible until it is unpacked at run-time. *Code overwriting* techniques replace or modify machine-language instructions at run-time. The effect of code overwrites is that a given address or memory buffer can contain different machine language instructions at different points in the program's execution. This defensive technique makes program analysis harder in two ways. First, for programs that overwrite their code, there is no time at which all of the code is present in the program's memory [32]. Second, when an overwrite occurs, structural analyses of the program's code become both incomplete and invalid. Incompleteness comes because the new code is not included in the analysis, while invalidity occurs because the analysis includes code that no longer exists.

Defensive malware also targets dynamic techniques, both by protecting the integrity of its code and by trying to detect that it is executing in a monitored environment. Integrity checks target patch-based instrumentation of the code, and frequently involve scanning the program's code space at run-time to compare the checksum of that region to the checksum of the original code. If the checksums do not match, the program's integrity has been violated. Environment checks look for signs that the program is executing in a supervised environment. For example, a malware program may attempt to detect that it is executing in a virtual machine [106] or that a debugger process is attached to it [42]. When malware detects tampering or a supervised environment, it exits early, typically before exhibiting its intended malicious behavior. These defensive techniques pose significant challenges to static and dynamic analysis and are discussed at greater length in Chapter 2.

1.2 Techniques

The goal of our hybrid analysis techniques is to deal with the nasty defensive mechanisms employed by malware binaries so that analysts can

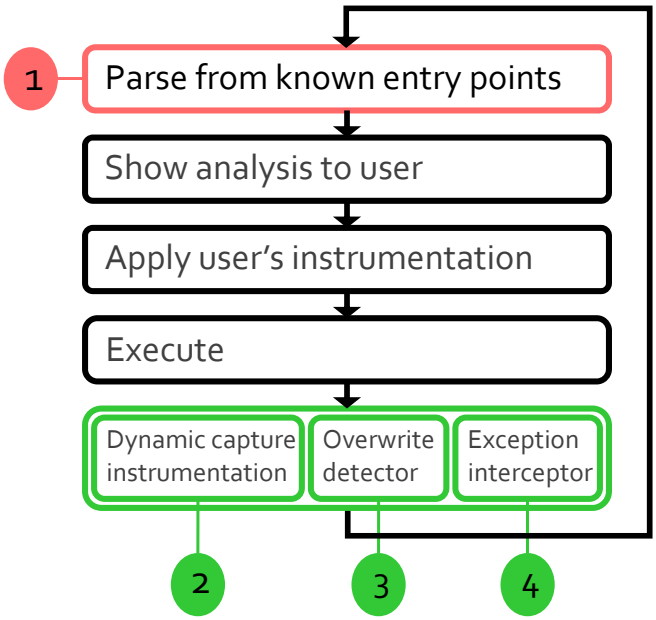


Figure 1.1: Hybrid algorithm for our binary analysis and instrumentation framework. Technique 1 (in red) applies static parsing techniques while the program is stopped, to build a structural analysis of the code. The analysis tool built on our framework consumes this analysis, and specifies instrumentation that we apply to the binary. We then execute the program, with dynamic analysis techniques 2, 3, and 4 (in green) in place to detect transitions to code that may require further analysis and instrumentation. We use the new entry points found by these techniques to start a new iteration of our hybrid algorithm.

naively analyze and instrument defensive malware as if it contained normal binary code. To this end, we develop static and dynamic analyses that are hardened to withstand defensive techniques. By incorporating these techniques into the hybrid algorithm shown in Figure 1.1, we mitigate the weaknesses of static and dynamic analyses while providing the benefits of both. Static parsing techniques find code and incorporate it into a structural analysis prior to its execution. We use dynamic analysis techniques to capture any code that was missed by our parser because

of obfuscation, code packing, and code overwrites. These dynamic techniques identify new entry points into the program that we use to seed further parsing, which restarts our iterative analysis and instrumentation algorithm. Further descriptions of algorithmic components in Figure 1.1 are presented below, together with a fifth supporting technique that hides our instrumentation from the malware.

1. **Parsing.** Parsing allows us to find and analyze binary code by traversing statically analyzable control flow starting from known entry points into the code. No existing algorithm for binary code analysis achieved high accuracy on arbitrarily obfuscated binaries, so we create a modified control-flow traversal algorithm [112] with a low false-positive rate. Our initial analysis of the code may be incomplete, but we fall back on our dynamic capture techniques to find new entry points into the code and use them to re-seed our parsing algorithm.

2. **Dynamic Capture.** Dynamic capture techniques allow us to find and analyze code that is missed by static analysis either because it is not generated until run-time or because it is not reachable through statically analyzable control flow. Our static analysis of the program's control flow identifies control transfer instructions that may lead to un-analyzed code; we monitor these control transfers using dynamic instrumentation, thereby detecting any transition to un-analyzed code in time to analyze and instrument it before it executes.

3. **Code Overwrite Monitoring.** Code overwrites invalidate portions of an existing code analysis and introduce new code that has not yet been analyzed. We detect code overwrites by applying DIOTA's [72] method of write-protecting memory pages that contain code and handling the signals that result from write attempts. The hard part is accurately detecting when overwriting ends, as large code regions

are often overwritten in small increments. Accurately detecting the overwrite's end is important because updating the analysis is expensive, and accurate detection allows us to update our analysis only once in response to a group of incremental overwrites. We detect the end of code overwriting in a novel way by using our structural analysis of the overwrite code to detect any loops that enclose the write operations and delaying the analysis update until the loop exits.

4. **Signal- and Exception-Handler Analysis.** Static parsing techniques assume that the program's control flow is entirely regimented by explicit control-transfer instructions, and do not account for the out-of-band control transfers caused by signal- and exception-raising instructions. The reason for this assumption is that statically determining whether an instruction will raise a signal or exception is a hard problem, both in theory and in practice [78, 93]. We use dynamic analysis to resolve signal- and exception-based control transfer obfuscations [44, 93]. We detect signal- and exception-raising instructions and find their dynamically registered handlers through dynamic techniques, and then add the handlers to our analysis and instrument them to control their execution.

5. **Stealthy instrumentation.** We provide the ability to instrument and modify the program without these changes being visible to malware programs that perform integrity checks. These stealthy techniques enable tools built on our dynamic instrumentation framework to hide their modifications, and hides the instrumentation used by our dynamic capture, code-overwrite monitoring, and exception-handler analysis techniques. We prevent the malware both from detecting our changes to its code, and from detecting the extra space that the instrumenter needs to accommodate its instrumentation in the

malware's address space. We hide our modifications by redirecting the program's read and write instructions to a copy of the program's code sections. We hide the instrumentation buffer through further instrumentation-based monitoring of these same read and write instructions.

From an analyst's perspective, analyzing and instrumenting defensive code with our hybrid algorithm is the same as analyzing non-defensive code with a binary analysis and instrumentation framework such as Vulcan [113] or Dyninst [57]. The only difference to the tools built on our framework is that we periodically deliver analysis updates in response to changes in the underlying binary code, affording the tool the opportunity to further analyze and instrument the new and modified portions of the program's control flow graph. Without our hybrid analysis techniques, Dyninst and Vulcan can only analyze and instrument binary code that is not hidden by obfuscation, code packing, and code overwrites.

Our techniques are applicable to a broad range of binary analysis and instrumentation tools. However, we do not compensate for all the ways that these tools control the program's execution, limiting their ability to analyze and control malware. In particular, analysis tools use a variety of mechanisms to supervise the malware's execution, the most prevalent of which are the debugger interface provided by the operating system, software drivers in the operating system, and virtual-machine-monitors outside of the monitored operating system. Each of these monitoring techniques can be detected by malware through different collections of techniques [42, 43, 106]. We implemented our techniques in Dyninst, which uses the OS-provided debugger interface, and though we make a best-effort to hide Dyninst's use of this interface from the malware, we do not claim that our implementation is impervious to all possible anti-debugging attacks. For example, any tool that relies on the debugger interface is fundamentally unable to hide itself from malware with rootkit

components that can directly access kernel data structures.

Where possible, our techniques have been designed to have expensive failure modes (in terms of computation resources) rather than incorrect failure modes. For example, our stealthy instrumentation techniques are based on conservative analyses that can fail to determine that instructions are safe to execute natively; in these cases the cost of our instrumentation techniques increases, but should never instrument programs incorrectly or unsafely. Similarly, our code-overwrite handling techniques respond correctly to code overwrites in all cases, but malware authors could design self-modifying programs for which our techniques would incur high execution-time overheads. For some aspects of our work that do not affect the correctness of our instrumentation techniques, we rely on heuristics that work well in practice but are not completely robust. In particular, a determined malware author could design a binary for which our static analysis of the code mistakenly includes many non-code bytes, and could also limit the accuracy of our function boundary identification techniques. However, these aspects of our structural analysis are not critical for instrumentation correctness or for automated malware analysis; they serve primarily for the use case in which a human analyst wishes to manually consume our structural analyses of the code. Fortunately, human analysts can tolerate some analysis errors. Furthermore, at present there are no alternative techniques that can construct structural analyses that are more robust with respect to these obfuscations.

1.3 Contributions

This dissertation makes the following contributions to the analysis of defensive malware.

- We recover structural constructs such as functions and loops from defensive binary code. In other words, we provide the benefits of static

analysis, including foundational control- and data-flow analyses such as control-flow graph construction and binary slicing [23].

- We provide a binary instrumentation technique that modifies the malware's code in accordance with the user's expectations while hiding its impact from the program. Specifically, we prevent the malware from detecting our changes to its code, and from detecting that we allocate extra space for instrumentation code in the malware's address space (this work was done jointly with Andrew Bernat [11]).

- By combining static and dynamic techniques we allow the analyst to find and analyze code that is beyond the reach of either static or dynamic analysis alone, thereby providing a fuller understanding of the malware's possible behavior. Prior hybrids of static and dynamic analyses do not work on defensive code, and are further limited to only finding and disassembling the code [79] and produce their analysis results only after the program has fully executed [71].

- Our hybrid techniques not only provide the first pre-execution structural analysis of defensive code, but provide this analysis to guide the use of instrumentation. By bringing analysis-guided instrumenters [57, 113] to malware for the first time, we allow analysts to be selective in the program components they monitor, the operations in those components that they select, and in the granularity of data they collect. Current tools that can monitor packed or obfuscated code do not provide flexible instrumentation mechanisms; they trace the program's execution at a uniform granularity, either providing fine-grained traces at the instruction or basic-block level [37, 82], or coarse grained traces (e.g., at interactions with the OS) [127]. These tools either bog the analyst down with irrelevant information (a significant problem for inexperienced analysts [99]), or can only give a sketch of the program's behavior).

- Our pre-execution analysis allows analysts to perform informed and controlled executions of malicious programs based on an understanding of the program's structure and the ability to instrument and modify the malware binary. By contrast, analysts frequently analyze malware in an iterative process, starting with a blind, uncontrolled execution of the malware to help them unpack it [53], followed by application of static techniques to analyze the unpacked code. The analyst must clean up the infected system before executing the program again, this time with some knowledge of the code's structure. This process is often manual-labor intensive [12, 126, 127], and fails to fully analyze packed binaries that exhibit polymorphism (see Chapter 2). Our hybrid approach avoids these problems while achieving the same goal of informed and controlled execution in a single pass.

1.4 Results

Our combination of defense-resistant parsing and instrumentation techniques facilitates rapid creation of higher-level analysis tools. To demonstrate this, we created the following tools using our techniques:

1. We built a general-purpose binary analysis and instrumentation framework that works on malware [11, 103]. We demonstrate that our hybrid techniques enable analysis-guided instrumenters such as Dyninst and Vulcan on malware by implementing our techniques inside of Dyninst. We thereby provide structural analysis of defensive binary code prior to its execution, and use that analysis for controlled malware executions. We demonstrate the efficacy of this framework both on real and representative synthetic malware samples that are highly defensive.

2. We built a customizable malware analysis factory on top of our hybrid analysis and instrumentation framework [103]. Our factory performs batch-processing of malware samples in an isolated environment, producing customizable reports on the structure and behavior of malware samples. Analysts at large security companies receive tens of thousands of new malware samples each day [86] and must process them efficiently and safely to determine which samples are of greatest interest. Our factory meets their needs while automatically finding and analyzing code that is beyond the reach of static or dynamic analysis alone.

3. We built a tool based on our hybrid analysis and instrumentation framework to study the most prevalent defensive techniques used by malware binaries. To determine what those techniques are, we apply our tool to binaries created by the packer toolkits that malware authors most often use to add defensive techniques to their binaries [17]. Using our tool, we catalog and describe the defensive techniques employed by these binaries and report on their relative frequency and impact.

The rest of this dissertation proceeds as follows. In Chapter 2 we discuss the defensive techniques that malware authors use to counter analysis and existing countermeasures to those techniques. In Chapter 3 we discuss our hardened static parsing techniques. In Chapter 4 we present our dynamic code-discovery techniques, including our instrumentation-based dynamic capture techniques for finding additional code, (Section 4.1), our techniques for dealing with code overwrites (Section 4.2), and our approach to dealing with exception- and signal-based control flow (Section 4.3). In Chapter 5 we discuss techniques that make our patch-based instrumentation techniques safe, even for programs that check for code-patches. In Chapter 6 we show our results, and we conclude in

Chapter 7.

2 BACKGROUND AND RELATED WORK

Our work is rooted in the field of obfuscated program analysis, which is a sub-field of program binary analysis. In this chapter we survey previous works on program binary analysis, describe the impact of binary code obfuscations on those works, and discuss approaches that researchers have devised for dealing with these obfuscations.

Prior to our research in this area, there had been no comprehensive studies of the obfuscation techniques that are present in malware code, though there have been studies in the related areas of binary packing [126], anti-debugging [42], and anti-unpacking [44] techniques. Since the first step in analyzing defensive malware is to understand what obfuscations are most-prevalent in real-world malware, we performed a broad examination of the obfuscation techniques used by the packer tools that are most popular with malware authors [18]. 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. We describe obfuscations that make binary code difficult to discover (e.g., control-transfer obfuscations, exception-based control transfers, incremental code unpacking, code overwriting); to accurately disassemble into instructions (e.g., ambiguous code and data, disassembler fuzz-testing, non-returning calls); to structure into functions and basic blocks (e.g., obfuscated calls and returns, call-stack tampering, overlapping functions and basic blocks); to understand (e.g., obfuscated constants, calling-convention violations, chunked control-flow, do-nothing code); and to manipulate (e.g., self-checksumming, anti-relocation, stolen-bytes techniques). For each obfuscation technique, we present the approaches that previous works have taken to dealing with this technique.

Typical surveys of previous works do not include a section on methodology, but we include one because in the process of doing our survey, we

required an active evaluation of prevalent packer tools. We follow the methodology section with a taxonomy of obfuscation techniques used by malware, together with the approaches taken by previous works for dealing with those techniques (Section 2.2). As they read this section, readers may wish to reference Section 2.3, where we provide a summary table of the obfuscation techniques that shows their relative prevalence in real-world malware. Section 2.4 also provides brief descriptions of the packer tools that are most-often used by malware authors. We summarize our discussion of previous works in Section 2.4.

2.1 Methodology

We used a combination of manual and automated analysis techniques to study malware obfuscations. 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 [18] 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 run-time 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 with the assistance of the malware analysis techniques that are the primary contributions of this dissertation [11, 103], as implemented in the Dyninst

binary code analysis and instrumentation tool [16]. This dissertation includes techniques to make instrumentation tools resistant to errors in the analysis [11], however, our initial set of techniques were not resistant to errors, and we therefore ran head-on into nearly every obfuscation technique employed by these programs [103]. We automatically generated statistical reports of defensive techniques employed by these packer tools with our obfuscation-resistant version of Dyninst, and we present those results in Section 2.3 of this chapter.

We also spent considerable time perusing each binary's obfuscated code by hand in the process of getting Dyninst to successfully analyze these binaries, aided by the OllyDbg [128] and IdaPro [52] 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.

2.2 The Obfuscation Techniques

We structure this discussion around foundational binary analysis tasks. For each task, we describe solutions to those tasks, present defensive techniques that malware authors use to complicate the tasks, and survey any counter-measures by which previous works have dealt 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.

Binary Code Extraction

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

Binary code extraction becomes much harder, however, for programs that create and overwrite code at run-time. Defensive malware binaries are the biggest class of programs with this characteristic, though just-in-time (JIT) compilers such as the Java Virtual Machine [67] (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.

Code packing

At least 75% of all malware binaries use code-packing techniques to protect their code from static analysis and modification [12, 119]. A packed

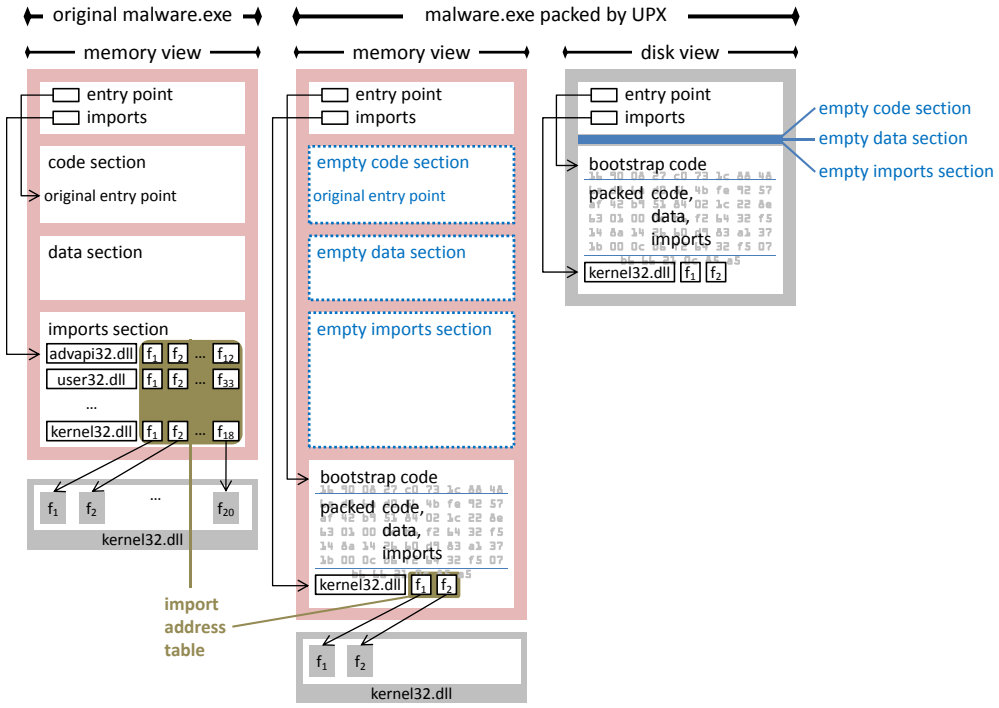


Figure 2.1: Abstract view of a representative packing transformation as performed by versions of the UPX packer up through Version 3.08. UPX compresses malware.exe's code and data, setting the packed binary's entry point to its bootstrap code, which will unpack the code and data into memory at run-time. UPX replaces malware.exe's Import Table and Import Address Table with its own, much smaller tables that import only the LoadLibrary and GetProcAddress functions. UPX uses these functions to reconstruct the original import table at run-time.

binary is one that contains a payload of compressed or encrypted code that it unpacks into its address space at run-time. In practice, most malware authors incorporate this technique by compiling their code into a normal program binary and then processing the binary with a packer tool to create a packed version. Figure 2.1 illustrates the packing transformation performed by UPX; most other packer transformations can be thought of as elaborations on UPX's basic scheme. The packer tool sets the executable's entry point to the entry point of bootstrap code that unpacks the payload and then transfers control to the payload's original entry point (OEP). When the bootstrap code unrolls packed code and data, it places them at the same memory addresses that they occupied in the original binary so that position-dependent instructions do not move and data accesses find the data in their expected locations. UPX also packs the Portable Executable (PE) binary format's Import Table and Import Address Table (IAT) data structures, which list functions to import from other shared libraries. These tables are packed both because they would reveal significant information about the payload code and because they are highly amenable to compression. Upon loading a binary into memory, the Windows linker/loader processes these tables and writes imported function addresses into the IAT, but since packed binaries decompress the payload binary's import tables after load time, packer bootstrap code must fill in the payload binary's IAT itself.

The most common elaboration on this basic recipe for binary packing is one in which portions of the packer's bootstrap code itself are packed. Most packers use a small unpacking loop to decompress a more sophisticated decompression or decryption algorithm that unpacks the actual payload. This incremental approach achieves some space savings, but more importantly, it protects the bulk of the bootstrap code itself from static analysis. The latter is clearly the motivation for ASProtect, 99% of whose metacode is dynamically unpacked, and for other similar "protec-

tor" 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 [92] 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 [28, 35]. 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 [8] and DynamRIO [14] have no difficulty in finding dynamic code, since they do not attempt to discover the program's code until just before it executes. However, this approach does not distinguish between code that is statically present in the binary and code that is created at run-time. Dynamic unpacking tools such as Renovo [60] and EtherUnpack [37] 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 [8] whole-system emulator and the Xen virtual-machine monitor [6], 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 [98], the debugger interface [104], interactive debugger tools [51, 96, 116], and sandboxed emulators [50, 115]. Unpacker tools that have monitored packed malware from the operating system track execution and memory writes at the coarser granularity of memory pages [53, 74]. 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 [74, 127]. The primary limitations of the fine-grained monitoring techniques are that they only identify code bytes that actually executed and they incur orders-of magnitude slowdowns in execution time. Meanwhile, the efficient design of coarse-grained techniques makes them suitable for anti-virus products, but their coarse memory-page granularity means that they cannot identify the actual code bytes on unpacked code pages and that they cannot identify or capture overwritten code bytes.

The aforementioned dynamic unpacking techniques deliver the captured code bytes so that static analysis techniques can be used afterwards to recover the program's instructions and code structure. The Bird interpreter [79] instead applies static code-parsing techniques before the program executes, and uses dynamic techniques to identify and instrument code that is missing from that analysis at run-time. However, Bird does not capture dynamically unpacked code or rebuild its analysis by re-applying its code-parsing techniques.

Code Overwriting

Self-modifying programs move beyond unpacking by overwriting existing code with new code at run-time. Code overwrites often occur on the small end of the spectrum, affecting a single instruction, or even just a single instruction operand or opcode. For example, the ASPack packer modifies the push operand of a `push 0, ret` instruction sequence at run-time to push the 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 paper only overwrite their own metacode. More complex code overwriting scenarios are possible, for example, the MoleBox packer tool and DarkParanoid virus [32] 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 [34] 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 [1, 60].

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 [1]. 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 [71, 127].

Disassembly

Once the code bytes have been captured, static analysis techniques can accurately disassemble most of the code in compiler-generated program binaries, even when those binaries have been stripped of all symbol information [52, 101]. The underlying technique employed by disassembly tools is to disassemble the binary code starting from known entry points into the program. *Linear-sweep* parsing [49, 107] 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 [112, 118] finds instructions by following all statically traversable paths through the program's control-flow starting from known function addresses. This technique is far more accurate, but misses the control transfer targets of instructions that determine their targets dynamically based on register values and memory contents. This weakness of recursive-traversal parsing is not significant for binaries that identify function entry address with symbol information, as most of the missing control transfer targets are to function entries or to jump table entries that can be identified through additional symbol information or static analysis [24]. 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 [102].

Unfortunately for the analyst, binary code can easily flout compiler conventions while remaining efficacious. Anti-disassembly techniques aim to

violate the assumptions made by existing parsing algorithms so that they can both hide code from the disassembler and corrupt the analysis with non-code bytes. Defensive binaries remove all symbol information, leaving only one hint about the location of code in the binary: the executable's entry point. To limit the amount of code that can be found by following control transfer edges from this entry point, these binaries obfuscate their control flow. Compensating for poor code-coverage with compiler-specific knowledge is usually not an option since much of the code is hand-written assembly code and therefore, highly irregular. Additionally, code obfuscations deliberately blur the boundaries between code and non-code bytes to make it difficult to distinguish between the two [26, 68, 93]. 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.

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 2.2a. Since the `call` instruction pushes a return address onto the stack, a misused `call` is usually paired with a `pop` instruction at the call target to remove the PC-relative return address from the stack (see Figure 2.2b). This easily implemented obfuscation attacks analysis tools in in three ways. First, parsers assume that there is a function at the call's target that will return to the call's *fall-through address* (i.e., the address immediately following the `call` instruction). Based on this assumption, recursive-traversal parsers assume that the bytes following a non-returning

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

(a)

Misused Call

`call <target>`
`...`
`.target`
`pop <register-name>`

(b)

Figure 2.2: 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.

`call` instruction represent a valid instruction sequence, and erroneously parse them as such. Second, the attack breaks an important assumption made by code parsers for identifying function boundaries, namely, that the target of a `call` instruction belongs to a different function than that of the `call` instruction itself. Finally, a binary instrumenter cannot move the `call` instruction of such a `call-pop` sequence without changing the PC-relative address that the pop stores into a general-purpose register. Moving the call instruction without compensating for this change usually results in incorrect program execution, as packer metacode frequently uses the PC-relative address as a base pointer from which to access data. On average, 7% of all `call` instructions used in packer metacode are non-standard uses of the instruction, and as seen in Table 2.1 of Section 2.4, most packer tools use this obfuscation.

```
push ADDR          push ADDR         pop  ebp
...                call .foo         inc  ebp
ret                ret               push ebp
                  .foo               ret
                   ret
```

(a) (b) (c)

Figure 2.3: Code sequences that tamper with the call stack. (a) and (b) are equivalent to `jmp ADDR`, while (c) shows a procedure that jumps to return-address + 1

Approaches 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. [62] 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 [68] and its code-finding techniques rely heavily on the presence of specific instruction sequences at function entry points; most obfuscated code does not exhibit such regularity. Though pure dynamic analysis approaches identify instructions accurately in the face of non-returning calls, they only find those instructions that execute in a given run of the program.

Call-stack tampering

While non-returning calls involve non-standard uses of the `call` instruction, call-stack tampering adds non-standard uses of the `ret` instruction as well. Figure 2.3 illustrates three call-stack tampering tricks used by the ASProtect packer. Figure 2.3a shows an obfuscated `push-ret` instruction sequence that is used as an equivalent to a `jmp ADDR` instruction. Figure 2.3b is a somewhat more complex instruction sequence that is also a `jmp ADDR` equivalent. Figure 2.3c shows a function that increments its return address by a single byte, causing the program to resume its execution at a location that recursive-traversal parsing would not detect without additional analysis.

Approaches To our knowledge, prior works have not used static analysis techniques to identify the targets of `ret` instructions, though this could improve the coverage and accuracy of disassemblers for obfuscated code. Dynamic analysis provides an alternative approach for resolving the targets of `ret` instructions (provided that they execute) and this approach is used by some dynamic instrumenters [14].

Obfuscated control-transfer targets

All of the packers in our study use indirect versions of the `call` and `jmp` instructions, which obfuscate control transfer targets by using register or memory values to determine their targets at run-time. Of course, even compiler-generated code contains indirect control transfers, but compilers use direct control transfers whenever possible (i.e., for transfers with a single statically known target) because of their faster execution times. One reason that packers often opt for indirect control transfers over their direct counterparts is that many packers compete for the bragging rights of producing the smallest packed binaries, and the IA-32 `call <register>` instruction is only 2 bytes long, while the direct `call <address>` instruc-

tion 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 [68]. 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 [5] 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 [24, 52, 128]. While dynamic analysis trivially discovers targets of indirect control transfers that execute, it may leave a significant fraction of the code un-executed and usually will not discover all targets of multi-way control transfer instructions. Researchers have addressed this weakness with techniques that force the program's execution down multiple execution paths [4, 76], but even these extremely

resource intensive techniques do not achieve perfect code coverage [110].

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 [93]. 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 [78]. This means that current disassembly algorithms will usually parse through fault-raising instructions into what may be non-code bytes that never execute. Another problem is that on Windows it is hard to find the exception handlers, since they can be registered on the call stack at run-time with no need to perform any Windows API or system calls. An additional difficulty is that the exception handler specifies the address at which the system should resume the program's execution, which constitutes yet another hidden control transfer.

Approaches Though static analyses will often fail to recognize exception-based control transfers, these transfers can be detected by two simple dynamic analysis techniques. The OS-provided debugger interface provides the most straightforward technique, as it informs the debugger process of any fault-raising instructions and identifies all registered exception handlers whenever a fault occurs. However, use of the debugger interface can be detected unless extensive precautions are taken [42], 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 re-registering its own handler.

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

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 [26]. 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 run-time. In some cases the transitions to these regions involve a control transfer instruction whose target is obviously invalid. For example, the last instruction of UPX bootstrap code jumps to an uninitialized memory region (refer back to Figure 2.1), an obvious indication that unpacking will occur

at run-time and that the target region should not be analyzed statically. However, binaries often contain data at control transfer targets and fall-through addresses that will be replaced by unpacked code at run-time. The most problematic transitions from code to junk bytes occur when the transition occurs in the middle of a straight-line sequence of code; ASProtect's seven sequentially 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. As to dealing with the ambiguity arising from transitions into regions that will contain dynamically unpacked code is to avoid the problem altogether by disassembling the code after the program has finished unpacking itself. However, this approach is not suitable for applications that use the analysis at run-time (e.g., for dynamic instrumentation), does not generalize to junk code detection at opaque branch targets, and does not capture overwritten code. Not capturing overwritten code is significant because

many packers overwrite or de-allocate code buffers immediately after use.

Code parsing techniques can cope with ambiguity by leveraging the fact that, though random bytes usually disassemble into valid x86 instructions, they do not share all of the characteristics of real code. Kruegel et al. build heuristics based on instruction probabilities and properties of normal control-flow graphs to distinguish between code and junk bytes [62]. Unfortunately, they designed their techniques for a specific obfuscator that operates exclusively on GCC-generated binaries [68] 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.

Disassembler fuzz testing

Fuzz testing [75] refers to the practice of stress testing a software component by feeding it large quantities of unusual inputs in the hope of detecting a case that the component handles incorrectly. IA-32 disassemblers are particularly vulnerable to fuzz testing owing to the complexity and sheer size of the instruction set, many of whose instructions are rarely used by conventional code. By fuzz testing binary-code disassemblers, packer tools can cause the disassembler to mis-parse instructions or mistake valid instructions for invalid ones. A simultaneous benefit of fuzz testing is that it may also reveal errors in tools that depend on the ability to correctly interpret instruction semantics (e.g., sandbox emulators and taint analyzers), which have a harder task than the disassembler's job of merely identifying the correct instructions. For example, one of the dark corners of the IA-32 instruction set involves the optional use of instruction prefixes that serve various purposes such as locking, segment-selection, and looping. Depending on the instruction to which these prefixes are applied, they may function as expected, be ignored, or cause a fault. The ASProtect packer does thorough fuzz testing with segment prefixes, while

Armadillo uses invalid lock prefixes as triggers for exception-based control flow. Other inputs used for fuzz testing include instructions that are rare in the context of malware binaries, and that might therefore be incorrectly disassembled or emulated (e.g., floating point instructions).

Various factors increase the thoroughness of disassembler fuzz testing. The polymorphic approach whereby PolyEnE and other packers generate bootstrap code allows them to create new variations of equivalent sequences of rare instructions each time they pack a binary. In the most aggressive uses of polymorphic code, the packed binary itself carries the polymorphism engine, so that each execution of the packed binary fuzz tests analysis tools with different permutations of rare instructions (e.g., ASProtect, EXEcryptor). Finally, packers include truly random fuzz testing by tricking disassembly algorithms into mis-identifying junk bytes as code, as we have discussed in Section 2.2. 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 [19], Ida Pro [52], ParseAPI [87]) 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 [97], Qemu [8], TSL [66]).

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 [57].

Function-boundary-detection techniques should also not expect the compiler to lay out a function's basic blocks in an obvious way, viz., in a contiguous range in the binary with the function's entry block preceding all other basic blocks. A common reason for which compilers break this assumption is to share basic blocks between multiple functions. The most common block-sharing scenarios are functions with optional setup code at their entry points and functions that share epilogues at their exit points. An example of optional setup code is provided by Linux's libc, several of whose exported functions have two entry points; the first entry point is called by external libraries and sets a mutex before merging with the code

at the second entry point, which is called from functions that are internal to libc and have already acquired the mutex. Meanwhile, some compilers share function epilogues that perform the task of restoring register values that have been saved on the call stack. As we will see in this section, function boundary identification techniques for obfuscated code must confront the aforementioned challenges as well as further violations of `call` and `ret` usage conventions, scrambled function layouts, and extensive code sharing between functions.

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 2.3.

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 [63]. The opposite and more prevalent problem of superfluous uses of the `call` instruction has not been addressed by prior works, except in the context of thunk detection for compiler-generated code [57].

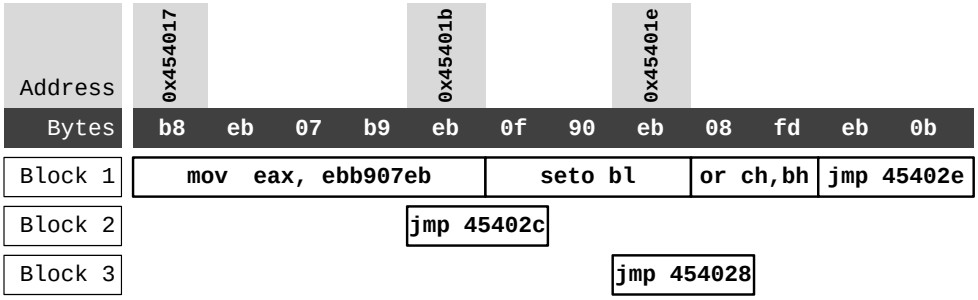


Figure 2.4: An example of overlapping instructions and basic blocks taken from the obfuscated bootstrap code of a binary packed by Armadillo. All three basic blocks actually execute.

Overlapping functions & basic blocks

Of the 12 packer tools we selected for this study, 7 share code between functions. As in compiler-generated code, the most common use of code sharing is for optional function preambles and epilogues. Some packed binaries achieve tiny bootstrap code sizes by outlining short instruction sequences (i.e., moving them out of the main line of the function's code) into mini-functions and calling into various places in these functions to access only those instructions that are needed. Outlining code in this way results in a lot of code sharing and strange function layouts; for example, the function's entry block is often at a larger address than 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 2.4). 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 [79, 123] 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 [52]. 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.). 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 [79, 123]. Since multiple blocks and instructions may indeed map to any given address (see Figure 2.4), tool data structures should be designed to account for this possibility.

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 [52] or OllyDbg [128], 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

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

Not obfuscated	Obfuscated

`mov ebp, -0xab7`	`call <next>`
	`pop ebp`
	`sub ebp, 0x40e207`

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

Figure 2.5: 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.

understand when it is structured the way it is written, viz., in functions with well-defined interactions.

The remainder of this section presents additional obfuscations that impact code-comprehension. We begin with machine-language instructions whose constant operands are obfuscated, then discuss calling-convention violations, and conclude by describing the role of do-nothing code in obfuscated programs.

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 boot-

strap 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 2.5 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 [55, 122]. 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 [15]. This approach produces programming language statements from which constant 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 anti-unpacking techniques that we will discuss in Section 2.2, there are general-purpose unpacking tools that may be able to automate the reverse-engineering task [13, 127]. However, for binaries that do employ anti-unpacking, the analyst must reverse-engineer the binary by hand and must therefore manually de-obfuscate uses of the OEP address. In these situations, analysts de-obfuscate 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.

Calling-convention violations

Calling conventions standardize the contract between caller functions and their callees by specifying such things as where where to store function parameters and return values, which of the caller/callee is responsible for clearing parameter values from the call stack, and which of the architecture's general-purpose registers can be overwritten by a called function. Though there are many calling conventions for the x86 platform [46], 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 x86 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 [54, 62] and instrumentation efficiency [57], 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 [73], DynamoRIO [14], PIN [70]).

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 [22]. 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, multi-layer packing, and anti-patching (see Section 2.2).

Do-nothing code that never executes can sometimes be eliminated by the malware normalization techniques that Bruschi et al. built into the Boomerang decompiler [15]. 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 2.2.

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 [127] or system calls [105]) 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.

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 [7, 58, 124, 127]. 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 x86 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 [57, 58]. 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 x86 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.

Stolen bytes

Figure 2.6b 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 [57, 58, 127]. 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 run-time, after the shared library has been loaded and calls to the imported function have been linked through the Import Address Table. This technique must therefore modify 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 [124], Detours [58], TTAnalyze [7]) 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

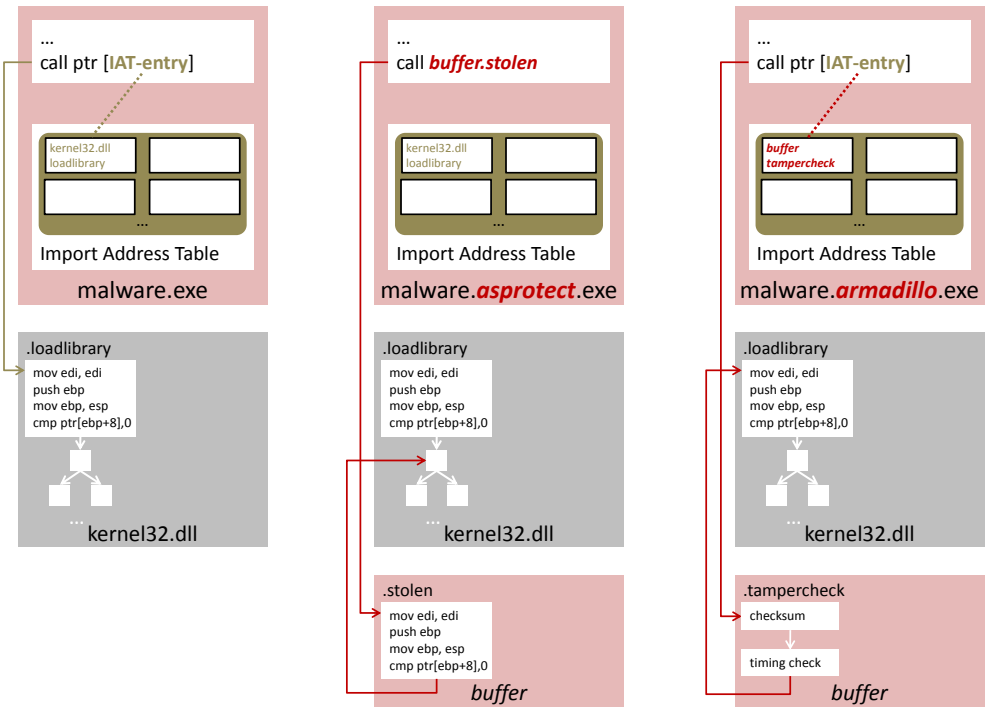


Figure 2.6: Packed programs may interpose on inter-library calls originating from their packed payload to resist reverse-engineering and code-patching techniques. In a non-packed program, as in part (a), calls to functions in shared libraries use indirect call instructions to read their targets from Import Address Table entries. Part (b) illustrates ASProtect's implementation of the stolen-bytes technique, which evades binary instrumentation at the entry points of functions in shared libraries by making a copy of the first block of an imported function and redirecting the call instruction to point at the "stolen" block. Part (c) illustrates Armadillo's method of hooking inter-library calls by replacing IAT addresses with the address of an Armadillo function that performs anti-tampering techniques before forwarding the call on to its intended destination.

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.

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 [73], DynamoRIO [14]). 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 [11]. 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 [8] provides the foundation for several malware analysis tools [7, 21, 76]). Whole-system monitoring has also been achieved with tools that extend virtual machine hypervisors [37]. Though it is not possible to monitor malware's execution in a provably undetectable way [37], 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 [89], which originated as the XenAccess project [90], perform the necessary introspection

techniques for 32-bit Windows and Linux guest systems, but may not be compatible with other monitoring tools.

Self-checksumming

Many packed binaries verify that their code has not been tampered with by applying self-checksumming techniques [3]. These packers take a checksum over the program's code bytes and then recalculate that checksum at run-time 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. [125]) 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 [100]. This shadow-memory-based approach becomes more difficult for programs that unpack or overwrite their code at run-time, however, as the dynamically written code bytes must be present in execution-context memory so that they can execute [48].

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 [13, 127], 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 2.2 we discussed defensive techniques that attack the code extrac-

tion task and in this section will focus on techniques that make packer metacode difficult to bypass. We describe the anti-unpacking techniques used by the 12 tools of this study; for a broader discussion of possible anti-unpacking techniques we refer the reader to Peter Ferrie's surveys on the topic [44, 45].

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 run-time, and in some cases (e.g., ASProtect) the code leading up to the control transfer is polymorphic and unpacked to a different address on each execution of the same packed binary. To counter known cipher-text attacks based on 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 [12, 117, 126]. 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 [17, 18]. 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 [28, 35]. The sheer number of unique packing approaches means that their techniques are probably not well-suited for anti-virus 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 [53] and OmniPack unpackers [74] track writes at a memory-page granularity, and detect thousands of false-positive unpacking instances for packers that place code and writable data on the same memory pages. Pandora's Bochs [13], 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.

Payload-code modification

The second challenge of anti-reverse-engineering is preventing the analyst from bypassing the defensive metacode that the packer tool places in the 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 2.6c. 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 2.6b. To make matters worse, AS-Protect 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 [37, 51, 60], create unpacked binaries that are amenable to static analysis but cannot actually execute [22, 127], or work

only on a subset of packed binaries [13, 28, 35].

2.3 Obfuscation Statistics

We now proceed to quantify the prevalence 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 [18], are most-often used to obfuscate real-world malware. We used Dyninst, our binary analysis and instrumentation tool [57, 103], 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 2.1. 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 2.1 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

Table 2.1: 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	4.8	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	2	4	5	3
	A2 Overwrite instances	0	0	1	2	0	0	4	0	1	28	10	0
	A3 Overwrite byte count	0	0	64	167	0	0	179	0	194	1126	9889	0
	A4 Overwrite of executing function	0	0	1	0	0	0	0	0	0	12	4	0
Obfuscated Instructions	B1 % of calls that are indirect	83%	75%	95%	44%	11%	28%	30%	92%	18%	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	2	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	0	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	16	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	2	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	2	0	89	0
	E3 Metacode uses WinAPI funcs not in IAT	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×
	E4 Timing check	×	×	×	×	×	×	×	×	×	×	✓	×
	E5 Anti-Debug: via WinAPI calls	×	×	×	×	×	×	×	×	×	✓	✓	×

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

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 [72, 103].

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 2.1, 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 non-standard ways to obfuscate their control flow (row B3). `Call` instructions are misused even more frequently, in which case they are usually paired 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 x86 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 [62, 79]. 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 x86 compilers never do, even in highly optimized code (row C9). Similarly, compilers avoid placing writable data on code pages, since this has dangerous security implications, whereas obfuscated pro-

grams 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 [26], or more often, by causing control flow to fall through into bytes that will be decrypted in place at runtime (row C11).

Anti-Rewriting: Analysts frequently try to rewrite packed binaries to create versions that are fully unpacked and bypass the obfuscated bootstrap code. One way in which packed binaries make this task challenging is by placing code in strange places, such as the Portable Executable file header and in buffers that are created by calls to the VirtualAlloc Windows API function. 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).

Anti-Tracing: Tracing techniques are commonly used to study the

behavior of obfuscated programs, and may be collected at different granularities, ranging from instruction-level traces to traces of Windows API calls, and system calls. Some packers 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). For normal binaries, analysts look at the Import Address Table data structure to determine which Windows API functions are used by the program [58], but most malware binaries circumvent the IAT for many of their Windows API calls (row E2), 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 2.1 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 2.1, 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 [95].

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 non-essential 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 writable 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 de-allocates 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 of `call` and `ret` instructions for non-standard purposes, and overwrites an instruction operand at run-time to modify a control flow target.

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

for the purposes of binary obfuscation [31], 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 anti-disassembler obfuscations. For example, ASProtect causes the program to parse into ASProtect-packed binaries 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.

2.4 Summary

As most code obfuscations specifically target static analysis techniques, it is not surprising that most previous works include strong dynamic analysis components. The few purely static techniques that have been developed for obfuscated code analysis have focused on countering the effects of individual obfuscators [15, 62, 63] and do not account for the breadth of obfuscation techniques that are often present in a single malware sample.

Most dynamic techniques designed for malware have modest goals with regards to structural code analysis, focusing instead on instruction-level instrumentation [37, 70], or on simply capturing the raw bytes in dynamic code regions for delivery to an anti-virus tool [53, 74], to a reverse-engineering process that produces an unpacked binary [13, 51, 127], or to a third-party analysis tool [38, 60].

Of the prior works with dynamic analysis components that do provide structural code analysis, some do not handle code unpacking and code overwriting [71], while tools that do build structural analyses do so only

after the code has finished executing [1, 39, 127]. Since prior works do not provide structural analysis of obfuscated code until after its execution, dynamic tools have not been able to leverage the rich structural analysis products that are available when analyzing non-defensive binaries.

3 STATIC ANALYSIS

The purpose of our static parsing techniques is to accurately identify binary code and analyze its structure. Like most other parsing algorithms, our techniques produce an interprocedural control flow graph of the program, organized into basic blocks and functions. However, most parsing techniques for obfuscated code have been tailored to specific obfuscations or obfuscators rather than building approaches for arbitrarily obfuscated code [62, 63, 120]. The two primary measures of quality of a static parsing technique are that it identifies binary code with good accuracy and coverage. We prioritize accurate code identification because an incorrect parse can cause incorrect program behavior by leading analysis tools to instrument non-code bytes, and is ultimately not very useful to an analyst. The competing goal of achieving good code coverage through parsing is less of a priority for us because our dynamic techniques help compensate for lapses in coverage by capturing statically unreachable code at run-time.

A secondary measure of quality of a parsing technique is whether it accurately identifies function boundaries. Accurate function-boundary identification is challenging in obfuscated code, both because `call` and `ret` instructions are sometimes not used at function entry and exit boundaries, and because these instructions are sometimes used at non-function boundaries in place of `jmp` instructions. In this chapter, we present our accurate parsing and function-labeling algorithms, which are based on a combination of recursive control-flow traversal parsing and binary slicing techniques.

3.1 Accurate Code Parsing

Recursive control-flow traversal parsing [112, 118] is the basis for most accurate parsing techniques, but it makes four unsafe assumptions about

control flow that can reduce its accuracy (see Section 2.2). First, it assumes that function-call sites are always followed by valid code sequences. Compilers violate this assumption when generating calls to functions that they know to be non-returning, while obfuscated programs (e.g., Storm Worm [94]) often contain functions that return to unexpected locations by tampering with the call stack. Second, the algorithm assumes that both targets of a conditional branch instruction can be taken and therefore contain valid code. Obfuscated programs can exploit this assumption by creating branches with targets that are never taken, thereby diluting the analysis with junk code that will never execute [26]. Third, the algorithm assumes that control flow is only redirected by control transfer instructions. Obfuscated programs can use an apparently normal instruction to raise a signal or exception, thereby transfering control to code that is hidden in a signal or exception handler. The handler can further obfuscate control flow by telling the operating system to resume execution away from the signal- or exception-raising instruction (e.g., Yoda's Protector [30, 31]), potentially causing junk code to be parsed following the instruction [93]. Fourth, the algorithm assumes that the code in the program binary is statically present. Occasionally this problem is easily recognized, as when the UPX packer uses a `jmp` instruction to transfer control into a section of the binary that is not initialized. More often, however, a packed programs will decrypt code in place and have control-flow fall through into dynamically unpacked code regions without executing a control-transfer instruction (e.g., ASProtect).

In our experience with analysis-resistant binaries, we have found that by far the most targeted vulnerability is the assumption that code follows each `call` instruction. Because of the prevalence of non-returning calls and call-stack tampering in obfuscated code, we use binary slicing techniques to help us determine whether call instructions will return. The problem of branches with targets that never execute is a daunting one because of

the ubiquity of the conditional branch instruction, but we have not found this to be a prevalent problem in packed malware. Even harder to detect with static techniques are the problems of signal- and exception-based obfuscations and fall-through transitions into unpacked code, as these require analysis at every instruction. We also develop lightweight static techniques that eliminate much of the junk code that would otherwise be parsed because of these obfuscations. After application of our parsing techniques, most of the remaining errors in the disassembly are due to signal- and exception-based obfuscations and in-place code-unpacking, and are detected and resolved by our dynamic techniques at run-time. We proceed by discussing our techniques for dealing with non-returning calls, followed by our additional junk-code avoidance techniques. Our dynamic techniques are presented in Chapters 4, 5, and 6.

Non-returning calls

When a called function either never returns or returns to an unexpected location by tampering with the call stack [68], one or more junk bytes may follow `call` instructions at call sites to that function. The simplest approach to dealing with these non-returning calls would be to adopt the assumption taken by Kruegel et al.'s obfuscated code parser [62], that function calls never return, and rely on run-time monitoring of return instructions to discover code that follows call sites (Kruegel et al. used static compensation techniques). It is preferable, however, to detect non-returning calls statically as this increases overall code coverage by correctly analyzing code at call fall-through addresses that might not execute when we run the program. Thus, we develop static techniques to detect call-stack tampering, but retain run-time monitoring of return instructions as our technique of last resort.

We take advantage of the depth-first nature of our parsing algorithm to use the analysis of called functions in deciding whether or not to continue

```
push ADDR          pop  ebp
...                inc  ebp
retn               push ebp
                   retn
 (a)                (b)
```

Figure 3.1: Code sequences that tamper with the function's return value on the call stack

parsing after `call` instructions. If our analysis of the called function contains no paths that end in a return instruction, we do not resume parsing after call sites to the function. If the called function does contain return instructions, we perform static call stack analysis [63, 69, 108] on the called function to determine if it tampers with the stack or stack pointer to change its return address.

Our stack analysis is based on binary slicing techniques [23], which are program slices, as adapted for machine-language instructions. A binary slice identifies instructions that influence the value that is assigned to a particular register or memory location. We address call-stack tampering by using binary slices to determine the control-flow target of each of a called function's `ret` instructions. More precisely, we perform an intra-procedural backwards binary slice on the call-stack input of each `ret` instruction, which delivers a graph of instructions that together define the value read by the `ret` instruction (see Bernat et al. for details on the slicing techniques that we use [11]). For a normal function that does not tamper with its call stack, the binary slice shows that its `ret` instructions will read stack inputs whose values depend only on the `call` instruction that invoked the function. However, for a function that does tamper with its call stack, the binary slice delivers a graph of instructions from which we must extract the `ret` instruction's target. In practice, obfuscated `ret` targets are usually constant addresses, and when they are not, we have found that they can be specified by a linear formula based on the caller-specified

return address. We therefore apply symbolic evaluation to simplify the graph down to a constant or linear formula. As each node in the graph represents an operation performed by an instruction in the function, this involves an ordered traversal of the graph that begins at input nodes and iteratively reduces operations that consume constant or linear inputs down to constants or linear formulas. When our symbolic evaluation fails to simplify an operation (e.g., because it performs a non-linear operation on the caller-specified return address) it produces an inconclusive result. This can also happen when we try to perform a stack analysis for a called function that is not fully statically analyzable, typically because it contains unresolved indirect control flow. In these cases, we mark the function as possibly non-returning, and disallow parsing at the fall-through addresses of all `call` instructions that invoke the function. Fortunately, we are usually able to compensate for these lapses in code coverage with the dynamic code-discovery techniques that we present in the next chapter.

Our call-stack analysis detects the actual target of many stack-tampering techniques that alter normal return addresses; two simple examples are shown in Figure 3.1. Figure 3.1a shows an instruction sequence that transfers control to `ADDR` upon executing the return instruction. Figure 3.1b shows a sequence used by the ASPack and ASProtect packers [2] to increment the return address of a function by a single byte. In both cases, our call-stack analysis informs the parser of the actual return address of the called function. In the former case, the binary slice and symbolic evaluation deliver `ADDR` as the target of the program's return instruction, while in the latter case, they correctly identify the return target as one greater than the fall-through address of the calling instruction. Our parser uses these target addresses to seed additional parsing in place of the fall-through addresses of call sites to the function.

Additional junk code avoidance

We use a lightweight analysis to compensate for other obfuscations that can cause the recursive-traversal algorithm to parse junk code. We were able to use expensive binary slicing techniques to deal with non-returning calls because these slices only needed to be applied on a per-function basis. Regrettably, we cannot afford to apply such a heavyweight analysis approach to detect junk code at targets of all conditional branch instructions, as this instruction is widely used; in our study of the obfuscated code used by prevalent packer tools, conditional branch instructions appeared in 1 of every 12 instructions. It would be even more expensive to apply this approach to detect junk after exception-raising instructions and at fall-through transitions to unpacked code regions, since we would need to check for these transitions after every instruction in the program.

Our experience with packed binaries has taught us that most junk code consists of non-code bytes such as data, in-place encrypted instructions, ASCII strings, and padding bytes. Regrettably, the density of the IA-32 instruction set is such that random byte patterns almost always represent valid instructions. Fortunately, junk bytes do not share all of the same characteristics as compiler-generated code, and generally exhibit the following properties:

1. Junk code contains numerous instructions that are extremely rare in normal or obfuscated code, sometimes because they can only be executed in a privileged context. Many of these rare instructions have one-byte opcodes that occur frequently in junk code.

2. Junk instructions that perform load and store operations have arbitrary memory addresses as operands and most of these instructions would result in memory access violations if they were to execute.

3. Junk code based on random bytes contains about as many jump and branch instructions as normal code because these instructions take

up a large percentage of the IA-32 instruction-set's opcode space. However, the targets of junk branches are far more likely to be unaligned with the program's actual instructions, producing more overlapping basic blocks than there are in normal obfuscated code.

4. Junk instructions often use source and destination registers of unusual sizes.

5. Constant operands in junk instructions are random and are usually larger than the constants used by normal instructions.

We have found the first two properties of junk code to be the most reliable indicators of junk code. Techniques for code parsing have leveraged the third, fourth, and fifth characteristics of junk code to disambiguate between junk and compiler-generated code [52, 102, 107, 120] and between junk and obfuscated code [62]. However, other researchers have not developed techniques to disambiguate between junk and arbitrarily obfuscated code. One significant problem is that obfuscated code also contains the third, fourth, and fifth characteristics of junk code in some measure. The third property also holds for some obfuscated code, and when two basic blocks overlap, it is not clear whether both blocks are valid, both invalid, or whether one is valid and the other invalid. The fourth property is also true of packer bootstrap code, which frequently decrypts code at a byte level, often using IA-32's compact string operation instructions to do so. The fifth property of junk code is also a characteristic of instruction-constant obfuscations, which are prevalent in malware code. This leaves the first two properties of junk code as the most reliable disambiguators between junk code and obfuscated code. However, while rare and privileged instructions are inexpensively identifiable, determining which instructions will raise a signal or exception is a difficult problem, both in theory and in practice [78].

We construct a blacklist of instructions that are either privileged or exceedingly rare in normal code. When our parser encounters a privileged or rare instruction, the parser ends the current basic block after that instruction, suspends parsing, and adds the block to a list of basic blocks that end abruptly. This approach eliminates all junk code after rare instructions. The following are the most prevalent instructions that are indicative of a bad parse:

- `0x6300 arpl *[eax],ax`: The `arpl` instruction is privileged and causes a fault if executed by user-mode code. Its one-byte opcode also corresponds to the ASCII code for 'c', and since junk bytes often consist of string data, the `arpl` instruction occurs more frequently than at the expected 1/256 occurrence rate. The second byte of the instruction contains the operand and can take any value.

- `0x06 push eb`: This is one of four single-byte push instructions that push segment registers on the stack. Not only are these instructions exceedingly rare in normal code, we have not seen them in any obfuscated code. In junk code, however, they occur even more frequently than at the expected 1/64 rate, since they are single-byte instructions whose opcodes are small numbers: 0x06, 0x0e, 0x16, and 0x1e. By observation, small numbers are more prevalent in junk bytes; we can hypothesize about why this is so, but do not have principled reasons to explain it. Two-byte `push`'s and `mov`'s that target segment registers are similarly rare in normal code and we also use them to indicate a bad parse.

- `0x0000 add *[eax],ax`: This is the most common instruction that indicates a bad parse, as the Windows loader initializes all memory bytes to zero. Furthermore, some compilers use this instruction as padding between basic blocks.

Though our black-listing technique does not identify binary code with perfect exclusion of junk code, there are enough rare single-byte opcodes in the IA-32 instruction set that we can use them to eliminate the majority of junk code with extremely low false-positive rates. When false-positive junk code identification does occur, we are prepare to resolve the error at run-time by instrumenting after rare instructions. If the rare instruction actually does execute, the instrumentation will trigger additional parsing through the dynamic capture techniques that we present in Chapter 4. In most cases, however, the rare instruction triggers a fault, causing exception-based control flow that does not return to the subsequent instruction. Our parsing algorithm has done the right thing in these situations by excluding the subsequent instructions.

Our junk-code avoidance technique is well-suited to our purposes because it works well in practice on malware binaries. When evaluating this technique against the obfuscated bootstrap code of prevalent packer tools, our techniques stop parsing within 17 bytes on average after a transition into a junk region and within 93 bytes in the worst case. Furthermore, we are able to remove most of the remaining junk instructions from our disassembly at run-time; we remove junk after signal- or exception-raising instructions when they execute and reveal the succeeding bytes to be unreachable, and we remove junk bytes corresponding to in-place encrypted code bytes when those bytes are decrypted at run-time, revealing dynamically unpacked code. These techniques limit parsed junk code to the extent that it is highly unlikely to overwhelm a human analyst. Furthermore, we have designed our instrumentation techniques to tolerate disassembly errors. Even when our parse induces us to instrument junk bytes that the program uses as data, this has no impact on the program's behavior because we use techniques that prevent the program from reading from patched code and data (see Chapter 5 for a description of our safe instrumentation techniques).

There are no static parsing techniques that achieve perfect exclusion of junk code, but since our black-listing technique detects transitions into junk code within an average of 17 bytes, this technique would serve as a starting point for more accurate junk-detection by focusing heavyweight techniques on the parsed bytes preceding any rare instructions.

3.2 Accurate Function Identification

As described in Section 2.2, there are two obfuscations that can make it difficult to accurately detect the boundaries between functions in binary code. First, replacing `call` and `ret` instructions with equivalent instruction sequences creates the illusion of fewer functions than the program actually contains. Second, `call/ret` instructions can be used out of context to create the illusion of additional functions. The first of these obfuscations was addressed by Lakhotia et al. [63], but we have not seen it occur with any more regularity in obfuscated code than in optimized code. Analysis tools built for optimized code also have to deal with missing `call` and `ret` instructions because optimizing compilers implement tail calls with a `jmp <target>` instruction instead of `call <target> ; ret` instruction pairs. We chose to adopt the simple heuristic techniques by which analysis tools for optimized code deal with missing `call/ret` instructions [16] because tail-call optimizations are also common in obfuscated code, and these heuristics work well in practice on the obfuscated binaries we have studied. We therefore focus on providing solutions for the opposite and more prevalent problems of superfluous uses of the `call` and `ret` instructions.

We define *superfluous call instructions* to be `call` instructions that are used to transfer control to a target and do not use the return address that the `call` pushes onto the stack to influence the target address of subsequent return instructions. By this definition, a `call` targeting code that pops the return address from the stack is superfluous if subsequent

```
mov edx,ecx          jmp .trg          mov esp, esp+4
pop edi              ...               ...
push eax             .trg
pop esi              pop edi
...                  ...
 (a)                  (b)               (c)
```

Figure 3.2: Code sequences taken from the targets of `call` instructions in the ASProtect packer's obfuscated code that remove the return address from the stack. These `calls` are effectively being used as `jmps`, and the return address is either thrown away or used as a base address for subsequent memory access operations.

`ret` instructions return to a location that is higher up on the call stack. In Figure 3.2 we show examples of superfluous calls taken from the ASProtect packer. We have found that the code at superfluous call targets usually removes the return address from the stack within a few instructions, but as seen in part (b), this does not always happen in the first basic block, which often consists of a single `jmp` instruction. As shown in part (c), the return address may be removed by a non-`pop` instruction.

We approach superfluous call-instruction identification by using *look-ahead parsing* techniques at each call target. Look-ahead parsing techniques allow our parser to perform exploratory parsing at call targets prior updating our call- and control-flow-graph data structures by adding new function and basic block objects for the code at the call target. To detect the different uses of superfluous calls shown in Figure 3.2, we perform lookahead parsing through the first multi-instruction basic block at each call target. We apply symbolic evaluation over these instructions to identify whether the program removes the return address from the call stack. Though this is not a rigorous solution to the problem of superfluous call identification, it works well in practice for the obfuscated code we have studied. Some error is acceptable in function-boundary identification as it

serves mostly to assist a human analyst in comprehending the program and does not impact the correctness of most instrumenters and analysis tools.

We define *superfluous return instructions* to be `ret` instructions that do not transfer control flow to the fall-through address of `call` instructions that invoke the `ret's` function. We detect superflous `ret` instructions through the call-stack tampering techniques that we presented in the previous section; `ret` instructions always target call fall-through addresses in the absence of stack tampering. Our call-stack tampering analysis provides the only static techniques we are aware of that attempt to detect `ret` targets; static analyses that are built for compiler-generated code do not need to worry about non-standard uses of this instruction. The primary benefit of applying static techniques to `ret` target prediction is that they reveal addresses at which to seed further parsing, increasing our parser's code coverage. We still treat superfluous `ret` instructions as function exit points and their targets as new function entries, since we have not seen instances of code-sharing between the function containing a superfluous `ret` instruction and the code at its target. Furthermore, we have observed that most instances of superfluous `ret` instructions transfer control across sections of the binary or across code buffers, often to the original entry point of a packed program binary.

4 DYNAMIC CODE DISCOVERY TECHNIQUES

Having found and analyzed as much of the code as possible by traversing the program's statically analyzable control flow, we turn to dynamic analysis techniques to find code that is not statically analyzable, and to correct any errors in our static analysis. Statically un-analyzable code includes code that is originally present in the binary but is reachable only through un-analyzable pointer-based address calculations or exception-based control flow, and code that was not initially present because it is dynamically generated, sometimes in place of existing code. We analyze this code at run-time through a collection of dynamic techniques that satisfy the following two constraints. First, our dynamic techniques identify entry points into un-analyzed code before it executes and expand our analysis by seeding parsing at these entry points. Second, our techniques identify and remove any dead code from the program's control-flow graph (e.g., junk code after a signal- or exception-raising instruction, or overwritten code). Since our analysis of the program changes at run-time, these constraints require that our data structures for the binary code be expandable and contractable.

We introduce dynamic techniques that resolve the full breadth of prevalent obfuscations currently used by malware and develop new techniques to remove dead and unreachable code from the analysis when it is overwritten. These techniques support our overall goal of providing analysis-guided instrumentation on malware, by keeping our analysis up to date as the program's execution reveals new code and changes existing code. The Bird instrumenter comes closest to our dynamic techniques in its use of dynamic instrumentation to discover missing code at indirect control transfer targets, but they do not build an analysis representation that is available to users of their tool, and they make several simplifying assumptions that limit their instrumenter's applicability to defensive malware code [79]. We

also build on the code overwrite detection and signal-handler discovery techniques used by Diota [72], adapting these techniques for use on malware, and adding techniques to update our analysis in response to these events.

We begin by describing the instrumentation techniques by which we expand our analysis to cover statically hidden and dynamically unpacked code. We then discuss our techniques for detecting and responding to code overwrites. We conclude this chapter with a description of the dynamic techniques that enable us to find and analyze exception-based control flow.

4.1 Instrumentation-Based Code Capture

During parsing, we mark all program locations that could transition from analyzed code to un-analyzed code. We dynamically instrument these program locations so that we will detect all transitions into un-analyzed code before that code executes. Our dynamic capture instrumentation monitors the execution of instructions that meet any one of the following criteria:

1. *Control transfer instructions that use registers or memory values to determine their targets.* Indirect jump and call instructions are often used by obfuscated programs to hide code from static analysis. The FSG packer [47] is a good example, with one indirect function call for every 16 bytes of bootstrap code. In the case of indirect call instructions, when our parser cannot determine a call's target address, it also cannot know if the call will return, so it conservatively assumes that it does not. We determine whether indirect control transfers leave analyzed code by resolving their targets at run-time with dynamic instrumentation. For indirect call instructions, resolving the call target also allows us to parse after the call site if we can determine that the called function returns.

2. *Return instructions of possibly non-returning functions.* Return instructions are designed to transfer execution from a called function to the caller at the instruction immediately following the call site; unfortunately they can be misused by tampering with the call stack. As detailed in Section 3.1, our parser is usually able to determine the targets of the return instructions of called functions, in which case we continue parsing after call sites to those functions. When our parser's analysis is inconclusive for a particular function, we instrument its return instructions to determine their targets.

3. *Control transfer instructions into invalid or uninitialized memory regions.* Control transfers to dynamically generated code can appear this way because packed binaries often unpack code into uninitialized or dynamically allocated memory regions (e.g., binaries created by the UPX packer [85]). Our instrumentation of these control transfer instructions executes immediately prior to the control transfer into the region, by which time the region should contain valid code, allowing us to analyze it before it executes.

4. *Instructions that terminate a code sequence by reaching the end of initialized memory.* Some packer tools (e.g., Upack [40]) and custom-packed malware (e.g., the Rustock rootkit [20]) transition into dynamically generated code without executing a control transfer instruction. Most binary executable formats allow code to be mapped into larger memory regions, resulting in the possibility of a valid code sequence that runs all the way up to the end of initialized memory without a terminating control transfer instruction. Packers that use this technique unpack the remainder of the function body into the region's uninitialized memory so that when the function is invoked, control flow falls through into the unpacked code. We detect this scenario by instrumenting the last instruction of code sequences that end abruptly

without a final control transfer instruction. When this instruction executes, we trigger analysis of the following instructions.

5. *Rare instructions* We terminate parsing after rare and privileged instructions that are uncommon in normal and obfuscated code, under the presumption that our parse has veered into junk bytes (see Section 3.1). Since it is possible that the rare instructions are legitimately used by the program, we instrument immediately after all rare instructions so that if they do execute, we can trigger pre-execution parsing of the subsequent instructions.

Our dynamic capture instrumentation supplies us with entry points into un-analyzed code that submit to our code parser. Before extending our analysis by parsing from these new entry points, we determine whether the entry points represent un-analyzed functions or are extensions to the body of previously analyzed functions. We treat call targets as new functions, and treat branch targets as extensions to existing functions. The targets of non-obfuscated return instructions always are immediately preceded by an analyzed call instruction, in which case we parse the return instruction's target as an extension of the calling function. When a return target is not immediately preceded by a call instruction, we conclude that the call stack has been manipulated and parse the return instruction's target as a new function.

Our dynamic capture techniques are independent of the underlying instrumentation mechanism, but cost issues arise from our choice to implement these techniques in the Dyninst instrumentation library [16]. Dyninst controls programs from a separate process that contains the analysis and instrumentation engine. The cost problem arises because our code-discovery instrumentation executes in the monitored process and must context switch to the Dyninst process to determine whether a control-transfer target corresponds to new or to previously analyzed code. For

transfers to previously analyzed target addresses, the Dyninst process does not need to perform analysis updates, so it context switches back to the instrumented process without performing any work. We eliminate these two context switches for analyzed control transfer targets by caching them in the address space of the instrumented process. We adjust our dynamic capture instrumentation to look up control transfer targets in this cache and make the context switch to Dyninst only for target addresses that have not been seen before.

4.2 Response to Overwritten Code

Code overwrites cause problems for analysis by simultaneously invalidating portions of the control flow graph and introducing new code that has yet to be analyzed. Most analysis tools cannot analyze overwritten code because they assume the code to be static and do not update their code representations as the program overwrites itself. We have developed techniques to address code overwrites by using dynamic techniques to detect the overwrites and to update our CFG representation of the program before the overwritten code executes.

Our motivating goal of providing pre-execution analysis of overwritten code grants us some flexibility in terms of when we update our CFG of the program in response to an overwrite. The earliest possible response to a code overwrite is immediately after the execution of the write instruction that modifies the program's code, while the latest possible response is to delay the update until just before the execution of an overwritten instruction. To choose efficient detection and analysis-update approaches, it is important to understand how code overwrites work in practice. As shown in Table 2.1 of Section 2.3, malware employs several tactics to make overwrite remediation a challenging problem. The primary challenges we face in developing techniques to respond to code-overwrites are the

following:

1. Ranges of consecutively overwritten code bytes are nearly always overwritten one byte at a time, usually in a loop, but sometimes by a `rep`-prefixed `mov` instruction. In binaries created by prevalent packer tools, these overwritten ranges vary in size from 1 byte to over 8 kilobytes. Since updating our control-flow graph of the program is a resource-intensive operation, it is imperative that we batch CFG updates for incremental overwrites.

2. Obfuscated code frequently writes to data on code pages. For example, binaries packed by MEW do not overwrite any of their code but perform eight writes to code pages for every byte of payload code that they unpack. These writes cause false-positive code-overwrite detections for a widely used detection mechanism that works by removing write permissions from code pages and handling the access-rights violations that occur when the program writes to those pages. [53, 73, 74, 98].

3. Obfuscated code frequently writes to code and data on its own memory page. This practice makes it yet harder to respond to accurately detect code overwrites by manipulating memory-page access rights. Researchers have detected code overwrites by identifying code pages to which the program writes and from which it subsequently executes by toggling the write and NX bits for those pages so that the write and subsequent execute events cause access violations that their tools can handle [53, 74]. Unfortunately, binaries that write to the page from which code is currently executing cause thousands of false-positive code-overwrite detections.

The first problem we must solve in formulating an efficient response to the code overwrite problem is how to detect code overwrites. PolyUnpack used an approach to detect binary unpacking that also works for

code overwrite detection, and works by single-stepping the program and checking instructions immediately prior to their execution to see if they have been modified [104]. This single-step approach has the benefit that it delays our CFG updates until the last possible moment, but also the downside that single-step execution is extremely slow. Alternatively, we could detect code overwrites as soon as they occur by monitoring write instructions; independently of how we implement this technique, it is inherently more efficient than PolyUnpack's approach, as write instructions constitute a small fraction of the overall number of executed instructions, and PolyUnpack checks them all. The downside to write-based overwrite detection is that it does not batch incremental code overwrites, but we can use it as an efficient detection strategy and use additional techniques to trigger the corresponding analysis update at a later time.

Our solution to code-overwrite detection adapts Diota's write monitoring techniques for our purposes [72, 73]. Diota monitors writes to all memory pages that are both writable and executable by removing write permissions from those pages, thereby causing writes that might modify code to raise an access-rights violation that Diota intercepts. As illustrated in Figure 4.1a, we have adapted this mechanism to be more efficient on packed binaries, which typically mark most of their memory as writable and executable, by removing write permissions only from memory pages that contain analyzed code.

The next challenge we face is identifying an appropriate time at which to update our analysis. The naïve approach shown in Figure 4.1b fails to handle incremental code overwrites efficiently because it updates the analysis each time a code byte is overwritten. Instead, we detect the first write to a code page but allow subsequent writes, leveraging our pre-execution analysis of the program to analyze the overwrite loop and delaying analysis updates until the loop is done executing. This delayed-update approach divides our response to code overwrites into two components that we now

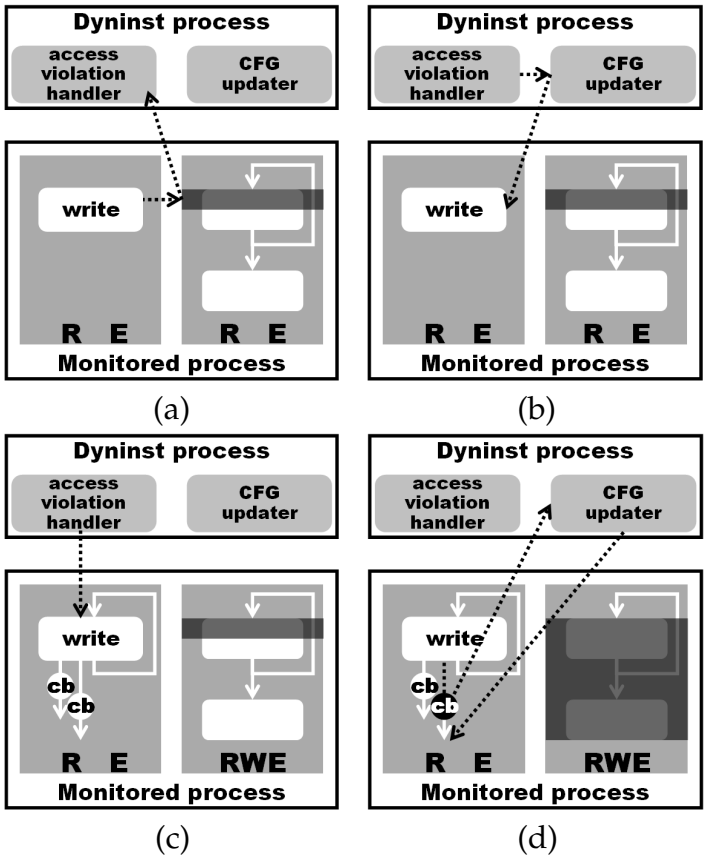


Figure 4.1: Our approach to detecting code writes is shown in part (a). In (b) we show a correct inefficient response to overwrites that updates the CFG in response to each code write, triggering major processing. In (c) and (d) we show our optimized code-write response techniques. In (c) our optimized handler for access-rights violations instruments the write loop at its exit points with callbacks to our CFG updater and restores write permissions to the overwritten page. When writing completes in (d), the instrumentation at a loop exit triggers a callback to our CFG updater.

describe in detail: a handler for the access-rights violation resulting from the first write attempt, and a CFG update routine that we trigger before the modified code has a chance to execute.

Response to the Initial Access-Rights Violation

When a write to a code page results in an access-rights violation, our first task is to handle the exception. To handle the malware's exceptions appropriately, we use the faulting instruction's target address to disambiguate between real access-rights violations and overwrite-detection violations. Our overwrite-detection mechanism introduces write-permission violations that we handle ourselves; for real access-rights violations we apply the techniques of Section 4.3 to analyze the malware program's handler and pass the signal or exception back to the malware. There are multiple acceptable approaches by which to intercept these signals and exceptions, our implementation of this technique is built on standard use of the debugger interface; the operating system gives the debugger process a first chance at handling all signals and exceptions. Instrumenters that live in the same process as the monitored program intercept signals and exceptions by keeping their own handlers registered in front of any handlers used by the monitored program [72].

Our overwrite-detection handler also decides when to trigger the routine by which we update our control flow graph in response to the overwrite. Correctness demands that we update our analysis before the modified code executes, but we can usually satisfy this constraint while batching incremental code overwrites, by triggering the update after the program is done overwriting its code. An apparently straightforward approach to accomplishing this goal is to restore write permissions for the overwritten page and remove execute permissions from the page, thereby causing a signal to be raised when the program attempts to execute code from the overwritten page (similar to the approach taken by OmniUnpack [74], Justin [53], and Saffron [98]). Unfortunately, this approach requires modifications to the operating system, as Windows will not remove execute permissions from a process's memory pages if a debugger (Dyninst in our case) is attached to it. Furthermore, this technique fails to detect the end

of incremental overwrites in the common scenario in which the program overwrites code on the memory page from which it is currently executing.

As illustrated in Figures 4.1c and 4.1d, we are able to delay updating the control flow graph until the end of overwriting by leveraging our pre-execution analysis of the program to analyze the overwrite loop and delay analysis updates until the loop is done executing. We detect the loop by performing natural loop analysis on the function that contains the faulting write instruction, and on the chain of calling functions that have frames on the call stack. When there is more than one loop to choose from, we adopt a heuristic that selects the largest loop in the currently executing function, moving up the call stack to find a loop if the current function contains none. The intuition behind this choice was to choose a loop that is big enough to contain the entire overwrite loop, without selecting a loop that is so large as to encompass the majority of the program's code, at which point the overwrite is likely to modify code pertaining to the loop. We ensure that the CFG update routine will be invoked when the loop exits by instrumenting its exit edges with callbacks to our CFG update routine.

To keep each of the loop's subsequent writes to the code page from triggering additional access rights violations, our overwrite handler restores write permissions to the overwritten code page. We deal with overwritten code ranges that span multiple code pages by maintaining a list of overwritten code pages for each loop. As the overwrite loop triggers additional access violations by spilling over into additional code pages, we add new pages to the list and restore write permissions to those pages. Our handler must also save a pre-write copy of each overwritten memory page so that when the write loop exits, the CFG update routine will be able to identify the page's overwritten instructions by comparing the overwritten page to the pre-write copy of the page.

We must take extra precautions when a loop's write instructions modify code on any one of the loop's own code pages, as restoring write permis-

sions to those pages allows the program to overwrite the loop's code and execute modified instructions before we have analyzed them. To safeguard against this scenario, if the loop writes to one of its own code pages, we add bounds-check instrumentation to all of the loop's write operations so that any write to the loop's code will immediately trigger our CFG update routine. This scenario is exceedingly rare, however. In most cases the write loop terminates and instrumentation at one of the loop's instrumented exit edges causes the CFG update routine to be invoked.

Updating the Control Flow Graph

Our control-flow graph update routine performs a sequence of tasks that we list below and then proceed to describe in detail:

1. It identifies overwritten and unreachable code.
2. It removes dead code from the analysis.
3. It triggers re-parsing of code in overwritten regions.
4. It calls back to the user tool, presenting the updated analysis.
5. It applies our dynamic capture instrumentation to new and modified binary code.
6. It removes write permissions from the modified code pages and resumes execution.

Dead-code identification. Our first task is to identify overwritten code and code that has become unreachable because of the overwrite. Finding the overwritten blocks is the first and easier of the two identification problems. We begin by identifying changed bytes on the modified code pages by comparing our pre-write copies of those pages to their current contents. We identify overwritten basic blocks by checking for overlap

between these modified bytes and the blocks on the overwritten code pages.

Correct program instrumentation does not require that we remove unreachable blocks from our analysis, but since our analysis may ultimately be consumed by a human user, it is important to strip all dead code from it so that the human user can focus on the relevant code. We detect dead code by removing all blocks from the CFG that are only reachable from overwritten code blocks, but in doing so there are two complicating factors to consider. First, some basic blocks are shared by multiple functions, meaning that we can only remove a block if the overwrite makes it unreachable in the context of all functions to which it belongs. Second, some obfuscated programs overwrite portions of the function that is currently executing and portions of functions that have frames on the call stack. This means that even when the entry point of one of these functions is overwritten, making all of its subsequent blocks unreachable from the function's entry point, there is at least one additional entry point into the functions from which the program will resume execution (more if recursion causes the function to appear in multiple call-stack frames). Thus, we remove a basic block from the control-flow graph only if it is unreachable from overwritten blocks, from the active frame's PC address, and from the return address of all additional frames on the call stack). We recover these stack frames using Dyninst's stack-walking capabilities; this Dyninst component is also available as the independent StackwalkerAPI library [88], so this dependence does not limit our overwrite-response techniques to being used by Dyninst.

As final step that we must perform before actually removing the dead code, we prepare to re-parse the overwritten code regions by identifying all *live basic blocks* (i.e., basic blocks that are not slated for deletion) that have control-flow edges into overwritten basic blocks. The challenge in identifying these live blocks is that the immediate predecessors of over-

written blocks may be slated for deletion because they are only reachable from overwritten code. Thus, we identify live blocks by traversing source edges in the CFG, starting from overwritten blocks and proceeding in a breadth-first search that terminates at live blocks or when we detect that we are in a loop that consists entirely of dead blocks.

Dead code removal. We remove dead code from the CFG and from our internal data structures, a conceptually simple task, but one that requires careful implementation so as not to leave any dangling pointers to deleted data structures.

Re-parsing of overwritten code. We analyze the new code that has been generated in the place of overwritten code by triggering parsing at the CFG edges into modified code regions. We identify these control-flow edges in our dead-code identification step, as described above.

User callback. We inform the analyst's tool of the changes to the program's control flow graph, providing lists of new basic blocks and information about the deleted blocks. This callback gives the tool a chance to consume our updated analysis, to re-instrument modified functions, and to add instrumentation to newly discovered functions.

Dynamic capture instrumentation of new code. Because code overwrites add new code to the program, we must apply dynamic capture instrumentation to the new and modified code regions before resuming the program's execution.

Resuming the program's executions. Having updated our analysis and instrumentation of the program, we re-protect the overwritten code pages by again removing write permissions from them. At this point we resume the monitored program's execution, with our dynamic analysis techniques ready to resolve any further obfuscations that we might encounter.

4.3 Exception-Handler Analysis

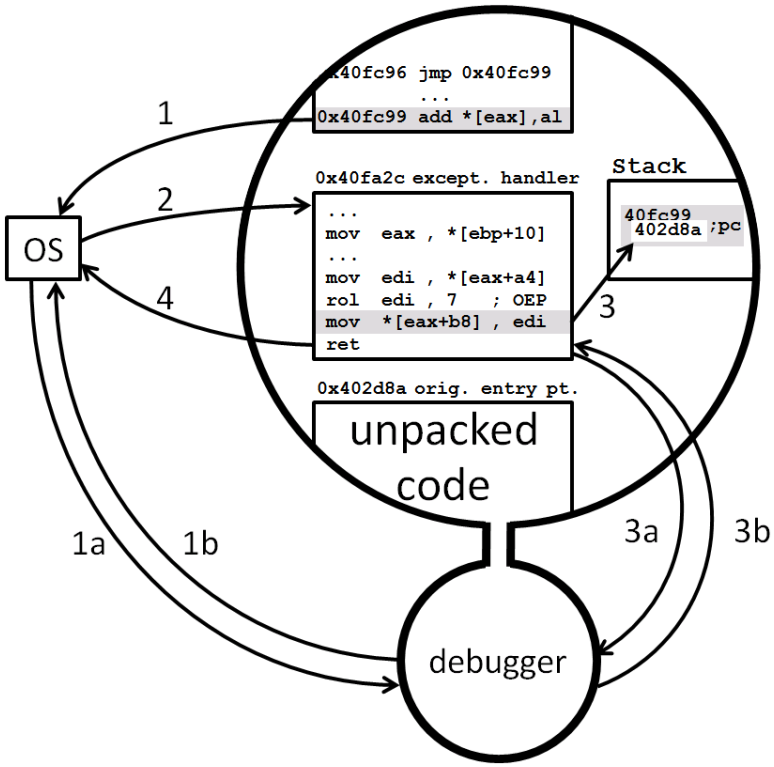
Analysis-resistant programs often are obfuscated by signal- and exception-based control flow. Static analyses cannot reliably determine which instructions will raise signals or exceptions, and have difficulty finding signal and exception handlers because they are usually registered at run-time and are often not unpacked until run-time. The ability to analyze and control the execution of signal and exception handlers is important for obfuscated binaries because many of them use these handlers to perform tasks that have nothing to do with signal and exception handling; for example, their handlers may unpack or overwrite existing code (e.g., PECompact [25]).

Signal and exception handlers can further obfuscate the program by causing control flow redirections [31, 93]. When a signal or exception is raised, the operating system provides the handler with context information about the fault, including the value of the program counter. The handler can cause the operating system to resume the program's execution at a different address by modifying this saved PC value. Figure 4.2 illustrates how the Yoda's Protector packer [30] implements this technique to obfuscate its control transfer to the packed program's original entry point. Yoda's Protector raises an exception (step 1), causing the operating system to invoke Yoda's exception handler (step 2). The exception handler then overwrites the saved PC value with the address of the program's original entry point (step 3), so that when the handler exits (step 4), the operating system will resume the program's execution at that address (step 5).

To find signal and exception handlers, we adopt the approach used by interactive debuggers and dynamic instrumenters, that is, we intercept signals and exceptions at run-time through the debugger interface provided by the operating system [72]. As illustrated in Figure 4.2, the operating system gives the debugger process a first chance at resolving signals and exceptions. We use this mechanism to find the program's registered handlers at run-time, and subsequently analyze and instrument

them. We find handlers in Windows programs by traversing the stack of structured exception handlers that are registered to the faulting thread. Finding handlers is even easier in Unix-based systems because only one signal handler can be registered to each signal type. We analyze the handler as we would any other function, and mark the faulting instruction as an invocation of the handler.

We guard against the possibility of a handler-based redirection of control flow by instrumenting the handler at its exit points. After analyzing the handler, but before the handler executes, we insert our exit-point instrumentation and copy the saved value of the program counter register (step 1b of Figure 4.2). We then inform the analyst's tool of the signal or exception and of the newly discovered handler code, allowing it to insert instrumentation of its own. We then return control to the operating system and invoke the monitored program's exception handler. When the handler has completed its execution, our exit-point instrumentation triggers a callback to our process (steps 3a-3b of Figure 4.2), where we check whether the handler has modified the saved PC value. If we detect a change, we analyze the code at the new address, instrument it, and allow the analyst to insert additional instrumentation.



1: A store to address 0 causes an access violation and the OS saves the fault's PC on the call stack.
- ***1a:*** *The OS informs the attached debugger process of the exception.*
- ***1b:*** *We analyze the registered exception handler, instruments its exit points, and returns to the OS.*

2: The OS calls the program's exception handler.

3: The exception handler overwrites the saved PC with the program's original entry point.
- ***3a:*** *The handler's exit point instrumentation transfers control to Dyninst.*
- ***3b:*** *Dyninst detects the modified PC value, analyzes the code at that address, and resumes the handler's execution.*

4: The handler returns to the OS.

5: The OS resumes the program's execution at the modified PC value, which is the program's original entry point.

Figure 4.2: The normal behavior of an exception-based control transfer used by Yoda's Protector is illustrated in steps 1-5. Steps 1a-1b and 3a-3b illustrate our analysis of the control transfer through its attached debugger process.

5 STEALTHY INSTRUMENTATION

In our approach to statically and dynamically analyzing malware, we use binary instrumentation to help monitor and control the execution of malware program binaries. We must therefore instrument malware executables in such a way that they cannot tell that we are instrumenting them. There are two reasons why stealthy instrumentation of malware is a challenging problem. First, malware actively tries to detect instrumentation. For example, malware programs may try to perform self-checksumming to detect any modifications made to their code, and as discussed in Section 2.2, many binary instrumenters rely on *code patching* techniques that overwrite portions of the malware's code [16, 65, 113, 114]. Self-checksumming techniques detect the presence of code patches by comparing a pre-computed checksum of the code to a checksum that is computed at run-time over the program's code bytes. Second, many instrumenters identify code based on static analyses of the program binary [16, 33, 64, 65, 79, 113, 114], and as discussed in Chapter 3, malware obfuscations make it extremely difficult to accurately disambiguate between code and data bytes; even our conservative parsing techniques occasionally mis-identify junk bytes as code. Thus, based on a static analysis that includes some non-code bytes, these instrumenters may patch over data, resulting in unintended effects to the program's behavior. In this chapter, we present instrumentation techniques that stealthily instrument malware binaries that are highly defensive, even when our analysis mis-identifies data bytes as code.

Interestingly, optimizing compilers also generate code that is sensitive to instrumentation. For example, optimizing compilers sometimes generate instructions that grab a constant from nearby code bytes that happen to match a needed value, thereby obtaining some modest savings in overall code size. If the instrumenter has patched over those code bytes, the sensitive instructions will behave differently than in the original binary, and will

often affect the overall behavior of the program. Furthermore, compiler-generated code that has been stripped of symbol information is difficult to statically analyze with perfect exclusion of non-code bytes [102, 107]. Thus, the techniques we develop with an eye to safely instrumenting obfuscated malware are also beneficial to binary analysts outside of the forensic malware analysis domain. Areas for which binary instrumentation is a fundamental monitoring technology include program auditing [129], precise behavior monitoring [91], debugging [36], attack detection [121], and performance analysis [111].

Current binary instrumenters have two significant weaknesses. First, they attempt to preserve the original binary's *visible behavior* (that is, the output produced for a given input) with ad-hoc instrumentation techniques that do not take into account the full spectrum of current defensive techniques by which malware can detect their instrumentation. Second, in their efforts to preserve original visible behavior, instrumenters may impose significant and unnecessary execution overhead by preserving aspects of the original binary's behavior that do not impact its visible behavior. These two weaknesses have the same cause: the lack of a formal model of how the techniques they use to insert instrumentation impact the visible behavior of the original code. Instead, current binary instrumenters rely on ad-hoc approximations of this impact [14, 16, 70, 73, 81].

In this chapter, we build on a formal specification of how inserting instrumentation will affect the visible behavior of the instrumented binary. This specification allows our stealthy instrumentation algorithm to precisely know which aspects of the original binary's behavior it must preserve to ensure that the instrumented binary's visible behavior is compatible with that of original binary. More precisely, we leverage a dataflow analysis that uses this specification to determine an overapproximation of the impact of instrumenting a binary on its visible behavior; this analysis can be used to replace the ad-hoc approximations used by current

binary instrumenters. We demonstrate that these techniques allow us to instrument programs stealthily by successfully instrumenting highly defensive program binaries that attempt to detect any modifications of their code. Furthermore, in our experiments, our techniques allowed us to safely instrument non-defensive code with 46% lower overhead than widely used binary instrumenters.

This chapter presents work that was done jointly with Andrew Bernat. As part of this joint work, Bernat developed our formalization of visibly compatible behavior, built up a taxonomy of the ways in which programs can be sensitive to instrumentation, and was primarily responsible for the aspects of our instrumentation algorithm that reduce the overhead of our instrumentation techniques on non-defensive binary code [9, 11]. I contributed the instrumentation techniques that guarantee correct behavior in the face of deliberate attempts to detect the use of instrumentation and the techniques that serve to make instrumentation transformations safe even when non-code bytes are patched over because of errors in the instrumenter's analysis of the program's code.

We have implemented the techniques described in this chapter in the Dyninst binary analysis and instrumentation framework [16] and created a *sensitivity-resistant* prototype, that we call SR-Dyninst. Our instrumentation techniques will replace Dyninst 7.0's instrumentation infrastructure in its next public release.

This chapter is organized as follows. In Section 5.1, we provide background on binary instrumentation and summarize Bernat's contributions to this work, including the formal definition of visibly compatible behavior that our instrumentation algorithm strives to preserve. We present an overview of our instrumentation algorithm in Section 5.2. In Section 5.3, we describe the techniques by which we detect and compensate for malware's deliberate attempts to recognize the side effects of instrumentation. Finally, in Section 5.4 we present experimental results that highlight the

execution-time overhead of our stealthy instrumentation techniques.

5.1 Background

In this section, we begin by describing *code relocation*, the foundational code transformation technique by which binary instrumenters add code to program binaries. We then proceed to summarize portions of our joint work on stealthy instrumentation that were contributed primarily by Andrew Bernat [9, 11]. These include a taxonomy of the ways in which instrumented programs can be affected by code relocation and a formalization of the properties of original program behavior that must be preserved in an instrumented program. Finally, we describe how our instrumentation algorithm leverages these concepts, and define the program representations used by our algorithm.

Code relocation. To understand how inserting instrumentation can affect the visible behavior of the binary, it is important to understand how instrumenters modify program binaries. Binary code is usually sufficiently compact that instrumenters cannot insert instrumentation code directly into the original code. Instead, instrumentation tools create a copy of the original code that they execute instead of the original. This code is copied with a technique we call *relocation* that produces a new version of the code that preserves the behavior of the original but contains sufficient space to insert instrumentation. Current relocation methods can be described as a combination of three basic operations: *moving* original code to new locations, *adding* instrumentation code to the moved code, and *patching* the original code's location with control transfer instructions that will re-route the program's execution towards the instrumented code.

Taxonomy of sensitive instructions. Code relocation may affect the behavior of instructions within the binary; we call such instructions *sensitive*. We separate sensitive instructions into four classes. Three of these

categories represent instructions that are sensitive because their inputs are affected by modification: *program counter (PC)* sensitivity, *code as data (CAD)* sensitivity, and *allocated vs. unallocated (AVU)* sensitivity. The fourth category, *control flow (CF)* sensitivity, represents instructions whose control flow successors are moved. PC sensitive instructions access the program counter and thus will perceive a different value when they are moved. CAD sensitive instructions treat code as data, and thus will perceive different values if they read code bytes that have been patched by the instrumenter. Problems that result from instrumenting data bytes because they were mistakenly included in the instrumenter's static analysis can be treated as CAD sensitivies, as the program treats these bytes as data, but the instrumenter believes them to be code. AVU sensitive instructions attempt to determine what memory is allocated by accessing unallocated memory and thus may be affected when the instrumenter allocates new memory to hold moved and added code. The notion of AVU-sensitivity was first identified and developed as part of our joint work. Bernat wrote a simple program that determines the shape of its address space by identifying allocated memory pages; this program detects the modifications made by widely used instrumenters. CF sensitive instructions have had a control flow successor moved and thus may transfer control to an incorrect location. For each category of sensitivity we define how the inputs and outputs of the sensitive instruction are changed by modification.

External versus internal sensitivity. Our ability to stealthily instrument code with lower overhead than current instrumentation techniques depends largely on our ability to distinguish between sensitive instructions that are externally sensitive and those that are only internally sensitive. An instruction is *externally sensitive* if it is sensitive to instrumentation and its modified behavior will also cause the visible behavior of the binary to change. Instructions that are *internally sensitive* to instrumentation produce different behavior that does not cause visible changes to the pro-

gram's behavior. When current instrumenters identify an instruction they believe to be externally sensitive, they *compensate* for its sensitivity by replacing it with code that emulates its original behavior. For example, an instrumenter may emulate a moved call instruction by saving the original return address on the call stack and then branching to the target of the call. This approach imposes overhead that may become significant if the emulation sequence is frequently executed. For example, emulating all branches can impose almost 200% overhead [70], while doing so for all memory accesses as well can impose 2,915% overhead [33]. Therefore, the instrumenter should only add compensation code where necessary to preserve visible behavior from the effects of externally sensitive instructions. More precisely, overapproximating internally sensitive instructions as externally sensitive imposes unnecessary overhead, as these instructions do not change visible behavior.

Whether an instruction is externally or internally sensitive cannot be derived from how the behavior of the instruction was affected by the code change; it also depends on how this behavior affects the surrounding code. For example, consider the effects of changing the location of a function that contains a call instruction. A call instruction saves the address of the subsequent instruction on the call stack; calls are sensitive to being moved because a move will change this address. Similarly, the return instructions in the callee are sensitive if they reference this address. If the only instructions that use the stored return address are these return instructions, then moving the call will not change the program's control flow (and its visible behavior will not change). In this case, both the call and return instructions are internally sensitive. However, our studies of malware and optimized code has shown that these return addresses are often used for other purposes. For example, malware binaries often pop return addresses off of the call stack and into general-purpose registers, subsequently using the return address as a pointer to data. In these cases,

moving the call usually affects the program's visible behavior, rendering the call externally sensitive. We build dataflow analyses that disambiguate between internally and externally sensitive instructions.

Output flow compatibility. We formalize our requirement that an instrumented program preserve the original program's behavior in terms of denotational semantics [109]. Two programs have the same denotational semantics if, for the same input, they produce the same output. Requiring strict semantic equivalence would not allow instrumentation to consume input or produce output; we address this limitation by assuming instrumentation code has its own input and output spaces and defining *compatible visible behavior* as denotational semantic equivalence over the input and output spaces of only the original program.

Determining denotational semantic equivalence between two programs is frequently an undecidable problem, but three characteristics of binary instrumentation make it tractable. First, since relocation only moves and adds but never deletes code there is a correspondence between each basic block in the original binary and a basic block in the instrumented binary. Second, we assume that executing added code will not change the behavior of the original code because executing instrumentation has no cumulative semantic effect on the surrounding code. Third, instrumentation does not purposefully alter the control flow of the original code, so the execution order of the instrumented binary will be equivalent to the original when the new locations of moved code are taken into account.

We include these three requirements in our definition of *output flow compatibility*. A formal definition of output flow compatibility is presented in our joint research paper [11] and in Andrew Bernat's Ph.D. dissertation [9]. Conceptually, we require that the instrumented binary have the same denotational semantics as the original binary over the original binary's input and output spaces, and that the control flow graph of the instrumented binary be equivalent to that of the original binary when the

inserted instrumentation code is disregarded.

Algorithm Overview. Our instrumentation algorithm builds on the concepts of this section in the following fashion. Our instrumenter relocates code to insert instrumentation. We then perform a dataflow analysis that identifies externally sensitive instructions with no false negatives (guaranteed) and with fewer false positives than current techniques. In other words, this dataflow analysis identifies all instructions that we must compensate for, while allowing us to execute most internally sensitive instructions natively, thereby reducing the instrumentation's execution-time overhead. We determine how a sensitive instruction affects the behavior of the binary by using symbolic evaluation [29]; if the program's visible behavior may change, we conclude the instruction is externally sensitive.

We then describe the techniques by which we compensate for externally sensitive instructions. We can often achieve efficient compensation by performing *group transformations* that replace a sequence of affected code as a single unit rather than compensating separately for each individual externally sensitive instruction, as current instrumenters do. This technique results in a 23% decrease in overhead when instrumenting position-independent code (such as is frequently found in shared libraries).

Andrew Bernat was primarily responsible for concept and implementation of our internal-versus-external sensitivity analysis and group compensation techniques. In practice, the primary benefits of these techniques are their ability to inexpensively compensate for program-counter and control-flow sensitivities. The code-as-data and the allocated-versus-unallocated sensitivities that we focus on in this chapter turn out not to benefit extensively from these techniques because they depend on program slices over binary code [23], a dataflow analysis technique that is notoriously imprecise. For these reasons, in this chapter we elide the details of our internal-versus-external sensitivity analysis and group compensation techniques; these details are provided in our joint paper [11] and in Bernat's

dissertation [9]).

Program representation. We represent a binary program in terms of a process state, control flow graph (CFG), and data dependence graph (DDG). We extend the conventional definition of a process state to include input and output spaces; this extension allows us to represent an input operation as a read from an abstract *input location* and an output operation as a write to an abstract *output location*. We adhere to the conventional definition of a CFG as used throughout this dissertation, viz., a collection of basic blocks connected by control flow edges. We represent the data flow of a program with a data dependence graph (DDG). The conventional definition of a DDG over binaries [61] may overapproximate data dependences between instructions that define multiple locations, as is common in real instruction sets (Figure 5.1), so we provide more precise dependence information by splitting such instructions into sets of single-definition *operations* and using these operations as nodes in the DDG. We show an example of our extended DDG in Figure 5.1.

5.2 Algorithm Overview

This section presents our algorithm for sensitivity-resistant binary instrumentation that stealthily instruments program binaries. To provide context, we compare our algorithm to a generic algorithm representative of existing binary instrumenters [14, 16, 70, 81]. These algorithms are compared and contrasted in Figure 5.2. Both our sensitivity-resistant algorithm and the generic algorithm are divided into three phases: preparation, code relocation, and program compensation. The preparation phase selects which code to relocate and allocates space for this code; this phase is the same in both algorithms. The code relocation phase copies the selected code, applies compensatory transformations to preserve its original behavior, and writes the transformed code into the program. The program

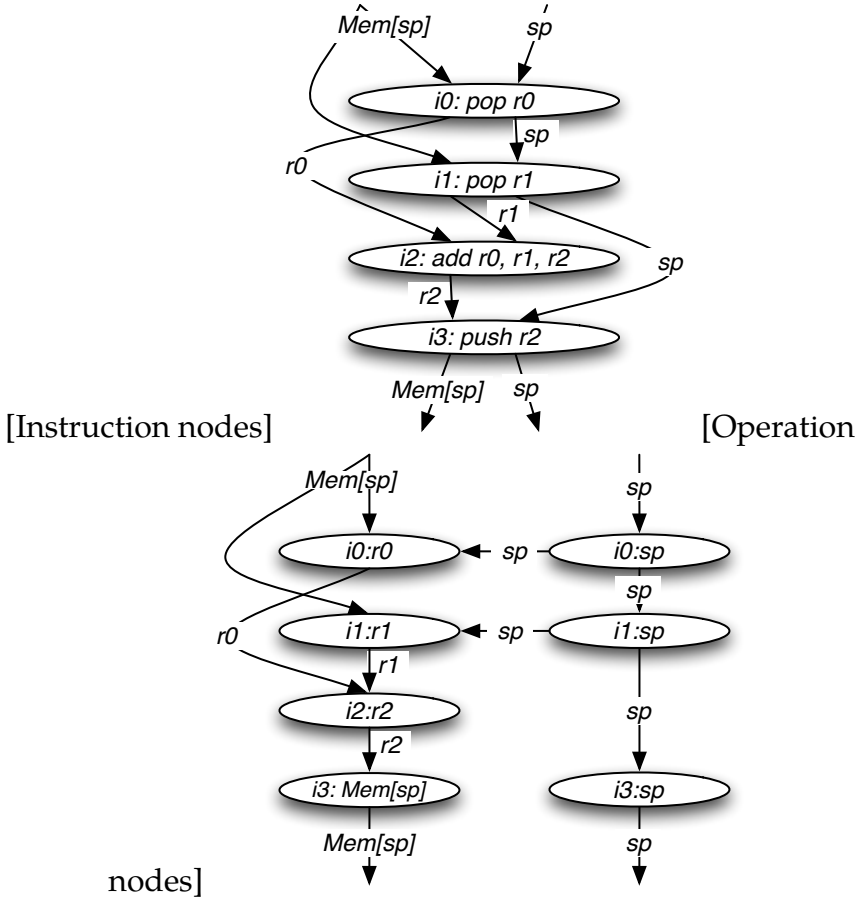


Figure 5.1: Data dependency graphs. Figure (a) illustrates the problems of representing instructions as single nodes. In this graph it is possible for paths to "cross" definitions; for example, there is a path from the definition of r_0 by i_0 to the definition of the stack pointer sp by i_3, when in the actual program there is no such dependence. Our extended DDG, shown in (b), makes the intra-instruction data dependencies explicit and excludes erroneous paths. For clarity, we omit the condition register *cc* and the program counter *pc*.

Generic Instrumentation Algorithm

```
1:  Instrument(program, instCode, instPoint)
2:      // Preparation Phase
3:      codeToRelocate = SelectCode(program, instPoint)
4:      newAddr = AllocateSpace(codeToRelocate+instCode)
5:      // Code Relocation Phase
6:      relocCode = RelocateCode(codeToRelocate)
7:      program.Write(relocCode ∪ instCode, newAddr)
8:      // Program Compensation Phase
9:      TransferToInstrumentation(relocCode)
10: RelocateCode(codeToRelocate)
11:   foreach insn in codeToRelocate
12:     if IsPCorCFSensitive(insn)
13:       relocCode.insert(Emulate(insn))
14:     else relocCode.insert(insn)
15:   return relocCode
16: TransferToInstrumentation(relocCode)
17:   foreach insn in relocCode
18:     branch = GetBranch(insn.origAddr, insn.relocAddr)
19:     program.Write(insn.origAddr, branch)
```

Sensitivity Resistant Algorithm

```
1:  Instrument(program, instCode, instPoint)
2:      // Preparation Phase
3:      codeToRelocate = SelectCode(program, instPoint)
4:      newAddr = AllocateSpace(codeToRelocate+instCode)
5:      // Code Relocation Phase
6:      relocCode = SR_RelocateCode(codeToRelocate)
7:      program.Write(relocCode ∪ instCode, newAddr)
8:      // Program Compensation Phase
9:      SR_TransformExistingCode(program)
10: SR_RelocateCode(codeToRelobcate)
11:   foreach insn in codeToRelocate
12:     if IsExternallySensitive(insn)
13:       relocCode.insert(SelectEfficientCompensation(insn))
14:     else relocCode.insert(insn)
15:   return relocCode
16: SR_TransformExistingCode(program)
17:   foreach insn in program
18:     if IsExternallySensitive(insn)
19:       program.Write(SelectEfficientCompensation(insn),insn.addr)
```

Figure 5.2: An overview of previous instrumentation techniques and our contributions. We highlight portions of the generic algorithm that our sensitivity-resistant algorithm replaces in red, and our new code in green.

compensation phase determines whether any non-selected instructions must also be transformed and applies appropriate transformations.

Our work addresses two weaknesses in the generic algorithm. First, the generic algorithm may fail to correctly identify and transform externally sensitive instructions, thus failing to preserve the original visible behavior. Second, this algorithm may apply compensation transformations to instructions that are not externally sensitive, thus incurring unnecessary run-time overhead. These weaknesses are due to the use of ad-hoc sensitivity identification techniques; we address them by using analysis to identify which instructions are externally sensitive and by transforming only these instructions. We proceed by describing each phase of the instrumentation process, highlighting the changes we make the generic algorithm.

Preparation phase Instrumenting a sequence of code requires expanding the sequence to create sufficient space to insert instrumentation. Since this frequently can not be done in place, the code is instead relocated to newly allocated memory. In its preparation phase, the instrumenter selects which code will be relocated with a function `SelectCode` and allocates memory with a function `AllocateSpace`:

`SelectCode`: This function identifies a region of code (e.g., a basic block or function) to be relocated. This region must cover at least the location being instrumented. There is no consensus among previous instrumenters on the size of the region to relocate; some instrumenters relocate a single instruction, while others relocate either a basic block, a group of basic blocks, or a function. Our algorithm is compatible with any of these choices of region size.

`AllocateSpace`: This function allocates space to contain the combination of relocated code and instrumentation (e.g., by expanding the binary on disk or mapping additional memory at run-time). Previous instrumenters have assumed that allocating memory has no effect on the behav-

ior of the program, which may not be the case if the program includes AVU-sensitive instructions. Our algorithm addresses this possibility by explicitly detecting AVU-sensitive instructions in its code relocation and program compensation phases.

Code Relocation Phase This phase relocates the selected code to the memory allocated during the preparation phase, creating a sequence that should preserve the behavior of the original code. Previous instrumenters use relocation techniques that identify and compensate for PC- and CF-sensitive instructions, but these previous techniques do not consider CAD- or AVU-sensitivity. Assuming there are no AVU-sensitive instructions is not safe. The assumption that instructions are not CAD sensitive may be safe if the instrumenter does not patch original code. Code patching serves to redirect the program's control flow from un-instrumented to instrumented code; patch-free instrumenters instead redirect control flow by instrumenting every control transfer in the program [14, 70, 81]. Since this patch-free style of instrumentation imposes significant overhead for even a single piece of inserted instrumentation, many instrumenters prefer patch-based techniques, as they allow the overhead of instrumentation to increase proportionately with the cost of instrumenting the program [10]. Our analysis-driven algorithm allows all classes of instrumenters to stealthily instrument programs by using analysis to identify all sensitive instructions, including CAD- and AVU-sensitive instructions. We represent this change by replacing the generic algorithm's call to its `RelocateCode` function with a call to `SR_RelocateCode` (line 6).

`RelocateCode`: This function examines each selected instruction (line 11), determines which instructions are PC- and CF-sensitive (line 12), and replaces them with code that emulates their original behavior (line 13). All instructions that are not sensitive are copied without modification (line 14). This function produces a code sequence that will have the same behavior as

the original when executed at the new address if all sensitive instructions were properly identified. `RelocateCode` also creates the appropriate space between relocated instructions to make room for instrumentation; this is not shown. For clarity, we describe this function in terms of a single pass; however, some instrumenters use a fix-point iteration to further optimize the relocated code.

`SR_RelocateCode:` Our work improves `RelocateCode` with analysis that identifies externally sensitive instructions; this analysis is represented by `IsExternallySensitive` on line 12. In our `SelectEfficientCompensation` function (line 13), we look for more efficient compensation transformations than can be attained through separate emulation of each original instruction. Our implementations of these two functions for PC- and CF-sensitive instructions are described in detail in Andrew Bernat's dissertation [9] and in our joint paper [11]. We describe our sensitivity analysis and compensation techniques for CAD- and AVU-sensitive instructions in Section 5.3.

Program Compensation Phase This phase attempts to preserve the original behavior of sensitive instructions that were not relocated, and therefore has no effect if all code is relocated [14, 70, 81]. Previous patch-based instrumenters have used it to insert jumps to relocated code with a `TransferToInstrumentation` function [16, 79]. This function does not consider the possibility of CAD- or AVU-sensitive instructions in non-relocated code. We address this by instead using `SR_TransformExistingCode`, which uses our analysis to identify all externally sensitive instructions.

`TransferToInstrumentation:` This function patches original code with branches to the corresponding locations of relocated code (lines 18 and 19). This approach ensures that CF sensitive instructions do not affect the program's behavior, but overwrites original code and thus may trigger CAD sensitivity.

`SR_TransformExistingCode`: This function is similar in structure to `SR_RelocatedCode` and shares many elements of its analysis. We examine each instruction (line 17) to identify the externally sensitive ones (line 18). We then apply an efficient compensatory transformation and write the transformed code to the program (line 19). For clarity, our description of this algorithm has assumed that the compensatory transformation does not increase the size of the code or affect the sensitivity of additional instructions. As these assumptions may not hold in practice, we use a fix-point algorithm that adds any additional affected instructions and converges when no additional code must be relocated.

5.3 CAD and AVU Detection and Compensation

In this section, we focus on the following three aspects of program binaries that make stealthy instrumentation difficult. The first is that binaries may contain CAD-sensitive instructions that could detect patched code bytes. Many defensive malware binaries employ CAD-sensitive code, and optimizing compilers also generate CAD-sensitive instructions that grab constants from nearby code bytes when these bytes happen to match a needed value. Second, the program could contain AVU sensitive instructions could detect that we have allocated extra space in the program's address space. Third, there is a great deal of ambiguity between code and non-code bytes in program binaries, making it extremely difficult for instrumenters to identify code with perfect exclusion of junk bytes (see Section 2.2). Thus, even if a non-defensive program contains no CAD-sensitive instructions, the instrumenter may still cause changes to visible behavior by patching data bytes that it believes to be code. Interestingly however, since the instrumenter's analysis believes these junk bytes to be code, its analysis will also consider any instructions that read from these

bytes to be CAD sensitive. Thus, it becomes doubly important for us to solve the problems of CAD-sensitivity detection and compensation, as these techniques will also compensate for junk-code instrumentation.

CAD and AVU sensitivities are similar in that they both involve memory access instructions; CAD instructions read from patched code bytes while AVU instructions access memory addresses that should be un-allocated. Thus, all load instructions are potentially CAD sensitive, while all memory access instructions are potentially AVU sensitive. We use the same approach to detect whether memory access instructions are externally CAD and AVU sensitive, and use similar approaches to compensate for these types of program sensitivities. We proceed by discussing detection of CAD and AVU sensitive instructions, and then discuss how we compensate for these instructions.

External CAD and AVU Sensitivity Detection

To guarantee visibly-compatible program behavior, we must identify a set of instructions that includes all instructions that are externally CAD and AVU sensitive, and then compensate for the effects of instrumentation on these instructions. We also want to achieve low instrumentation overhead, and one way to accomplish this is to choose this set so that it includes few instructions that are neither AVU nor CAD sensitive, and few instructions that are internally CAD sensitive. We do not seek to exclude internally sensitive AVU instructions, because all AVU instructions are externally sensitive. Internally sensitive AVU instructions do not exist because AVU instructions trigger an access violation in the original binary and our instrumented code must trigger this same control-flow edge to satisfy our definition of output-flow compatibility (see Section 5.1).

In this section, we begin by discussing the problem of disambiguating between sensitive and non-sensitive memory access instructions. We then discuss the task of disambiguating between instructions that are internally

versus externally CAD sensitive. We also discuss performance considerations that argue in favor of emulating all memory access instructions in the program, and then present our solution based on these considerations.

Identifying non-sensitive instructions. To identify non-sensitive memory access instructions, we must know the range of addresses at which these instructions can read or write. If the target of a memory access instruction does not correspond to patched code, it is a non-sensitive instruction, though this evaluation is subject to change along with changes in our instrumentation and analysis of the program. Memory access instructions that read and write to the program's call stack are not sensitive except in the rare case that the malware has executable code on the call stack. A similarly easy determination of sensitivity can be made for load and store instructions that have fixed target addresses. Regrettably, for many memory access instructions, determining whether they are sensitive requires a sophisticated analysis; most load and store instructions determine their targets at run-time based on register values and memory contents. The *value set analysis* designed by Balakrishnan and Reps is a natural fit for these instructions, as it is designed to bound the set of input (and output) values that are consumed (and produced) by x86 instructions [5]. Regrettably, this type of analysis has limitations that make it unsuitable for defensive malware code. Its most significant problems are that it assumes a full static analysis of the program and does not account for code overwrites. We could, of course, recompute value-set analysis each time we discover new or overwritten code, but value-set analysis was not designed for incremental updates and is extremely resource-intensive to compute, even a single time. Thus, the need to recompute value-set analysis at run-time in response to code changes would defeat the reason for which we would want to use it, which is to reduce execution-time overhead by not compensating for non-sensitive instructions.

Identifying internally CAD-sensitive instructions. A second way to

```
1:  mov esi, ptr[0x40dc00]
2:  ...
3:  call esi
```

Figure 5.3: A code sequence used by the NPack packer. The target of the call on line 3 depends on the the contents of memory read by the load instruction at line 1. If instrumentation modifies this memory value, the call may transfer to an invalid address, resulting in an access violation.

reduce compensation overhead is to not compensate for instructions that are only internally CAD sensitive. Internally CAD-sensitive instructions are loads that would not impact the program's control-flow or output values if they were to read a different value from memory than was read by the original binary. To identify these instructions, we could apply the same approach we use to identify externally sensitive PC instructions. That is, we would identify a set S of potentially affected instructions by performing a forward slice [23] from the load instruction. We would then apply symbolic evaluation [29] to the instructions in S to determine what impact the load's modified value actually has on these instructions. Unfortunately, we usually do not know the modified value, so symbolic evaluation cannot tell whether the affected instructions in S will execute without raising an access violation. For example, consider the sequence of instructions shown in Figure 5.3. If instrumentation changes the memory contents read by the instruction on line 1 to an unknown value, we will not know whether the call instruction on line 3 will transfer control flow to a valid address. This limitation sharply limits the number of internally sensitive instructions that we could detect with this approach.

Benefits of compensating for all memory access instructions. Though our CAD and AVU detection techniques are somewhat limited, they do tell us that we do not need to compensate for memory access instructions that target the call stack, unless the stack contains code. They also help us to

identify some load and store instructions that are safe to execute natively because they do not target patched code locations. However, there are significant benefits to compensating for all memory access instructions that could target executable memory regions, whether or not they target patched code in those regions. In particular, compensating for all such instructions allows us to eliminate concerns about CAD and AVU sensitivities from our instrumentation algorithm's `SR_TransformExistingCode` function (lines 16-19 of Figure 5.2). This means that we do not need to recompute our sensitivity analysis over the entire program each time we discover new code in the program, remove dead code, or add an additional piece of instrumentation. For defensive malware binaries, this benefit is substantial, since we have to update our analysis and instrumentation many times as the program executes. By eliminating CAD and AVU sensitivity considerations from `SR_TransformExistingCode`, we make it far more efficient, allowing this function to focus exclusively on external CF sensitivities, which we can detect inexpensively (external PC sensitivities are detected and compensated for in function `SR_RelocateCode`).

Our approach. Based on the above considerations, we choose to compensate for potential CAD and AVU sensitivities in all memory access instructions in the instrumented binary, with the exception of instructions that access only the call-stack. However, the program binary itself typically constitutes only a small fraction of the code that is present in the address space of a malware program. Since this additional code interacts with our instrumented binary, it too may be sensitive to the instrumenter's modifications. Most of this additional code is in system libraries, but the instrumented program also interacts with kernel code, either by trapping directly into the kernel, or by reaching kernel code indirectly through calls to system library functions. Though system library and kernel code are not malicious or deliberately defensive, some malware binaries pass pointers to their instrumented code regions as arguments to functions in

system-libraries and the kernel, making the system code CAD-sensitive with respect to the malware binary.

We do not instrument kernel code, so we must draw a line beyond which we compensate for CAD-sensitive code with an alternative mechanism that presents an unpatched view of the binary to the uninstrumented code. We choose to set this line at the boundary between the malware binary and system libraries (provided that these have not been modified by the malware) for the following two reasons. First, on the Windows operating system, the exported functions of system libraries are well documented because they implement the Windows API interface; this allows us to analyze system libraries and ignore calls to functions that do not take pointers as parameters. Second, we exclude more code from instrumentation and its associated overhead by drawing the boundary at the system library interface rather than at the kernel interface.

Compensation for CAD and AVU Sensitivity

We compensate for a memory access instruction that could be CAD or AVU sensitive by ensuring that its behavior matches that of the original instruction. We begin by describing our instrumentation-based compensation techniques for the simpler case of AVU-sensitive instructions, followed by our more elaborate techniques that compensate for CAD-sensitivity. We conclude with a discussion of compensation techniques that we apply at transitions to library code that could be CAD sensitive.

AVU Compensation. In the original binary, an AVU-sensitive instruction triggers an access violation by accessing un-allocated memory. We emulate this behavior in the following way. We track all memory that our instrumenter adds to the program's address space and insert instrumentation at memory access instructions to perform bounds-checking on the instruction's target address. If the address targets memory that was allocated by the instrumenter, our instrumentation triggers an access vio-

lation by re-directing the memory access to address 0. We ensure that the instrumented program's exception handler sees the same data regarding the AVU exception as the handler in the original binary, by intercepting and instrumenting the handler based on the techniques of Section 4.3.

Instrumentation-based CAD compensation. Compensating for AVU sensitivity is easier than compensating for CAD-sensitivity in that an AVU-sensitive load should fail to read anything, while a CAD-sensitive load should read the same value that was read by the original binary. Shadow memory techniques [80] provide a straightforward solution to identifying the correct value and ensuring that it is read by CAD-sensitive loads. In particular, we maintain a copy of any code bytes that get modified by the instrumenter and compensate for CAD-sensitive load instructions by having them read from the unmodified shadow copy. To keep these shadow copies up-to-date as the program changes, we must also modify store instructions that write to shadowed memory regions so that they write to the shadow copy.

In implementing a shadow memory approach, we must decide at which granularity to maintain the shadow memory and consider the performance implications of this choice. The first alternative we consider is to shadow only those regions of the binary that are patched by the instrumenter. This approach has two advantages. First, this approach shadows the smallest possible amount of memory in the monitored application, and therefore has the smallest space overhead of any shadow-memory approach. Second, for malware binaries, this approach requires no additional CAD compensation for the program's store instructions, as we already detect and monitor write instructions that could overwrite analyzed code (see Section 4.2) and the patched memory we would be shadowing is a subset of this code. We could therefore trivially keep the shadow memory regions up to date after a code overwrite by copying overwritten code bytes to shadow memory. Meanwhile, most non-defensive binaries do not

overwrite their code, so there would be no need for us to compensate for store instructions in this case.

The alternative is to shadow the program's code at a coarse granularity, meaning that shadowed regions could include uninstrumented code and non-code bytes. For non-defensive binaries and their shared libraries, each binary typically contains its code in a single section, and we can shadow each binary's section as a whole. We usually cannot shadow defensive binaries with a single block, because these programs often place code in memory ranges that are separated by large gaps of unallocated memory. However, our experience shows that most defensive malware binaries place their code in only two to four ranges. The greatest disadvantage to shadowing memory at a coarse granularity is that keeping this memory up to date requires that we instrument all store instructions that could target shadowed memory. However, this disadvantage is not all that significant because we instrument these instructions anyway to detect and compensate for possible AVU sensitivities, and part of the instrumentation overhead would be shared between the two compensation techniques. On the positive side, shadowing memory pages at a coarse granularity allows our instrumentation of the program's load and store instructions to be highly efficient. In particular, this instrumentation can detect memory accesses that target shadow memory with a single bounds-check for non-defensive binaries and with only a only a few checks for malware binaries. By contrast, for fine-grained shadowing, our instrumenter could easily patch thousands of program locations, and shadowing each individually patched region would require us to instrument each load with an expensive check to determine whether its target address lies within any of these shadowed regions.

Based on these considerations, we choose to shadow memory at a coarse granularity. We examine the performance impact of our compensation techniques with experimental results that we present in Section

5.4. These results show that the cost of our CAD and AVU compensation techniques is outweighed by efficiency gains in PC and CF sensitivity detection and compensation, resulting in instrumented execution times that are faster than those of widely used instrumenters that compensate for sensitive code with ad-hoc techniques.

CAD compensation for system-library and kernel code. When instrumenting defensive malware, we must compensate for CAD sensitivities in code that is external to the malware binary itself. These sensitivities arise when the malware passes pointers to its patched code regions as arguments to functions in system-library or kernel code. When the library code reads from these addresses, the visible behavior of the instrumented program could deviate from that of the original binary. We address this problem by presenting an unpatched view of the malware program to external library code that could be CAD sensitive. We temporarily recreate the unpatched binary by copying changes in shadow memory to the original program, with the effect of erasing the instrumenter's code patches. We call this technique *patch hiding*. We create this view when potentially sensitive third-party code is invoked; when the invoked library function returns to the malware, we restore our code patches and resume normal execution. Patch hiding includes the following sequence of steps:

1. We instrument all control transfers that transition from the program binary to system-library or kernel code. Many of these control transfers are already being monitored by our dynamic capture instrumentation (see Section 4.1), as most inter-library calls are implemented with indirect control transfer instructions that grab their targets from memory at run-time.

2. Our instrumentation looks up the call target in a table of common library functions that do not take pointers as arguments. If the target matches a function in this table, we skip the patch hiding step and continue the program's execution, otherwise we continue to step 3.

3. We perform patch hiding by overwriting the malware binary's original memory with its shadow memory. This technique updates the original memory with any changes made to the program's shadow memory and overwrites all code patches we have made to the original binary.

4. We instrument the library function's return instructions so that they will call back to the code routine of step 6. If the malware code invokes kernel code directly (by executing a trap instruction), we cannot instrument the return address of the invoked kernel function, so we set a hardware breakpoint at the trap instruction's fall-through address. When the kernel function returns, this hardware breakpoint will trigger, alerting our debugger process, which can then invoke step 6.

5. We continue the program's execution.

6. We re-instate our code patches and continue the program's execution.

We do not perform patch hiding on non-defensive binaries, as it would be uncharacteristic of such programs to perform the adversarial act of asking a system library function to read from the non-defensive binary's code. On the other hand, we do compensate for CAD sensitivities in the binary itself, because optimized code may read from code bytes and because the instrumenter's analysis of the code may include some junk bytes.

5.4 Results

The goal of our analysis and instrumentation algorithm is to preserve the semantics of the instrumented program while reducing the overhead imposed by instrumentation. We evaluated the overhead of our instrumentation algorithm by instrumenting Apache, MySQL, and the SPECint 2006

benchmarks, and comparing our algorithm's performance to the Dyninst 7.0 and PIN binary instrumenters. These results show that our techniques resulting in lower average overhead than Dyninst 7.0 and PIN, despite the additional costs we incur to compensate for CAD and AVU sensitivities. We present these experimenatl results in this section. We also performed experiments to establish our algorithm's ability to stealthily instrument defensive programs; these experiments are presented in Chapter 6.

We implemented our stealthy instrumentation algorithm in the Dyninst binary analysis and instrumentation toolkit, creating the SR-Dyninst research prototype from Dyninst's 7.0 version. SR-Dyninst identifies sensitive instructions using information provided by the InstructionAPI component of Dyninst, and builds a new symbolic evaluation and slicing component to assist in our identification of externally sensitive instructions. This semantic evaluation component uses a semantic instruction model provided by the ROSE compiler suite [97]. While our implementation and experiments were done in the context of Dyninst, the techniques and software we built can be used to extend other instrumentation tools, such as PIN, to have the same capabilities as those we added to Dyninst.

We wish to measure the change in instrumentation overhead due to our stealthy instrumentation approach as compared to the overhead of Dyninst 7.0 and the widely used PIN instrumenter [70]. To do so, we measured the execution overhead caused by executing moved and transformed code instead of original code. So that our measurements would not include the cost of any user-specified instrumentation code, we instrumented every basic block in the program but added no user code.

For SR-Dyninst (and Dyninst 7.0) we instrumented each binary with Dyninst's static binary rewriter. PIN does not provide an equivalent static rewriting capability, and thus our performance numbers for PIN include their dynamic translation cost as well as the cost of executing transformed program code. However, from their previously published results [70], this

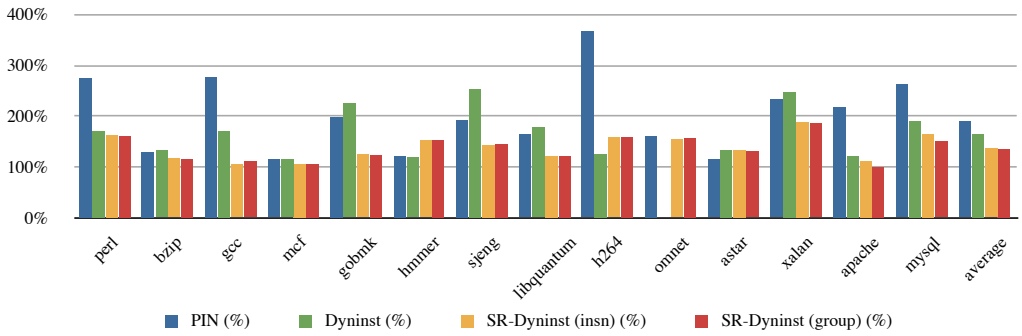


Figure 5.4: Performance of our approach compared to Dyninst 7.0 and PIN. We show two sets of results for our approach. The first uses only instruction transformations, while the second includes group transformations for thunks. The y-axis is execution time normalized to the unmodified execution time.

dynamic translation cost is small for long-running benchmarks and thus we do not believe it significantly impacts our results.

Our performance experiments were run on a set of binaries consisting of the SPECint 2006 benchmark suite, Apache, and MySQL. Each of these programs was built from source with default arguments. We instrumented both the program binary and any libraries on which it dependeds. We ran the SPECint suite using the reference data set inputs and tested Apache and MySQL with their provided benchmarking tools. These experiments were run on a 2.27 GHz Intel quad-core Xeon machine with 6GB of memory.

Our performance results are shown in Figure 5.4. The y-axis is the execution time normalized to the uninstrumented run time (100%). SR-Dyninst results in an average overhead of 36%, which is lower than both Dyninst 7.0 (66%) and PIN (90%). Our instrumentation overhead numbers are due to two factors. First, our instrumentation algorithm natively executes instructions that other instrumenters mistakenly deem to be PC- or CF-sensitive. Second, our group transformation techniques allow us to compensate for PC sensitivities in thunk functions with greater efficiency than the individual-instruction transformations employed by other instru-

menters (details of these techniques are in Bernat's dissertation [9] and our joint paper [11]). This benefit is most apparent in the Apache (12% to 0.4%) and MySQL (66% to 51%) benchmarks, because they execute more library code than the SPEC benchmarks, and thunk functions are most common in position-independent library code. Overall, these performance gains are so substantial that they hide the additional cost of compensating for CAD and AVU sensitivities. There are only two benchmarks (hmmer and h264) for which compensating for CAD and AVU sensitivities outweighs the efficiency gains of our other techniques. For these two binaries, Dyninst 7.0 and PIN execute more efficiently than SR-Dyninst because Dyninst 7.0 does not compensate for CAD or AVU sensitive code, while PIN does not compensate for AVU sensitive code (as PIN does not modify the original code, there will be no CAD sensitive instructions). However, these minor efficiency gains come at a price: malware programs can detect the instrumentation techniques used by PIN and Dyninst 7.0 by using AVU- and CAD-sensitive code. As a final note of interest, Dyninst 7.0 failed to correctly run the omnetpp benchmark due to an incorrect handling of exception-throwing code that was PC sensitive; our approach transparently handled this problem.

6 MALWARE ANALYSIS RESULTS

In this chapter, we evaluate our malware analysis and stealthy instrumentation techniques, both on real and representative synthetic malware samples that are highly defensive. We begin by revisiting our study of obfuscations used by the packer tools that are most frequently used to obfuscate malware. We presented a discussion of these obfuscations in Chapter 2, but since that chapter also serves as our summary of related works, we omitted a discussion of how our analysis techniques fare against those obfuscations. We present that discussion here, as this collection of obfuscations provides a thorough test of the technical contributions of this dissertation.

We then proceed to demonstrate our ability to analyze real-world malware binaries. Analysts at large security companies receive tens of thousands of new malware samples each day [86] and must process them efficiently and safely to determine which samples are of greatest interest. We build a *malware analysis factory* that performs batch-processing of malware samples in an isolated environment, producing customizable reports on the structure and behavior of malware samples. We describe the output of this analysis factory, and show its analysis of the notorious Conficker A binary as an illustrative example.

6.1 Analysis of Packed Binaries

In Chapter 2, we performed a broad examination of the obfuscation techniques used by the packer tools that are most popular with malware authors [18]. This study performs two useful purposes. First, our study provides a snapshot of the obfuscation techniques that we have seen to date. We hope that this study will guide future work in malware analysis and de-obfuscation. Second, our study demonstrates the ability of our

analysis techniques to cope with the obfuscations that are most-often employed by malware. In this section, we discuss the ability of our techniques to deal with each of these obfuscations in turn, in the same order that they were presented in Section 2.2. In doing so, we refer to the statistics we gathered to quantify the prevalence of these obfuscations, which are summarized in Table 2.1 of Section 2.3. For more details on the methodology we used to perform this study, we refer the reader to Section 2.1.

Code packing: A packed binary contains a payload of compressed or encrypted code that it unpacks into its address space at run-time. At least 75% of all malware binaries are packed [12, 119], and every binary in this study is packed. Furthermore, most of these binaries unpack in multiple stages (see row A1 of Table 2.1). We discover packed code at run-time, just before it executes, by using our dynamic capture instrumentation techniques (see Section 4.1) and exception-monitoring techniques (see Section 4.3) to monitor all potential transitions into un-analyzed code. Our study of packed binaries amply demonstrates our ability to find and analyze packed code, as we were able to analyze each packed payload before it executes, even for binaries that unpack in multiple stages.

Code Overwriting: Many malware binaries overwrite existing code with new code at run-time. For our analysis and instrumentation tool to correctly respond to code overwrites, we must first detect overwrites, and then respond by safely removing overwritten and unreachable code, and re-parsing new code at entry points into overwritten regions.

As discussed in Section 4.2, efficiently detecting code overwrites is challenging because malware binaries may overwrite large regions of code in one-byte increments. Our efficient detection techniques are predicated on our ability to batch analysis updates in response to incremental overwrites by using our structural analysis of the code to detect write loops and delaying our update until the loop exits, as long as it is safe to do so. Our analysis of packed binaries demonstrates that this technique is safe; it

detects each of the 46 instances of code overwrites performed by these binaries, and in each case, we update our analysis before the overwritten code executes. Our ability to batch analysis updates also dramatically improves the performance of our techniques. Before implementing overwrite batching, our initial prototype took over 3 hours to execute an instrumented UPack binary; overwrite batching reduced this time to 28 seconds [103].

Safely removing overwritten and unreachable code from the program's control-flow graph after an overwrite is a challenging problem. In particular, we must carefully determine which code is truly safe to remove after an overwrite, and do so using the techniques described in Section 4.2. Our study of packed programs demonstrates that our techniques correctly remove dead code in a wide variety of overwrite instances, including all 17 cases in which dead code removal becomes especially challenging because of overwrites to a currently executing function (see row A4 of Table 2.1).

We analyze new code that replaces overwritten code, by applying our code-parsing techniques at entry points into overwritten regions. By correctly analyzing and instrumenting the code-overwrite techniques of prevalent packer tools, we confirm that our parsing techniques successfully analyzes the new code that is introduced by overwrites.

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 return instruction to resume execution at the instruction following the `call`. However, as shown in row B4 of Table 2.1, all but two of prevalent packer tools use `call` instructions in cases where control-flow does not return to the fall-through address of the call. Our parsing techniques deal with the possibility of non-returning calls by assuming that a call does not return unless a binary slice [23] of its called function can conclusively demonstrate that it does (see Section 3.1). Our study of prevalent packer tools demonstrates the merits of our conservative parsing techniques by correctly preventing us from parsing

at the fall-through edge of all 181 non-returning calls.

Call-stack tampering: Obfuscated binaries frequently tamper with the call stack so that `ret` instructions transfer control to targets that are not call fall-through addresses. To our knowledge, however, ours is the only tool to apply static analysis techniques to `ret` target prediction (see Section 3.1). When our static techniques cannot guarantee that a function does not tamper with its call stack, we fall back on our dynamic capture techniques to monitor the targets of the function's `ret` instructions. In our study of packed binaries, most instances of call-stack tampering involve `push <addr> ; ret` sequences and are resolved by our static techniques, though ASPack, ASProtect, and Yoda's Protector contain more elaborate instances of stack tampering (see row B3 of Table 2.1) that we also analyze successfully with our static techniques.

Obfuscated control-transfer targets: Many packer tools use indirect control transfers to obfuscate their control-flow targets, thereby hiding them from static analysis. Our static techniques resolve standard uses of indirect control transfers, such as indirect jumps that implement jump tables and indirect calls that get their targets from the Import Address Table. To resolve non-standard indirect control transfers, we we rely on our dynamic capture instrumentation techniques (see Section 4.1). In our study of packed binaries (and in general), our dynamic capture techniques identify control-flow targets for indirect control transfers that execute. The only indirect control transfers that remain unresolved by our hybrid techniques are those that evade our static techniques by not conforming to a standard usage pattern and that evade our dynamic techniques by not executing. These unresolved indirect control transfers can result in gaps in our control-flow graph's coverage of the program.

Exception-based control transfers: Signal- and exception-handling mechanisms allow for the creation of obfuscated control transfers whose source instruction (the fault-raising instruction) and target address (the

signal or exception handler) are well-hidden from static analysis techniques [93]. An additional difficulty is that the exception handler specifies the address at which the system should resume the program's execution, and this constitutes yet another hidden control transfer. We detect signals and exceptions through the OS-provided debugger interface, and instrument the malware's handlers to detect the address at which they direct the system to resume the program's execution (see Section 4.3). Our techniques respond correctly to each of the 10 exception-based control transfers (row E1) employed by this collection of packed binaries, discovering the exception-raising instruction, its exception handler, and the target at which the system will resume the program's execution.

Ambiguous code and data: Packed binary code frequently introduces ambiguities between code and data, by allowing its control flow to fall through into junk code, with no intervening control-transfer instructions. We address this problem with two techniques: our parser's junk-code avoidance techniques reduce the amount of parsed junk bytes, while our stealthy instrumentation techniques ensure that instrumenting analyzed junk bytes will not impact the correctness of our instrumenter's transformations.

Our code parser uses a cost-effective approach to detect when it has transitioned into invalid code bytes; it stops disassembling when it encounters privileged and rarely used instructions that are usually indicative of a bad parse (see Section 3.1). For the packed binaries of this study, our technique allows us to stop parsing junk code within an average of 17 bytes after a transition into a junk region and within 93 bytes in the worst case, whereas we would often parse several hundreds of junk bytes if not for these junk-avoidance techniques. Since mistakenly parsed junk bytes do not cause problems for our instrumentation techniques, the primary goal of junk-code avoidance is to not overwhem the analyst with non-code bytes in the CFG, and our junk-avoidance techniques make significant

progress in this regard. In theory, our technique can cause us to prematurely stop parsing at rare instructions that are actually executed by the program. We prepare to compensate for such cases by applying dynamic capture instrumentation to rare instructions so that we will parse after them if the rare instruction executes. However, this precaution was not needed for any of the packed binaries that we studied.

A second concern resulting from code and data ambiguities is that when our parser mistakenly considers junk bytes to be code, our instrumenter may patch over junk bytes that the program uses as data, causing changes to the program's visible behavior. As explained in Section 5.3, since our instrumenter's analysis believes these junk bytes to be code, its analysis will also consider any instructions that read from these bytes to be CAD sensitive. Thus, our CAD-sensitivity detection and compensation techniques also compensate for junk-code instrumentation, as demonstrated by our successful instrumentation of the five packed binaries that introduce code and data ambiguities with control-flow that falls through into non-code instructions (row C11).

Disassembler fuzz testing: Some obfuscated malware binaries fuzz-test disassemblers and emulators with random and unusual instructions (rows B6 and C11). The use of fuzz testing means that it is usually wiser to leverage an existing, mature disassembler (e.g., XED [19], Ida Pro [52], ParseAPI [87]) 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 [97], Qemu [8], TSL [66]). Our conservative parsing techniques are based on the Dyninst framework's ParseAPI component, and our symbolic evaluation techniques (which we use to simplify binary slices) use ROSE to interpret instruction semantics. Both of these tools are exceptionally well tested.

Obfuscated calls and returns: Binary packer tools frequently use `call` and `ret` instructions where a `jmp` is more appropriate. These superfluous

`call` and `ret` instructions create the illusion of additional functions in the program's code. We address the problem of superfluous `call` instructions by performing look-ahead parsing through the first multi-instruction basic block at each call target, to determine whether the called code removes the return address from the call stack (see Section 3.2). Though this is not a rigorous solution to the problem, it works well in practice for the packer tools we have studied. While we cannot know exactly how many superfluous `call` instructions there are in these binaries, our lookahead technique significantly simplifies our analysis of obfuscated binaries; we identify 54 superfluous calls in the ASProtect packer alone, each of whose targets would have been parsed as a new function.

Superfluous `ret` instructions do not transfer control flow back to the fall-through address of a `call` instruction. We detect superflous `ret` instructions through our call-stack tampering techniques (see Section 3.1), as `ret` instructions always target call fall-through addresses in the absence of stack tampering.

Overlapping functions and basic blocks: Many malware binaries interleave blocks that pertain to different functions, to make 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 [52]. 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.). Our SR-Dyninst tool does update its CFG of the program at run-time, but lacks a GUI for interactive perusal of the disassembled code. A marriage of Ida Pro's GUI and our techniques for updating CFGs would allow for easy perusal of functions with interleaved blocks, but no such tool exists at present.

Overlapping basic blocks occur when valid instructions start at different offsets and share code bytes (see Figure 2.4). 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 [79, 123]. We design the data structures we use in SR-Dyninst to account for the possibility that multiple blocks and instructions can indeed map to any given address. Not only can we correctly analyze overlapping blocks, but SR-Dyninst correctly instruments the 17 overlapping blocks used by the PolyEnE, ASProtect, and Yoda's Protector packers, and correctly instruments a pair of overlapping blocks in the Conficker A binary.

Obfuscated constants: Malware authors use constant obfuscations to make their code harder to understand, with the goal of slowing down analysts that try to make sense of the program's instructions. Though our analysis and instrumentation techniques make no attempt to remove constant obfuscations, they provide two techniques that allow analysts to resolve these obfuscations. First, analysts can resolve constant obfuscations with our binary slicing and symbolic evaluation techniques. Second, analysts can resolve constant obfuscations with our instrumentation techniques, which allow them to observe the values produced by obfuscated instructions at run-time.

Calling-convention violations: Despite the differences between the many calling conventions for the x86 platform [46], and the fact that aggressively optimized binaries ignore many of these conventions, compiler-generated code does adhere to some conventions that do not hold for obfuscated code. These differences are relevant to our techniques because they may induce binary instrumenters to modify binary code based on the assumption that the original values of some registers do not need to be preserved across function-call boundaries. In particular, to correctly instrument the non-standard calling conventions used by obfuscated code,

we had to remove Dyninst 7.0's assumption that caller functions do not make branching decisions based on the values of status-register flags that are set by a called function. Instead, SR-Dyninst only assumes that registers set in a callee function are unused by the caller only if this is proved by binary slicing techniques [23]. This conservative approach allows us to correctly instrument the four packed binaries of this study that read status flags across function boundaries (row C9).

Do-nothing code: Some obfuscated programs incorporate code that does no useful work, to dilute the interesting parts of the program with semantic no-ops. Our analysis and instrumentation techniques do not attempt to eliminate this code, but our structural analysis and instrumentation techniques help users to focus on parts of the program that perform behaviors they are interested in. For example, the user may use our structural analysis to identify and only monitor those functions that perform network communications.

Stolen bytes: The stolen-bytes technique pioneered by ASProtect copies the first block of a system-library function to another location and re-routes the program's control flow through this "stolen" block (see Figure 2.6). This technique causes one of two problems for instrumenters that use patch-based techniques, depending on whether the byte-stealing happens before or after the instrumenter patches the system-library functions. If the byte-stealing occurs first, the instrumenter's code patches will have no effect on the program's execution, as the library function's patched first block will not execute. To detect all system-library calls, we instrument all control transfers that could enter any part of a system-library function. If, on the other hand, the code-patching occurs first, the packer's byte-stealing code will steal the patched code block instead of the original code block. SR-Dyninst avoids this scenario by shadowing instrumented system-library functions based on the CAD-compensation techniques of Section 5.3. Thus, when the CAD-sensitive instructions in the packed

program steal bytes from system-library functions, they steal original code bytes and not the patched code that Dyninst places at the function's entry points. We verify the correctness of these techniques by successfully instrumenting the ASProtect binary and logging its invocations of system-library functions.

Self-checksumming: Packers that employ self-checksumming binaries take a checksum over the program's code bytes and then recalculate that checksum at run-time to detect modifications to portions of the program's code or data. Our instrumentation algorithm hides the modifications it makes to the binary with the CAD-compensation techniques that we described in detail in Section 5.3. We evaluated the efficiency of these techniques by applying them to non-defensive binaries in in Section 5.4. This study of packed binaries demonstrates that these techniques can stealthily instrument defensive binaries by successfully instrumenting ASProtect, Yoda's Protector, and PECompact, each of which applies self-checksumming to its code (rows D6, D7).

Anti-OEP finding: Finding the original entry point of a packed binary is an important step towards creating unpacked versions of packed binaries (see Section 2.2 for details on binary unpacking techniques). Malware analysts are often interested in creating unpacked binaries because they can then build structural analyses by applying static analysis tools to these unpacked binaries. Our techniques create structural analyses of packed binaries without needing to create an unpacked version. For this reason, we make no attempt to identify the original entry point of packed programs.

Payload-code modification: Some packed binaries modify the payload code that they pack as a second means of defending against binary unpacking techniques. In particular, they wish to prevent the analyst from bypassing their defensive packer code, and do so by hooking the packed payload code with control transfers that invoke packer code. We are un-

aware of any generally applicable techniques that automate a solution to the problem of removing these payload-code modifications. Fortunately, these techniques do not present a problem for our analysis and instrumentation techniques, as we analyze packed binaries as a whole and have no need to separate packer code from malicious payload code.

6.2 Malware Analysis Results

Analysts at large security companies receive tens of thousands of new malware samples each day [86] and must process them efficiently and safely to determine which samples are of greatest interest. We use SR-Dyninst to create a customizable *malware analysis factory* that performs efficient batch-processing of malware binaries in an isolated environment. Our factory leverages our instrumentation and analysis techniques and is therefore able to find and analyze code that is beyond the reach of static or dynamic analysis alone. In its default configuration, our factory processes a collection of malware binaries, and for each one it outputs program CFGs that are annotated with code-coverage information, logs of invoked Windows API calls, and a stackwalk at the program's first network communication.

Malware analysts can easily customize our factory to achieve their desired analysis goals. Analyzing defensive malware binaries using SR-Dyninst requires no more skill from the analyst than performing the same task with Dyninst 7.0 on a conventional binary. In building analysis tools for Dyninst, malware analysts can take advantage of Dyninst's full analysis and instrumentation capabilities, for example, its ability to walk the program's call stacks, analyze and instrument functions, loops, basic blocks, and instructions, and much more.

To use Dyninst, the analyst writes a tool that links against the Dyninst library. The default configuration of our malware analysis factory is based

on a user tool that we wrote ourselves and whose main task is to perform code coverage over the malware binary. More precisely, our tool uses Dyninst to instrument every basic block in the malware program, (both statically present and dynamically unpacked blocks) and to remove the instrumentation from a block once it has executed. Our tool halts the malware at the point that it attempts its first network communication, exits, or reaches a 30-minute timeout. At that time, our tool uses Dyninst to walk the malware's call stacks, and prints out the details of each stack frame. Our tool also prints out Dyninst's control-flow graph of the binary in Graphviz format [41]. Each node in the CFG represents a basic block in the binary and is labeled with its address, though our CFG also includes nodes for the first blocks of invoked system library functions that it labels with their function names. We also encode our code-coverage results into this graph, by coloring basic blocks to distinguish between basic blocks that executed and blocks that did not. Writing user tools for Dyninst is quite simple; our tool specifies its code-coverage instrumentation in only fifty lines, and most of the remainder of the program is dedicated to producing our labeled CFG in Graphviz format.

We set up our malware analysis factory on an air-gapped system with a 32-bit Intel-x86 processor running Windows XP with Service Pack 2, inside of VMWare Server. In the host OS we wrote a script that processes a user-specified list of malware executables one at a time. For each malware binary in this list, our script hands a malware binary to our Windows VM, which is configured to automaticaly start execution of the malware under control of the SR-Dyninst instrumenter. When the malware sample either terminates, attempts to communicate over the network, or times out, our script stops the VM, extracts the output log from the VM's virtual disk file, and resets the VM to a stable checkpoint.

We analyzed 200 malware samples that were given to us by Offensive Computing [27] in December 2009. Our tool detected code unpacking in

27% of the samples, code overwrites in 16%, and signal-based control flow in 10%. 33% of the malicious code analyzed by our hybrid techniques was not part of the dynamic execution trace and would not have been found by dynamic analysis techniques. In addition to these statistics, for each malware sample, our factory produced an annotated CFG of its basic blocks, a log of the Windows API calls that it invoked, and a stackwalk at the program's first network communication.

As an example of the kinds of results produced by our factory, in Figure 6.1 and Table 6.1 we show two of its analysis products for the Conficker A malware binary. In Figure 6.1a we show our annotated CFG of the Conficker A binary in its entirety, while Figure 6.1b shows an excerpt of that graph, highlighting the fact that SR-Dyninst has captured static and dynamic code, both code in the executable and invocations of Windows system-library functions, and both code that has executed and code that has not executed but that may be of interest to the analyst. As seen in Figure 6.1a, the 30% of the blocks that we analyzed with our hybrid analysis techniques did not execute, and therefore would not have been detected by a dynamic analysis of Conficker.

Figure 6.1 shows our traversal of Conficker's call stacks at Conficker's first call to the `select` routine. The contextual information provided by stackwalks is extremely helpful to a human analyst, as is well-attested by users of interactive debugger tools such as GDB. As seen in this stack trace, we are able to identify the stack frames of functions in Conficker for which we lack symbol information, which is an important benefit of our analysis capabilities. While existing stackwalking techniques are accurate only for statically analyzable code [69], our hybrid analyses enable accurate stackwalking of obfuscated malware by virtue of having analyzed all of the code that could be executing at any given time.

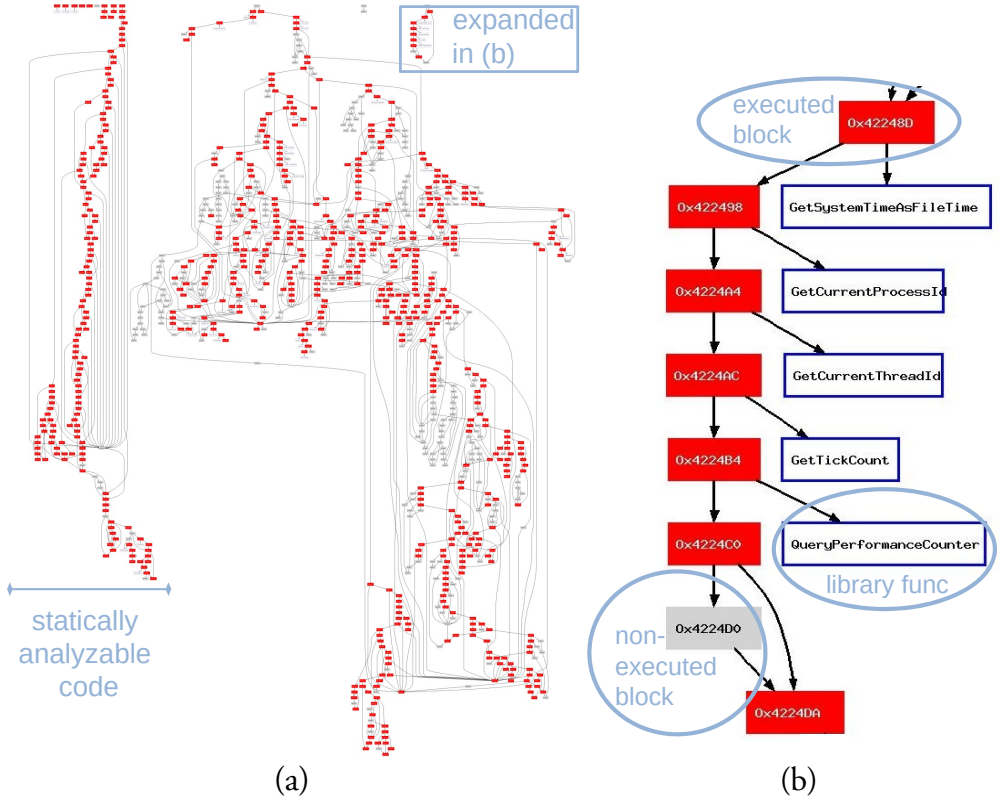


Figure 6.1: Two views of Conficker A's control flow graph. The CFG in part (a) can be explored in an interactive viewer, as shown in the excerpt from part (b). Conficker's statically analyzable unpacker code is marked in Figure (a), the rest of the code in the CFG is dynamically unpacked. As seen in part (b), basic blocks that have executed are colored in red, non-executed blocks are in grey, and system-library functions are labeled in white rectangles.

Conficker's communications thread

top	pc=7c901231 DbgBreakPoint_7c901230 in ntdll.dll	[Win DLL]
	pc=10003c83 DYNbreakPoint_10003c70 in dyn_RT.dll	[Instrument.]
	pc=100016f7 DYNstopThread_10001670 in dyn_RT.dll	[Instrument.]
	pc=71ab2dc0 select_71ab2dc0 in WS2_32.dll	[Win DLL]
base	pc=41f134 nosym1f058_41f058 in cf.exe	[Conficker]

Conficker's main thread

top	pc=7c90eb8f KiFastSystemCall_7c90eb8b in ntdll.dll	[Win DLL]
	pc=7c80d85c ZwDelayExecution_7c90d84a in ntdll.dll	[Win DLL]
	pc=7c8023ed SleepEx_7c80239c in kernel23.dll	[Win DLL]
	pc=7c802451 Sleep_7c802442 in kernel32.dll	[Win DLL]
	pc=41f20e nosym1f1cd_41f1cd in cf.exe	[Conficker]
	pc=41a640 nosym1a592_41a592 in cf.exe	[Conficker]
	pc=41a6eb nosym1a6d7_41a6d7 in cf.exe	[Conficker]
base	pc=4226b0 start_426f70 in cf.exe	[Conficker]

Table 6.1: An SR-Dyninst stack walk taken when the Conficker A binary executes Winsock's `select` routine. We show the call stacks used by both of Conficker's threads. The stack walks includes frames from our instrumentation, `select`, and Conficker.

7 CONCLUSION

State of the art analysis techniques for malware executables lag significantly behind their counterparts for compiler-generated binaries. This difference is due to the fact that 90% of malware executables actively resist analysis [17]. In particular, most malware binaries obfuscate their code to impede the static analyses by which binary analysis tools recover structural information from binary code. Furthermore, many malware binaries employ defenses like self-checksumming [3] to counter the dynamic techniques by which analysis tools monitor and control the program's execution.

The work of this dissertation allows program-binary analysts to study defensive malware with many of the same foundational techniques that they have at their disposal when they study non-defensive binaries. In particular, we provide techniques to build structural analyses of defensive code, prior to its execution. We also provide stealthy instrumentation techniques so that analysts can control malware execution based on that structural analysis without triggering checks based on self-checksumming or related techniques. We proceed by listing the primary contributions of this work and then discussing promising directions for future research.

7.1 Contributions

This dissertation develops the following techniques to analyze defensive malware binaries.

- We develop static techniques for heavily obfuscated code, thereby building foundational control- and data-flow analyses such as control-flow graphs and binary slices [23].

- We develop the following dynamic techniques to find and analyze defensive malware code before it executes: dynamic capture instrumentation that discovers statically unreachable code, overwrite handling techniques that detect and respond to code overwrites, and handler-interception techniques that detect and intercept signal and exception-based control flow. These dynamic techniques trigger additional parsing at entry points into code that is statically unreachable.

- By developing a hybrid analysis algorithm that combines our static and dynamic techniques, we find and analyze code that is beyond the reach of either static or dynamic analysis alone, and do so before this code executes.

- We provide a binary instrumentation technique that modifies the malware's code in accordance with the user's expectations while hiding its impact from the program. In particular, our stealthy instrumentation techniques prevent defensive malware from detecting our changes to its code, and from detecting that we allocate extra space for instrumentation code in the malware's address space.

We demonstrate the utility of these contributions with the following results:

- We show that our hybrid techniques enable analysis-guided instrumenters such as Dyninst and Vulcan [113] to operate on malware by implementing our techniques in SR-Dyninst, which builds on the code of Dyninst 7.0. By creating SR-Dyninst, we allow Dyninst users to build tools that analyze defensive malware code in exactly the same way that they analyze non-defensive binaries. We demonstrate the efficacy of our tool both on real and representative synthetic malware samples that are highly defensive.

- We built a customizable malware analysis factory on top of SR-Dyninst [103]. Our factory performs batch-processing of malware binaries in an isolated environment, producing customizable reports on the structure and behavior of the malware. This factory allows security companies to safely and efficiently process new malware samples in an automated way. The reports produced by our tool also help these companies determine which samples are of greatest interest and provide detailed information to help them understand these samples.

- We customized our malware analysis factory to study the most prevalent defensive techniques used by malware binaries. To determine what those techniques are, we applied our factory to binaries created by the packer toolkits that malware authors most often use to add defensive techniques to their binaries [17]. Our factory produces reports on the defensive techniques used by these binaries. Based on these reports, we catalog these techniques and report on their relative frequency and impact.

Some of our contributions to malware analysis and instrumentation have applications to other types of binary code. In particular, our stealthy instrumentation techniques are safer and more efficient than instrumentation techniques used by other prevalent instrumenters, such as PIN [70] and Dyninst 7.0 [57]. Owing to these benefits of our stealthy instrumentation techniques, our implementation of these techniques will replace Dyninst 7.0's instrumentation engine in the next Dyninst release. Another technique that is directly applicable to non-defensive binary code is our dynamic capture instrumentation, as non-defensive code may also contain many indirect control transfers whose targets are not statically analyzable. These instrumentation techniques would be similarly helpful in analyzing programs that generate code at run-time, just in time for it to execute.

An example of such a program is the Java Virtual Machine (JVM), which compiles java bytecode into machine-language code sequences at run-time and then executes them. Our code-overwrite handling techniques are also useful on this class of programs, as their dynamically generated code sequences are typically stored in a cache and are overwritten by new code once the cache is full. Finally, our exception-monitoring techniques also allow us to dynamically resolve exception-based control flow in non-defensive programs, and update our analysis in response to these exceptions.

7.2 Future Directions

We see many opportunities to build on this work in ways that will further aid efforts to analyze defensive malware. We present two of these opportunities below.

- Building on the notion that our techniques are useful for broad categories of programs, we intend to develop different modes of analysis and instrumentation for different classes of program binaries. At present, our SR-Dyninst implementation allows for two modes of analysis and instrumentation. The "normal" mode performs Dyninst's one-time static analysis of the code and instruments without performing CAD and AVU compensation. We have added a "defensive" mode that incorporates all of the techniques outlined in this dissertation. We intend to add an intermediate mode for conventional binaries that uses our dynamic capture instrumentation to resolve indirect control transfers at run-time, triggering additional parsing at dynamically identified control flow targets. In addition to these three modes, we want to develop techniques that will automatically select the appropriate mode based on characteristics of the program binary.

- Our current SR-Dyninst implementation uses the OS-provided debugging interface to monitor program binaries; replacing this dependence with virtual-machine-monitoring (VMM) techniques would bring two significant advantages. First, we would be able to fully analze malware that modifies the operating system with rootkit components. The OS-provided debugger interface only allows us monitor the user-space components of a program, but VMM techniques sit outside of the guest operating system and monitor it was a whole, and can therefore monitor rootkits. Second, the use of VMM techniques would allow us to more fully hide our presence from the monitored program. At present, a malware program's rootkit components can trivially detect our use of the debugger interface. Furthermore, despite our best efforts to hide our use of the debugger interface, there are far more ways for a program to detect a debugger process [42] than there are ways to detect the presence of VMM techniques [43], as these have a much smaller footprint in the guest virtual machine.

- This work represents a step towards our larger goal of rewriting defensive malware binaries into working non-defensive binaries that can be readily analyzed and instrumented by normal binary analysis tools. Our goal is similar to that of prior tools that have attempted to create unpacked binaries, as described in Section 2.2. However, the crucial difference between our techniques is that we do not attempt to bypass packer metacode when we rewrite these binaries. This difference means that defensive techniques that make packer metacode difficult to bypass are irrelevant to our approach, since we will include all of the packed binary's code in our rewritten version of the binary. Analysts have not adopted the approach that we propose because the changes they would make to the program binary could be detected by the packer's metacode. However, our

stealthy instrumentation techniques will allow us to modify the binary while hiding these modifications from its code.

The first step in rewriting a defensive malware binary is to build up a detailed structural analysis of its code, and this dissertation provides techniques that do so. The next step is to use this analysis to rewrite the binary, for example, by writing dynamically unpacked code into the binary, and adding symbol information to indicate the locations of the program's functions. Since the Dyninst 7.0 instrumenter also includes the ability to rewrite program binaries, we can naturally extend SR-Dyninst to take advantage of Dyninst's rewriting capabilities.

REFERENCES

[1] Anckaert, Bertrand, Matias Madou, and Koen De Bosschere. 2007. A model for self-modifying code. In *Workshop on information hiding*, 232–248. Alexandria, VA.

[2] Aspack Software: Aspack and Asprotect. `http://www.aspack.com/`.

[3] Aucsmith, David. 1996. Tamper resistant software: An implementation. In *Workshop on information hiding*, 317–333. Cambridge, U.K.

[4] Babic, Domagoj, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation. In *International symposium on software testing and analysis (issta)*. Toronto, Canada.

[5] Balakrishnan, Gogul, and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *Conference on compiler construction (cc)*, 5–23. New York, NY.

[6] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Symposium on operating systems principles*. Bolton Landing, NY.

[7] Bayer, Ulrich, Andreas Moser, Christopher Kruegel, and Engin Kirda. 2006. Dynamic analysis of malicious code. *Journal in Computer Virology* 2(1):66–77.

[8] Bellard, Fabrice. 2005. QEMU, a fast and portable dynamic translator. In *Usenix annual technical conference*. Anaheim, CA.

[9] Bernat, Andrew R. 2012. Abstract, safe, timely, and efficient binary modification. Ph.D. thesis, Department of Computer Sciences, University of Wisconsin.

[10] Bernat, Andrew R., and Barton P. Miller. 2011. Anywhere, Any Time Binary Instrumentation. In *Workshop on program analysis for software tools and engineering (paste)*. Szeged, Hungary.

[11] Bernat, Andrew R., Kevin A. Roundy, and Barton P. Miller. 2011. Efficient, Sensitivity Resistant Binary Instrumentation. In *International symposium on software testing and analysis (issta)*. Toronto, Canada.

[12] BitDefender. 2007. BitDefender anti-virus technology. White Paper.

[13] Böhne, Lutz, and Thorsten Holz. 2008. Pandora's bochs: Automated malware unpacking. Master's thesis, University of Mannheim.

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

[15] Bruschi, Danilo, Lorenzo Martignoni, and Mattia Monga. 2007. Code normalization for self-mutating malware. *IEEE Security and Privacy* 5(2).

[16] Buck, Bryan R., and Jeffrey K. Hollingsworth. 2000. An api for runtime code patching. *Journal of High Performance Computing Applications* 14(4).

[17] Bustamante, Pedro. 2008. Malware prevalence. Panda Research web article.

[18] ———. 2008. Packer (r)evolution. Panda Research web article.

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

[20] Chiang, Ken, and Levi Lloyd. 2007. A case study of the rustock rootkit and spam bot. In *First conference on hot topics in understanding botnets*. Cambridge, MA.

[21] Chow, Jim, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *Usenix annual technical conference*. Boston, MA.

[22] Christodorescu, Mihai, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. 2005. Malware normalization. Tech. Rep. 1539, Computer Sciences Department, University of Wisconsin.

[23] Cifuentes, Cristina, and Antoine Fraboulet. 1997. Intraprocedural static slicing of binary executables. In *International conference on software maintenance (icsm)*. Los Alamitos, CA.

[24] Cifuentes, Cristina, and Mike Van Emmerik. 1999. Recovery of jump table case statements from binary code. In *International workshop on program comprehension (icpc)*. Pittsburgh, PA.

[25] Collake, Jeremy. PECompact executable compressor. `http://www.bitsum.com/pecompact.php`.

[26] Collberg, Christian, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Symposium on principles of programming languages (popl)*. San Diego, CA.

[27] Computing, Offensive. 2009. `http://www.offensivecomputing.net`.

[28] Coogan, Kevin, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. 2009. Automatic static unpacking of malware binaries. In *Working conference on reverse engineering*. Antwerp, Belgium.

[29] Coward, P.D. 1988. Symbolic execution systems-a review. *Software Engineering Journal* 3(6).

[30] Danehkar, Ashkbiz. Yoda's protector. `http://sourceforge.net/projects/yodap/`.

[31] ———. 2005. Inject your code into a portable executable file. `http://www.codeproject.com/KB/system/inject2exe.aspx`.

[32] Dark Paranoid. 1998. Engine of eternal encryption. *Moon Bug* 7.

[33] De Bus, Bruno, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. 2004. The design and implementation of FIT: a flexible instrumentation toolkit. In *Workshop on program analysis for software tools and engineering (paste)*.

[34] Debray, Saumya, and William Evans. 2002. Profile-guided code compression. In *Conference on programming language design and implementation (pldi)*. Berlin, Germany.

[35] Debray, Saumya, and Jay Patel. 2010. Reverse engineering self-modifying code: Unpacker extraction. In *Working conference on reverse engineering*. Boston, MA.

[36] Dimitrov, Martin, and Huiyang Zhou. 2009. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *Conference on architectural support for programming languages and operating systems (asplos)*. Washington, D.C.

[37] Dinaburg, Artem, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware analysis via hardware virtualization extensions. In *Conference on computer and communications security*. Alexandria, VA.

[38] Dunlap, George W., Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. 2002. Revirt: Enabling intrusion analysis

through virtual-machine logging and replay. In *Symposium on operating systems design and implementation*. Boston, MA.

[39] Dux, Bradley, Anand Iyer, Saumya Debray, David Forrester, and Stephen Kobourov. 2005. Visualizing the behavior of dynamically modifiable code. In *International workshop on program comprehension (icpc)*. St. Louis, MO.

[40] Dwing. (Win)Upack. `http://wex.cn/dwing/mycomp.htm`.

[41] Ellson, John, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and dynagraph - static and dynamic graph drawing tools. In *Graph drawing software*.

[42] Falliere, Nicolas. 2007. Windows anti-debug reference. Infocus web article.

[43] Ferrie, Peter. 2006. Attacks on virtual machine emulators. In *Association of anti-virus asia researchers international conference*. Auckland, New Zealand.

[44] ———. 2008. Anti-unpacker tricks. In *International caro workshop*. Amsterdam, The Netherlands.

[45] ———. 2008. Anti-unpacker tricks - part one. *Virus Bulletin*.

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

[47] FSG: Fast Small Good Packer. 2005. `http://www.xtreeme.prv.pl/`.

[48] Giffin, Jonathon T., Mihai Christodorescu, and Louis Kruger. 2005. Strengthening software self-checksumming via self-modifying code. In *Annual computer security applications conference (acsac)*. Tucson, AZ.

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

[50] Graf, Tobias. 2005. Generic unpacking - how to handle modified or unknown PE compression engines? In *Virus bulletin conference*. Dublin, Ireland.

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

[52] ———. 2011. The IDA Pro disassembler and debugger. DataRescue. Version 6.2 `http://www.hex-rays.com/idapro/`.

[53] Guo, Fanglu, Peter Ferrie, and Tzicker Chiueh. 2008. A study of the packer problem and its solutions. In *Symposium on recent advances in intrusion detection (raid)*. Cambridge, MA: Springer Berlin / Heidelberg.

[54] Harris, Luane C., and Barton P. Miller. 2005. Practical analysis of stripped binary code. *SIGARCH Computer Architecture News* 33(5).

[55] Hex-Rays Decompiler. 2011. `http://hex-rays.com`. Version 1.6.

[56] Hind, Michael, and Anthony Pioli. 2000. Which pointer analysis should I use? In *International symposium on software testing and analysis (issta)*. Portland, OR.

[57] Hollingsworth, Jeffrey K., Barton P. Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable high performance computing conference*. Knoxville, TN.

[58] Hunt, Galen, and Doug Brubacher. 1999. Detours: Binary interception of win32 functions. In *Usenix windows nt symposium*. Seattle, WA.

[59] Institute, Ponemon. 2011. Second annual cost of cyber crime study.

[60] Kang, Ming G., Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Workshop on recurring malcode*. Alexandria, VA.

[61] Kiss, Akos, Jutid Jasz, Gabor Lehotai, and Tibor Gyimothy. 2003. Interprocedural static slicing of binary executables. In *Source code analysis and manipulation*. Amsterdam, The Netherlands.

[62] Kruegel, Christopher, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Usenix security symposium*. San Diego, CA.

[63] Lakhotia, Arun, Eric Uday Kumar, and M. Venable. 2005. A method for detecting obfuscated calls in malicious binaries. *Transactions on Software Engineering* 31(11).

[64] Larus, James R., and Eric Schnarr. 1995. Eel: machine-independent executable editing. *ACM SIGPLAN Notices* 31(11).

[65] Laurenzano, M., M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient static binary instrumentation for linux. In *Ieee international symposium on performance analysis of systems and software (ispass)*. White Plains, NY, USA.

[66] Lim, Junghee, and Thomas Reps. 2008. A system for generating static analyzers for machine instructions. In *International conference on compiler construction (cc)*. Budapest, Hungary.

[67] Lindholm, Tim, and Frank Yellin. 1999. *Java virtual machine specification*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[68] Linn, Cullen, and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Conference on computer and communications security*. Washington, D.C.

[69] Linn, Cullen, Saumya Debray, Gregory Andrews, and Benjamin Schwarz. 2004. Stack analysis of x86 executables. Manuscript.

[70] Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Conference on programming language design and implementation (pldi)*. Chicago, IL.

[71] Madou, Matias, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *Acm workshop on digital rights management*. Alexandria, VA.

[72] Maebe, Jonas, and Koen De Bosschere. 2003. Instrumenting self-modifying code. In *Workshop on automated and algorithmic debugging*. Ghent, Belgium.

[73] Maebe, Jonas, Michiel Ronsse, and Koen De Bosschere. 2002. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Workshop on binary translation held in conjunction with the conference on parallel architectures and compilation techniques (pact)*. Charlottesville, VA.

[74] Martignoni, Lorenzo, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In

Annual computer security applications conference (acsac). Miami Beach, FL.

[75] Miller, Barton P., Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of unix utilities. *Communications of the ACM* 33(12).

[76] Moser, Andreas, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Symposium on security and privacy*. Oakland, CA.

[77] ———. 2007. Limits of static analysis for malware detection. In *Annual computer security applications conference (acsac)*. Miami Beach, FL.

[78] Muth, Robert, and Saumya Debray. 2000. On the complexity of flow-sensitive dataflow analyses. In *Symposium on principles of programming languages (popl)*. Boston, MA.

[79] Nanda, Susanta, Wei Li, Lap-Cung Lam, and Tzi cker Chiueh. 2006. BIRD: Binary interpretation using runtime disassembly. In *Symposium on code generation and optimization (cgo)*. New York, NY.

[80] Nethercote, Nicholas, and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Conference on virtual execution environments (vee)*. San Diego, CA.

[81] ———. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Conference on programming language design and implementation (pldi)*. San Diego, CA.

[82] Nguyen, Anh M., Nabil Schear, HeeDong Jung, Apeksha Godiyal, Sam T. King, and Hai Nguyen. 2009. Mavmm: A lightweight and purpose-built vmm for malware analysis. In *Annual computer security applications conference (acsac)*. Honolulu, HI.

[83] Norton. 2010. Cybercrime report: The human impact.

[84] ———. 2011. Cybercrime report.

[85] Oberhumer, Markus F.X.J., Laszlo Molnar, and John F. Reiser. 2012. UPX: the Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`.

[86] Panda Security. 2011. Annual report Pandalabs 2010.

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

[88] ———. 2011. StackwalkerAPI programmer's guide. Version 2.0 `http://www.paradyn.org/html/manuals.html`.

[89] Payne, Bryan D. 2011. LibVMI `http://vmitools.sandia.gov/`. Version 0.6.

[90] Payne, Bryan D., Martim Carbone, and Wenke Lee. 2007. Secure and flexible monitoring of virtual machines. In *Annual computer security applications conference (ACSAC 2007)*. Miami Beach, FL.

[91] Peiser, Sean, Matt Bishop, Sidney Karin, and Keith Marzullo. 2007. Analysis of computer intrusions using sequences of function calls. *IEEE Trans. Dependable Secur. Comput.* 4(2):137–150.

[92] Perriot, Frederic, and Peter Ferrie. 2004. Principles and practise of x-raying. In *Virus bulletin conference*. Chicago, IL.

[93] Popov, Igor, Saumya Debray, and Gregory Andrews. 2007. Binary obfuscation using signals. In *Usenix security symposium*. Boston, MA.

[94] Porras, Phillip, Hassen Saidi, and Vinod Yegneswaran. 2007. A multi-perspective analysis of the storm (peacomm) worm. Tech. Rep., SRI International.

[95] ———. 2009. A foray into conficker's logic and rendezvous points. Tech. Rep., SRI International.

[96] Prakash, Chandra. 2007. Design of x86 emulator for generic unpacking. In *Assocation of anti-virus asia researchers international conference*. Seoul, South Korea.

[97] Quinlan, Dan. 2000. Rose: Compiler support for object-oriented frameworks. In *Conference on parallel compilers (cpc2000)*. Aussois, France.

[98] Quist, Danny, and Valsmith. 2007. Covert debugging: Cirvumventing software armoring techniques. In *Blackhat usa*. Las Vegas, NV.

[99] Quist, Danny A., and Lorie M. Liebrock. 2009. Visualizing compiled executables for malware analysis. In *Workshop on visualization for cyber security*. Atlantic City, NJ.

[100] Rosenblum, Nathan E., Gregory Cooksey, and Barton P. Miller. 2008. Virtual machine-provided context sensitive page mappings. In *Conference on virtual execution environments (vee)*. Seattle, WA.

[101] Rosenblum, Nathan E., Barton P. Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *Workshop on program analysis for software tools and engineering (paste)*. Toronto, Canada.

[102] Rosenblum, Nathan E., Xiaojin Zhu, Barton P. Miller, and Karen Hunt. 2008. Learning to analyze binary computer code. In *Conference on artificial intelligence (aaai)*. Chicago, IL.

[103] Roundy, Kevin A., and Barton P. Miller. 2010. Hybrid analysis and control of malware. In *Symposium on recent advances in intrusion detection (raid)*. Ottawa, Canada.

[104] Royal, Paul, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual computer security applications conference (acsac)*. Miami Beach, FL.

[105] Russinovich, Mark, and Bryce Cogswell. 1997. Windows NT system call hooking. *Dr. Dobb's Journal* 22(1).

[106] Rutkowska, Joanna, and Alexander Tereshkin. 2007. IsGameOver(), anyone? In *Blackhat usa*. Las Vegas, NV.

[107] Schwarz, Benjamin, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Working conference on reverse engineering*. Richmond, VA.

[108] Schwarz, Benjamin, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *2001 workshop on binary rewriting*. Barcelona, Spain.

[109] Scott, Dana, and Christopher Strachey. 1971. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab.

[110] Sharif, Monirul, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2008. Impeding malware analysis using conditional code obfuscation. In *Network and distributed system security symposium (ndss)*. San Diego, CA.

[111] Shende, S., and A. D. Malony. 2006. The tau parallel performance system. *International Journal of High Performance Computing Applications* 20(2):287–311.

[112] Sites, Richard L., Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary translation. *Communications of the ACM* 36(2).

[113] Srivastava, Amitabh, Andrew Edwards, and Hoi Vo. 2001. Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research.

[114] Srivastava, Amitabh, and Alan Eustace. 1994. ATOM: a system for building customized program analysis tools. In *Conference on programming language design and implementation (pldi)*. Orlando, FL.

[115] Stepan, Adrian E. 2005. Defeating polymorphism: beyond emulation. In *Virus bulletin conference*. Dublin, Ireland.

[116] Stewart, Joe. 2007. Unpacking with ollybone. Online tutorial. `http://www.joestewart.org/ollybone/tutorial.html`.

[117] Szappanos, Gabor. 2007. Exepacker blacklisting. *Virus Bulletin*.

[118] Theiling, Henrik. 2000. Extracting safe and precise control flow from binaries. In *Conference on real-time computing systems and applications*. Cheju Island, South Korea.

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

[120] Troger, Jens, and Cristina Cifuentes. 2002. Analysis of virtual method invocation for binary translation. In *Working conference on reverse engineering*. Richmond, VA.

[121] Tucek, Joseph, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. 2007. Sweeper: A lightweight end-to-end system for defending against fast worms. In *EuroSys*. Lisbon, Portugal.

[122] Van Emmerik, Mike, and Trent Waddington. 2004. Using a decompiler for real-world source recovery. In *Working conference on reverse engineering (wcre)*. Delft, The Netherlands.

[123] Vigna, Giovanni. 2007. Static disassembly and code analysis. In *Malware detection*, vol. 35 of *Advances in Information Security*. Springer.

[124] Willems, Carsten, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using CWSandbox. In *Symposium on security and privacy*. Oakland, CA.

[125] Wurster, Glenn, P. C. Van Oorschot, and Anil Somayaji. 2005. A generic attack on checksumming-based software tamper resistance. In *Symposium on security and privacy*. Oakland, CA.

[126] Yason, Mark Vincent. 2007. The art of unpacking. In *Blackhat usa*. Las Vegas, NV.

[127] Yegneswaran, Vinod, Hassen Saidi, and Phillip Porras. 2008. Eureka: A framework for enabling static analysis on malware. Tech. Rep. SRI-CSL-08-01, SRI International.

[128] Yuschuk, Oleh. 2000. OllyDbg. Version 1.10 `http://www.ollydbg.de`.

[129] Zhou, Jingyou, and Giovanni Vigna. 2004. Detecting attacks that exploit application-logic errors through application-level auditing. In *Annual computer security applications conference (acsac)*. Tucson, AZ, USA.