Hybrid Analysis and Control of Malware

Kevin A. Roundy and Barton P. Miller

Computer Sciences Department
University of Wisconsin
{roundy,bart}@cs.wisc.edu

Abstract. Malware attacks necessitate extensive forensic analysis efforts that are manual-labor intensive because of the analysis-resistance techniques that malware authors employ. The most prevalent of these techniques are code unpacking, code overwriting, and control transfer obfuscations. We simplify the analyst's task by analyzing the code prior to its execution and by providing the ability to selectively monitor its execution. We achieve pre-execution analysis by combining static and dynamic techniques to construct control- and data-flow analyses. These analyses form the interface by which the analyst instruments the code. This interface simplifies the instrumentation task, allowing us to reduce the number of instrumented program locations by a hundred-fold relative to existing instrumentation-based methods of identifying unpacked code. We implement our techniques in SD-Dyninst and apply them to a large corpus of malware, performing analysis tasks such as code coverage tests and call-stack traversals that are greatly simplified by hybrid analysis.

Key words: malware analysis, forensics, hybrid, de-obfuscation, packed code, self-modifying code, obfuscated code

1 Introduction

Malicious software infects computer systems at an alarming rate, causing economic damages that are estimated at more than ten billion dollars per year [1]. 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 the malware so they can recover infected systems and construct defenses. Thus, a primary goal of malware authors is to make these tasks as difficult and resource intensive as possible. This explains why 90% of malware binaries employ *analysis-resistance* techniques [9], the most prevalent of which are the run-time unpacking of compressed and encrypted code, run-time modifications to existing code, and obfuscations of control transfers in the code. Security companies detect thousands of new malware samples each day [45], yet despite the importance and scale of this problem, analysts continue to resort to manual-labor intensive methods.

Analysts accomplish their tasks by studying the malware's overall structure to identify the relevant code and then analyzing it thoroughly. Unfortunately, analysis-resistance techniques force the analyst out of the usual mode of binary

analysis, which involves statically analyzing the binary prior to instrumenting regions of interest and performing a controlled execution of the program. Instead, the analyst must execute the malicious code to analyze it, as static analysis can fail to analyze dynamically generated, modified, and obfuscated code. The analyst must therefore construct a virtual environment that allows the malware to interact with other hosts and that is sufficiently convincing that the malware will display its normal behavior while executing in isolation from the outside world. Many analysts prefer the analyze-then-execute model and therefore resort to expending considerable manual effort to strip analysis-resistance features from malicious binaries [12, 40].

The goal of our research is to simplify malware analysis by enabling a return to the traditional analyze-then-execute model, which has the benefit of bringing the malicious code under the analyst's control before it executes. We address these goals by combining static and dynamic techniques to construct and maintain the control- and data-flow analyses that form the interface through which the analyst understands and instruments the code. A key feature of our approach is its ability to update these analyses to include dynamically unpacked and modified code before it executes. Our work makes the following contributions:

- Pre-execution analysis and instrumentation makes it possible for the analyst to control the execution of malicious code. For example, our work allows interactions with other infected hosts to be simulated through instrumentation's ability to patch the program, removing the need for complex virtual environments involving multiple hosts. Additionally, our work can complement a virtualization strategy by identifying and disabling the malware's attempts to detect its virtual-machine environment [36].
- We give the analyst the ability to instrument malware intuitively and efficiently by providing data-flow analysis capabilities and a control flow graph (CFG) as an interface to the code. For example, the CFG allows us to find transitions to dynamically unpacked code by instrumenting the program's statically unresolved control transfers (see Section 5). By contrast, prior instrumentation-based unpacking tools did not maintain a CFG and therefore had to monitor all of the program's control transfers and memory writes to detect the execution of code that is written at run-time [23, 41]. We achieve a hundred-fold reduction in the number of monitored program locations.
- Our structural analysis allows analysts to be selective in the components they monitor, the operations in those components that they select, and in the granularity of data they collect. Current tools that can monitor analysis-resistant malware 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 [17, 36], or coarse grained traces (e.g., at interactions with the OS) [52]. These tools either bog the analyst down with irrelevant information (a significant problem for inexperienced analysts [42]), or can only give a sketch of the program's behavior.
- 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 combinations of static and dynamic analysis only operate on non-defensive code, and only find and disassemble the code [34] or produce their analysis results only after the program has fully executed [28].

Analysts have controlled and monitored malicious code either by executing the malware in a process that they control through the debugger interface [44], or by executing the malware in a virtual machine [36]. There are advantages to both approaches. The debugger approach makes it easy to collect process information and intercept events, and allows for the creation of lightweight tools that do not have to worry about anti-emulation techniques [36]. Among the benefits of virtual machines are that they isolate the underlying system from the malware's effects, they provide the ability to revert the machine to a clean state or to a decision point, and allow for stealthy monitoring of the program's execution [17, 36]. While in this paper we demonstrate an instrumentation and analysis tool that executes malicious processes through the debugger interface, our techniques are orthogonal to this choice and benefit both scenarios. For example, in the former case, pre-execution analysis and control allows the analyst to limit the damage done to the system, while the latter case benefits from the ability to detect and disable anti-emulation techniques.

Our analysis and instrumentation tool is not the first to analyze code prior to its execution (e.g., Dyninst [22], Vulcan [48]), but existing tools rely exclusively on static analysis, which can produce incomplete information even for binaries that are generated by standard compilers. Despite recent advances in identifying functions in stripped compiler-generated binaries, on the average, 10% of the functions generated by some compilers cannot be recognized by current techniques [43], and even costly dataflow analyses such as pointer aliasing may be insufficient to predict the targets of pointer-based control transfers [5, 21].

Most malware binaries make analysis and control harder by employing the analysis-resistance techniques of code packing, code overwriting, and control transfer obfuscations. *Code packing* techniques, wherein all or part of the binary's malicious code is compressed (or encrypted) and packaged with code that decompresses the malicious payload into the program's address space at runtime, are present in 75% of all malware binaries [8, 50]. Dealing with dynamic code unpacking is complicated by programs that unpack code in stages, by the application of multiple code-packing tools to a single malicious binary, and by a recent trend away from well-known packing tools, so that most new packed binaries use custom packing techniques [9, 10].

To further complicate matters, many packing tools and malicious programs *overwrite* code at run-time. While code unpacking impacts static analysis by making it incomplete, code overwriting makes the analysis invalid *and* incomplete. A static analysis may yield little information on a self-modifying program, as potentially little of the code is exposed to analysis at any point in time [2].

Malware often uses *control-transfer obfuscations* to cause static analysis algorithms to miss and incorrectly analyze large swaths of binary code. In addition to the heavy use of indirect control transfers, obfuscated binaries commonly include

non-conventional control transfer sequences (such as the use of the return instruction as an indirect jump), and signal- and exception-based control transfers [18]. Additionally, malicious binaries and packers often contain hand-written assembly code that by its nature contains more variability than compiler-generated code, causing problems for most existing code-identification strategies, as they depend on the presence of compiler-generated instruction patterns [24, 43].

We analyze binaries by first building a CFG of the binary through static parsing techniques. As the binary executes, we rely on dynamic instrumentation and analysis techniques to discover code that is dynamically generated, hidden by obfuscations, and dynamically modified. We then re-invoke our parsing techniques to update our CFG of the program, identifying the new code and presenting the updated CFG to the analyst. The structural information provided by our analysis allows us to discover new code by instrumenting the program lightly, only at control transfer instructions whose targets cannot be resolved through static analysis, making our tool's execution time comparable to that of existing unpacking tools despite the additional cost that we incur to provide an updated CFG. (see Section 8).

Other analysis resistance techniques that can be employed by malicious program binaries include anti-debugging checks, anti-emulation checks, timing checks, and anti-tampering (e.g., self-checksumming) techniques. Since our current implementation uses the debugger interface, we have neutralized the common anti-debugging checks [18]. In practice, anti-debugging and timing checks leave footprints that are evident in our pre-execution analysis of the code and that the analyst can disable with the instrumentation capabilities that we provide. Our research is investigating multiple alternatives for neutralizing the effect of anti-tampering techniques, but this work is beyond the scope of this paper.

We proceed by discussing related work in Section 2 and we give an overview of our techniques and algorithm in Section 3. Our code-discovery techniques are described in Sections 4-7. In Section 8 we show the utility of our approach by applying our tool to analysis-resistant malware, and to synthetic samples produced by the most prevalent binary packing tools. We conclude in section 9.

2 Related Work

Our work is rooted in the research areas of program binary analysis, instrumentation, and unpacking.

Analysis. Static parsing techniques can accurately identify 90% or more of the functions in compiler-generated binaries despite the lack of symbol information [43], but are much worse at analyzing arbitrarily obfuscated code [24, 51], and cannot analyze the packed code that exists in most malicious binaries [9]. Thus, most malware analysis is dynamic and begins by obtaining a trace of the program's executed instructions through single-step execution [17], dynamic instrumentation [41], or the instrumentation capabilities of a whole-system emulator [32]. The resulting trace is used to construct an analysis artifact such as a visualization of the program's unpacking behavior [42], a report of its operating

system interactions [6], or a representation that captures the evolving CFG of a self-modifying program [4]. As these analysis artifacts are all produced after monitoring the program's execution, they are potential clients of our analysis-guided instrumentation techniques rather than competitors to them.

Madou et al. [28] and Cifuentes and Emmerik [13] combine static and dynamic techniques to identify more code than is possible through either technique alone. Madou et al. start from an execution trace and use control-flow traversal parsing techniques to find additional code, whereas Cifuentes and Emmerik start with speculative static parsing and use an instruction trace to reject incorrect parse information. Their hybrid approaches to CFG-building are similar to ours in spirit, but they analyze only code that is statically present in the binary as they lack the ability to capture dynamically unpacked and overwritten code.

Instrumentation. Existing tools that provide analysis-guided binary instrumentation [22, 49, 48] cannot instrument code that is obfuscated, packed, or self-modifying, as their code identification relies exclusively on static analysis. Our tool uses Dyninst's [22] dynamic instrumentation and analysis capabilities, updating its analysis of the code through both static and dynamic code-capture techniques, prior to the code's execution.

The BIRD dynamic instrumenter [34] identifies binary code by augmenting its static parse with a run-time analysis that finds code by instrumenting control transfers that could lead to unknown code areas. BIRD works well on compiler-generated programs, but does not handle self-modifying programs and performs poorly on programs that are packed or obfuscated, as it is not optimized for extensive dynamic code discovery (it uses trap instructions to instrument all return instructions and other forms of short indirect control transfers that it discovers at runtime). BIRD also lacks a general-purpose interface for instrumentation and does not produce analysis tools for the code it identifies.

Other dynamic instrumentation tools forgo static analysis altogether, instead discovering code as the program executes and providing an instruction-level interface to the code (e.g., PIN [27], Valgrind [7]). These tools can instrument dynamically unpacked and self-modifying code, but do not defend against analysis-resistance techniques [18]. As with BIRD, the lack of a structural analysis means that it is difficult to selectively instrument the code and that it may not be possible to perform simple-sounding tasks like hooking a function's return values because compiler optimizations (and obfuscations) introduce complexities like shared code, frameless functions, and tail calls in place of return statements.

Unpacking. The prevalence of code packing techniques in malware has driven the creation of both static and dynamic approaches for detecting packed malicious code. Some anti-virus tools (e.g., BitDefender [8]) create static unpackers for individual packer tools at significant expense, but this approach will not scale with the explosive growth rate of new packing approaches [9, 35]. Anti-virus tools also employ "X-Ray" techniques that can statically extract the packed contents of binaries that employ known compression schemes or weak encryption [37]. Coogan et al. [15] use static analysis to extract the unpacking routine from a packed binary and then use the extracted routine to unpack it. These static

approaches are unsuccessful when confronted with malware that employs multiple packing layers (e.g., Rustock [12]), and Coogan et al.'s static techniques are also unable to deal with heavily obfuscated code [33, 46].

Most dynamic unpacking tools take the approach of detecting memory locations that are written to at run-time and later executed as code. OmniUnpack [30], Saffron [41], and Justin [20] approach the problem at a memory-page granularity by modifying the operating system to manipulate page write and execute permissions so that both a write to a page and a subsequent execution from that page result in an exception that the tool can intercept. This approach is efficient enough that it can be used in an anti-virus tool [30], but it does not identify unpacked code with much precision because of its memory-page granularity.

Other unpackers identify written-then-executed code at a byte level by tracing the program's execution at a fine granularity and monitoring all memory writes. EtherUnpack [17] and PolyUnpack [44] employ single-step execution of the binary, whereas Renovo [23] and "Saffron for Intel-PIN" [41] use the respective instruction-level instrumentation capabilities of the Qemu whole-system emulator [7] and the PIN instrumenter [27]. By contrast, our analysis-guided instrumentation allows us to unpack *and analyze* program binaries with a hundred-fold reduction in instrumented program locations and comparable execution times.

3 Technical Overview

Our hybrid algorithm combines the strengths of static and dynamic analysis. We use static parsing techniques to analyze code before it executes, and dynamic techniques to capture packed, obfuscated, and self-modifying code. Hybrid analysis allows us to provide *analysis-guided dynamic instrumentation* on analysis-resistant program binaries for the first time, based on the following techniques:

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 achieves high accuracy on arbitrarily obfuscated binaries, so we create a modified control-flow traversal algorithm [47] with a low false-positive rate. Our initial analysis of the code may be incomplete, but we can fall back on our dynamic capture techniques to find new entry points into the code and use them to re-seed our parsing algorithm.

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. This approach is similar to BIRD's [34], but monitors a smaller set of control transfers.

Code Overwrite Monitoring. Code overwrites invalidate portions of an existing code analysis and introduce new code that has not yet been analyzed. We adapt DIOTA's [29] mechanism for detecting code overwrites by write-protecting

```
1. Load the program into memory, paused at its entry point
2. Remove debugging artifacts
3. Parse from known entry points
4. Instrument newly discovered code
5. Resume execution of the program
6. Handle code discovery event, adding new entry points
7. Goto 3
```

Fig. 1: Algorithm for binary code discovery, analysis, and instrumentation

memory pages that contain code and handling the signals that result from write attempts. Accurately detecting when overwriting ends is important, as it allows us to update our analysis only once when large code regions are overwritten in small increments. 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, allowing us to delay the analysis update until the loop exits.

Signal- and Exception-Handler Analysis. We use dynamic analysis to resolve signal- and exception-based control transfer obfuscations [18, 38]. We detect signal- and exception-raising instructions and find their dynamically registered handlers through standard techniques, and then add the handlers to our analysis and instrument them to control their execution.

Figure 1 illustrates how we combine the above techniques into an iterative algorithm that allows us to provide analysis-guided dynamic instrumentation of analysis-resistant program binaries. The key feature of this algorithm is that it allows all of the program's code to be analyzed and instrumented before it executes. Our algorithm's incremental instrumentation of the code is similar to Mirgorodskiy and Miller's use of "self-propelled instrumentation" to trace a program's execution [31], but we also analyze and instrument analysis-resistant code, whereas they can instrument only statically analyzable code.

4 Parsing

The purpose of our parsing algorithm is to accurately identify binary code and analyze the program's structure, producing an interprocedural control flow graph of the program. Existing parsing techniques for arbitrarily obfuscated code have attempted to identify code with good accuracy and coverage, and have come up short on both counts [24]. Instead, we prioritize accurate code identification, as an incorrect parse can cause incorrect program behavior by leading to the instrumentation of non-code bytes, and is ultimately not very useful. The competing goal of good coverage is relatively less important, because our dynamic techniques compensate for lapses in coverage by capturing statically un-analyzable code at run-time and triggering additional parsing.

Control-flow traversal parsing [47] is the basis for most accurate parsing techniques, but it makes three unsafe assumptions about control flow that can re-

duce its accuracy. 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 [39]) often contain functions that return to unexpected locations by tampering with the call stack [25]. Second, the algorithm assumes that control flow is only redirected by control transfer instructions. Obfuscated programs often use an apparently normal instruction to raise a signal or exception, thereby transferring control to code that is hidden in a signal or exception handler [18]. The handler can further obfuscate control flow by telling the operating system to resume execution away from the signal- or exception-raising instruction, potentially causing non-code bytes to be parsed following the instruction [38]. Third, the algorithm assumes that both targets of conditional branch instructions can be taken and therefore contain valid code. Program obfuscators can exploit this assumption by creating branches with targets that are never taken, thereby diluting the analysis with junk code that never executes [14].

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, and we have addressed this vulnerability in our current parsing algorithm. We detect and resolve signal- and exception-based obfuscations at run-time (see Section 7), when we analyze and instrument any hidden code and correct our analysis to include the obfuscated control transfer. The use of branches with targets that are never taken dilutes the analysis with junk code but has thus far not been a problem for our hybrid analysis and instrumentation techniques. Our ongoing work will improve upon our parser's accuracy by adding static detection of some fault-based control transfers and never-taken branch targets, thereby making our instrumentation of the program safer and more efficient. In the meantime, our current parsing algorithm achieves significant accuracy improvements relative to existing techniques for parsing obfuscated code, allowing us to analyze and instrument most analysis-resistant programs.

Non-returning Calls. When a called function either never returns or returns to an unexpected location by tampering with the call stack [25], one or more junk bytes may follow the function call site. The simplest approach to this problem would be to adopt the assumption made by BIRD [34] and Kruegel et al.'s obfuscated code parser [24], that function calls never return, and then rely on run-time monitoring of return instructions to discover code that follows call sites. This runtime-discovery approach is taken by BIRD, and while it is our technique of last resort, our data-flow analysis of called functions can often tell us whether the function will return, and to where, thereby increasing the code coverage attained through parsing and avoiding unnecessary instrumentation.

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 parsing after call instructions. We do not resume parsing after the call site if our analysis of the called function contains no return instructions, or if our static call stack analysis [26] of the function detects that it tampers with the stack. Our call-stack analysis emulates the function's stack operations to detect whether the function

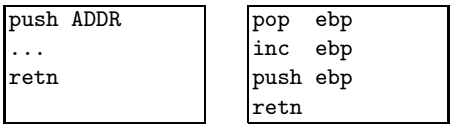


Fig. 2: Code sequences that tamper with return addresses on the call stack

alters its return address, either by overwriting the address or by imbalancing the call stack. Figure 2 illustrates two call-stack tampering tricks used by the ASPack packer that are easily detected by our analysis. Figure 2a shows an instruction sequence that transfers control to `ADDR` upon executing the return instruction, while Figure 2b shows a sequence that increments 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, and the byte immediately following the call site is not parsed as code.

5 Dynamic Capture

Having found and analyzed as much of the code as possible by traversing the program's statically analyzable control flow, we turn to *dynamic capture* techniques to find code that is not statically analyzable. Statically un-analyzable code includes code that is present in the binary but is reachable only through pointer-based address calculations, and code that is not initially present because it is dynamically unpacked. Our approach to both problems lies in monitoring those control transfers whose targets are either unknown or invalid when we originally parse them. More precisely, we use dynamic capture instrumentation to monitor the execution of instructions that meet one of the following criteria:

- *Control transfer instructions that use registers or memory values to determine their targets.* Obfuscated programs often used indirect jump and call instructions to hide code from static analysis. For example, the FSG packer has an indirect function call for every 16 bytes of bootstrap code. We determine whether indirect control transfers leave analyzed code by resolving their targets at run-time with dynamic instrumentation. In the case of indirect call instructions, when our parser cannot determine the call's target address, it also cannot know if the call will return, so it conservatively assumes that it does not; our instrumentation allows us to trigger parsing both at call target and after the call site, if we can determine that the called function returns.
- *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 4, during parsing we attempt to determine whether called functions return normally

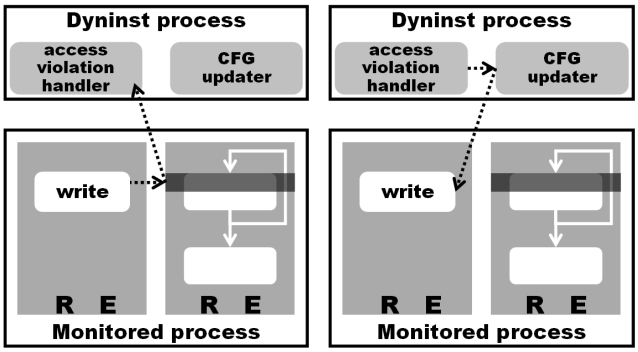
so that we can continue parsing after call sites to those functions. If our analysis is inconclusive we instrument the function's return instructions.
- *Control transfer instructions into invalid or uninitialized memory regions.* Control transfers to dynamically unpacked code can appear this way, as code is often unpacked into uninitialized (e.g., UPX) or dynamically allocated memory regions (e.g., NSPack). Our instrumentation of these control transfer instructions executes immediately prior to the transfer into the region, when it must contain valid code, allowing us to analyze it before it executes.
- *Instructions that terminate a code sequence by reaching the end of initialized memory.* Some packer tools (e.g., UPack) and custom-packed malware (e.g., Rustock [12]) transition to dynamically unpacked code without executing a control transfer instruction. These programs map code into a larger memory region so that a code sequence runs to the end of initialized memory without a terminating control transfer instruction. The program then unrolls the remainder of the sequence into the region's uninitialized memory so that when it is invoked, control flow falls through into the unpacked code. We trigger analysis of the unpacked instructions by instrumenting the last instruction of any code sequence that ends without a final control transfer instruction.

Our dynamic capture instrumentation supplies our parser with entry points into un-analyzed code. Before extending our analysis by parsing from these new entry points, we determine whether the entry points represent un-analyzed functions or if they 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 (unless the branch instruction and its target are in different memory regions). The target of a non-obfuscated return instruction is always 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 tampered with and parse the target as a new function.

Cost issues arise from our use of the Dyninst instrumentation library [22] because it monitors programs from a separate process that contains the analysis and instrumentation engine. The problem is that our code-discovery instrumentation requires context switching between the two processes to determine whether monitored control transfers lead to new or analyzed code. We reduce this overhead by caching the targets of these instructions in the address space of the monitored program, and context switching to Dyninst only for cache misses.

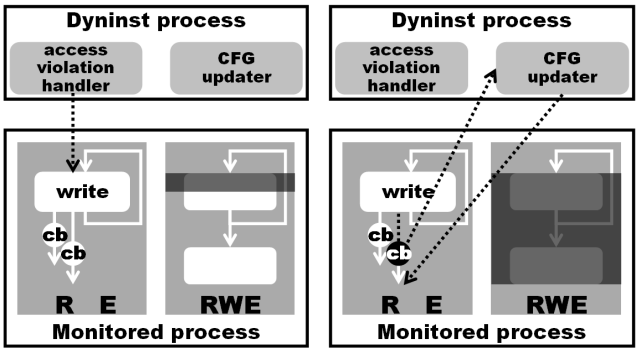
6 Response to Overwritten Code

Code overwrites cause significant problems for binary analysis. Most analysis tools cannot analyze overwritten code because they rely on static CFG representations of the code. Code overwrites cause problems for CFGs by simultaneously invalidating portions of the CFG and introducing new code that has yet to be analyzed. We have developed techniques to address this problem by updating the program's CFG and analyzing overwritten code before it executes.



(a) The monitored program overwrites an instruction. We have removed write permissions from code pages to cause code overwrites to raise access rights violations.

(b) A correct but inefficient approach that updates the CFG in response to each code write, triggering major processing.



(c) Our optimized handler instruments the loop's exit points with callbacks to our CFG updater and restores write permissions to the overwritten page.

(d) When writing completes, the instrumentation at a loop exit triggers a callback to our CFG updater.

Fig. 3: Our approach to detecting code writes is shown in Figure 3a, alternative methods for responding to code writes are shown in Figures 3b and 3c-3d.

To analyze overwritten code before it executes, we can either detect the modified instructions immediately prior to their execution, by checking whether the bytes of each executed instruction have changed [44], or detect writes to code as soon as they occur, by monitoring write operations to analyzed code regions. Monitoring each instruction for changes is expensive because it requires single-step execution of the program. Fortunately, we can efficiently detect write operations that modify code by adapting DIOTA's techniques for intercepting writes to executable code regions [29]. DIOTA monitors writes to all memory pages that are 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 3a, we have adapted this mechanism for 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.

A naïve approach based on monitoring writes to code pages might respond to write attempts by emulating each write and immediately updating the analysis, as shown in Figure 3b. Doing so is undesirable for efficiency reasons. Large code regions are often overwritten in small increments, and updating the CFG in response to every code write is unnecessarily expensive, as the analysis does not need to be updated until the overwritten code is about to execute. Instead, we catch the first write to a code page but allow subsequent writes, delaying the update to the window between the end of code overwriting and the beginning of modified code execution. This delayed-update approach divides our response to code overwrites into two components that we now 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 executes.

6.1 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. We disambiguate between real access-rights violations and protected-code violations by looking at the write's target address. Protected-code violations are artificially introduced by our code-protection mechanism and we handle them ourselves to hide them from the monitored program. For real access-rights violations we apply the techniques of Section 7 to analyze the registered handler and pass the signal or exception back to the program.

Our handler also decides when to update the CFG, attempting to trigger the update after the program has stopped overwriting its code, but before the modified code executes. A straightforward approach would be 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 [30], Justin [20], and Saffron [41]). Unfortunately, this approach fails in the common case that the write instruction writes repeatedly to its own page, when this approach effectively devolves into single-step execution. Instead, we apply the techniques shown in Figures 3c and 3d to detect the end of overwriting, and delay updating the CFG until then. We perform inter-procedural loop analysis on the execution context of the faulting write instruction to see if the write is contained in a loop, in which case we instrument the loop's exit edges with callbacks to our CFG update routine. We allow subsequent writes to the write-target's code page to proceed unimpeded, by restoring the page's write permissions. When the write loop terminates, one of the loop's instrumented exit edges causes the CFG update routine to be invoked. We take extra precautions if the write loop's code pages intersect with the overwritten pages to ensure that the write loop does not modify its own code. We safeguard against this case by adding bounds-check instrumentation to all of the loop's write operations so that any modification of the loop's code will immediately trigger our CFG update routine.

Our handler's final task is to save a pre-write copy of the overwritten memory page, so that when writing completes, the CFG update routine can identify the page's modified instructions by comparing the overwritten page to the pre-write copy of the page. If the loop writes to multiple code pages, the first write to each code page results in a separate invocation of our protected-code handler, which triggers the generation of a pre-write copy of the page, and associates it with the write loop by detecting that the write instruction lies within it. Our handler then restores write permissions to the new write-target page and resumes the program's execution. When the write loop finally exits, instrumentation at one of its exit edges triggers a callback to our CFG update routine.

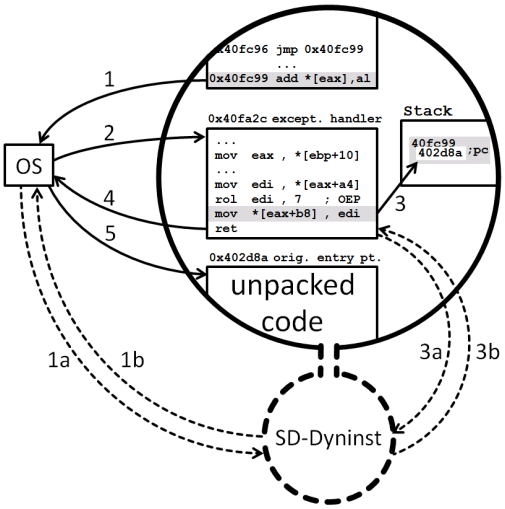
6.2 Updating the Control Flow Graph

We begin updating our analysis by determining the extent to which the code has been overwritten. We identify overwritten bytes by comparing the overwritten code pages with our saved pre-write copies of those pages, and then detemine which of the overwritten bytes belong to analyzed instructions. If code was overwritten, we clean up the CFG by purging it of overwritten basic blocks and of blocks that are only reachable from overwritten blocks. We analyze the modified code by seeding our parser with entry points into the modified code regions. We then inform the analyst of the changes to the program's CFG so that the new and modified functions can be instrumented. After adding our own dynamic capture instrumentation to the new code, we again remove write permissions from the overwritten pages and resume the monitored program's execution.

7 Signal- and Exception-Handler Analysis

Analysis-resistant programs are often 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, as they are usually registered (and often unpacked) at run-time. Current dynamic instrumentation tools do not analyze signal and exception handlers [27, 29], whereas we analyze them and provide analysis-based instrumentation on them. This ability to analyze and control the handlers is important on analysis-resistant binaries because the handlers may perform tasks that are unrelated to signal and exception handling (e.g., PECompact overwrites existing code).

Signal and exception handlers can further obfuscate the program by redirecting control flow [38]. When a signal or exception is raised, the operating system gives the handler context information about the fault, including the program counter value. The handler can modify this saved PC value to cause the OS to resume the program's execution at a different address. As shown in step 3 of Figure 4, this technique is used by the "Yoda's Protector" packer to obfuscate its control transfer to the program's original entry point (OEP) [16]. Yoda's Protector raises an exception, causing the OS to invoke Yoda's exception handler. The handler overwrites the saved PC value with the address of the program's OEP, causing the OS to resume the program's execution at its OEP.



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 SD-Dyninst process of the exception.*
1b: *SD-Dyninst analyzes the registered exception handler, instruments its exit points, and returns to the OS.*
2:The OS calls the program's exception handler.
3:The handler overwrites the saved PC with the address of the program's original entry point.
3a: *Instrumentation at the handler's exit point invokes SD-Dyninst.*
3b: *SD-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.

Fig. 4: 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 SD-Dyninst's analysis of the control transfer through its attached debugger process.

Analyzing Signal- and Exception-Based Control Flow. We find and analyze handlers by intercepting signals and exceptions at run-time. Signal and exception interception is possible whether we observe the malware's execution from a debugger process or virtual machine monitor (VMM). In our current implementation, SD-Dyninst is apprised of raised signals and exceptions through standard use of the debugger interface. A VMM-based implementation would automatically be apprised of signals and exceptions, and would use VM-introspection techniques to detect which of them originate from malicious processes [19].

As shown in Figure 4, upon notification of the signal or exception, we analyze and instrument the program's registered handlers. We find handlers in Windows programs by traversing the linked list of structured exception handlers that is on the call stack of the faulting thread. Finding handlers is even easier in Unix-based systems because only one 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 that the handler will redirect control flow by instrumenting it at its exit points. After analyzing the handler, but before it executes, we insert our exit-point instrumentation (step 1b of Figure 4). We inform the analyst's tool of the signal or exception and of the newly discovered handler code so that it can add its own instrumentation. We then return control to the OS, which invokes the program's exception handler. When the handler is done executing, our exit-point instrumentation triggers a callback to our analysis engine (steps 3a-3b of Figure 4), where we check for modifications to the

Table 1: Our analysis techniques applied to the most prevalent packer tools used to obfuscate malware. We analyzed all of the packed binaries that do not employ anti-tampering techniques.

Packer	Malware market share	Over-writes code	Anti tamper-ing	Time (seconds)				
				Pre-exec'n	Pre-payload		Post-payload	
					instr.	uninstr.	instr.	uninstr.
UPX	9.45%			23.44	0.50	0.02	2.80	0.02
PolyEnE	6.21%			22.55	1.24	0.02	2.81	0.02
EXECryptor	4.06%	yes	yes					
Themida	2.95%	yes	yes					
PECompact	2.59%	yes	yes	22.81	3.16	0.02	2.81	0.02
UPack	2.08%	yes		22.56	23.50	0.03	4.08	0.02
nPack	1.74%			23.21	1.54	0.02	2.80	0.02
ASPack	1.29%	yes		22.58	4.42	0.02	2.81	0.02
FSG	1.26%			22.53	1.38	0.03	2.78	0.02
Nspack	0.89%			22.52	2.69	0.03	2.78	0.02
ASProtect	0.43%	yes	yes					
Armadillo	0.37%	yes	yes					
Yoda's Protector	0.33%	yes	yes					
WinUPack	0.17%	yes		22.44	23.60	0.03	4.10	0.02
MEW	0.13%			22.56	3.87	0.03	2.80	0.02

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.

8 Experimental Results

We evaluated our techniques by implementing them in SD-Dyninst and applying it to real and synthetic malware samples. We show that we can efficiently analyze obfuscated, packed, and self-modifying code by applying our techniques to the binary packer tools that are most heavily used by malware authors, comparing these results to two of the most efficient existing tools. We demonstrate the usefulness of our techniques by using SD-Dyninst to create a *malware analysis factory* that we apply to a large batch of recent malware samples. Our analysis factory uses instrumentation to construct annotated program CFG's and a stackwalk at the program's first socket communication.

8.1 Analysis of Packer Tools

Packed binaries begin their execution in highly obfuscated metacode that is often self-modifying and usually unpacks itself in stages. The metacode decompresses or decrypts the original program's payload into memory and transfers control to the payload code at the original program's entry point.

Table 1 shows the results of applying our techniques to the packer tools that are most often used to obfuscate malicious binaries, as determined by Panda

Research for the months of March and April 2008, the latest dates for which such packer statistics were available [10]. We do not report results on some of these packers because they incorporate anti-tampering techniques such as self-checksumming, and SD-Dyninst does not yet incorporate techniques for hiding its use of dynamic instrumentation from anti-tampering. We excluded NullSoft's installer tool (with 3.58% market share) from this list because it can be used to create binaries with custom code-unpacking algorithms; though we can handle the analysis-resistance techniques contained in most NullSoft-based packers, we cannot claim success on all of them based on successful analysis of particular packer instances. We also excluded the teLock (0.63% market share) and the Petite (0.25% market share) packer tools, with which we were unable to produce working binaries. The total market share of the packer tools listed by Panda Research is less than 40% of all malware, while at least 75% of malware uses some packing method [8, 50]. This discrepancy is a reflection both of the increasing prevalence of custom packing methods and a limitation of the study, which relied on a signature-based tool to recognize packer metacode [11]; most custom packers are derivatives of existing packer tools, which are often modified with the express purpose of breaking signature-based detection.

In Table 1 we divide the execution time of the program into pre- and post-payload execution times, representing the time it takes to execute the binaries' metacode and payload code, respectively. In the uninstrumented case, we determine the proper time split by using a priori knowledge of the packed program to breakpoint its execution at the moment that control transfers from its metacode to its payload code. In the instrumented case, our code-discovery instrumentation automatically identifies this transition by capturing the control transfer to the payload code. We report on SD-Dyninst's pre-execution cost separately, as one of the major benefits of incorporating static analysis techniques into our approach is that we are able to frontload much of the analysis of the program so that it does not affect the program's execution time.

The most striking differences in Table 1 are in the pre-payload cost incurred by SD-Dyninst from one packer to the next. These differences are proportional to the number of occasions in which we discover and analyze new code in the metacode of these binaries. Our instrumentation of the UPX, PolyEnE, nPack, and Nspack packers caused little slowdown in their execution, as their metacode is static and not obfuscated in ways that substantially limit our ability to parse them prior to their execution, while the FSG, MEW, ASPack, UPack, and WinUpack packers are more heavily obfuscated and unpack in stages, requiring that we analyze their code incrementally. The largest contributor to the incremental analysis cost is SD-Dyninst's current inability to resolve the targets of indirect control transfers at parse time, coupled with a simplifying implementation decision to instrument whole functions at a time, meaning that discovery of a new basic block currently causes its entire function to be re-instrumented. SD-Dyninst's performance will improve significantly in the near future through the addition of code-slicing capabilities to Dyninst, which will allow SD-Dyninst to resolve many indirect control transfers at parse time.

Table 2: A comparison of pre-payload execution times and instrumented program locations in SD-Dyninst, Renovo, Saffron, and EtherUnpack on packed executables.

	Pre-payload time				Instrumented locations		
Packer	SD-D.	Ren.	Saff.	Ether	SD-D.	Ren.	Saff.
UPX	0.5	5	2.7	7.6	6	2,278	4,526
ASPack	4.4	5	fail	18.7	34	2,045	4,141
FSG	1.6	8	1.4	31.1	14	18,822	31,854
WinUpack	23.6	8	23.5	67.8	23	18,826	32,945
MEW	4.0	6	fail	150.5	22	21,186	35,466

In Table 2 we compare the overall expense of our techniques to the most efficient tools for identifying dynamically unpacked and modified code, Renovo [23], "Saffron for Intel PIN" [41], and EtherUnpack [17]. We executed Saffron and EtherUnpack on our own hardware, but the Renovo tool is not yet publicly available, so we compared against Renovo's self-reported execution times for packed notepad.exe executables, limiting this comparison to the top packer tools that they also analyzed in their study. We ran SD-Dyninst and Saffron on an Intel Core 2 Duo T2350 1.6GHz CPU with 2.5GB of memory, while Renovo's study was conducted on an Intel Core 2 Duo E6660 2.4GHz CPU with 4GB memory and EtherUnpack executed on an Intel Xeon E5520 2.27GHz CPU with 6GB of memory. These numbers reflect the post-startup time it took for SD-Dyninst, Renovo, Saffron, and EtherUnpack to execute the instrumented metacode of the various packer tools.

As seen in Table 2, except in the case of our unoptimized analysis of the WinUpack executable, our pre-payload execution times are comparable to those of Renovo, Saffron, and EtherUnpack *despite the fact* that our tool also analyzes the code while the other tools only identify its dynamically unpacked portions. The Saffron unpacker only partially unpacked the ASPack and MEW executables, as it quits at the first occurrence of written-then-executed code.

The savings afforded by our use of analysis-guided instrumentation help to amortize our analysis costs. Saffron instruments the program at every instruction, while Renovo instruments at all control transfers and at all write instructions (EtherUnpack's single-step mechanism does not rely on instrumentation). We estimated Saffron's use of instrumentation by modifying its source code to maintain a count of unique instrumented instructions, and estimated Renovo's use of instrumentation based on their algorithm description, by instrumenting the packed programs to maintain a count of unique control transfer and write instructions. Table 2 shows that our structural analysis allows us to instrument the program at fewer than a 100th of the locations instrumented by Saffron and Renovo, because our structural analysis allows us to limit our use of instrumentation to instructions whose targets are statically unresolved.

8.2 Malware Analysis

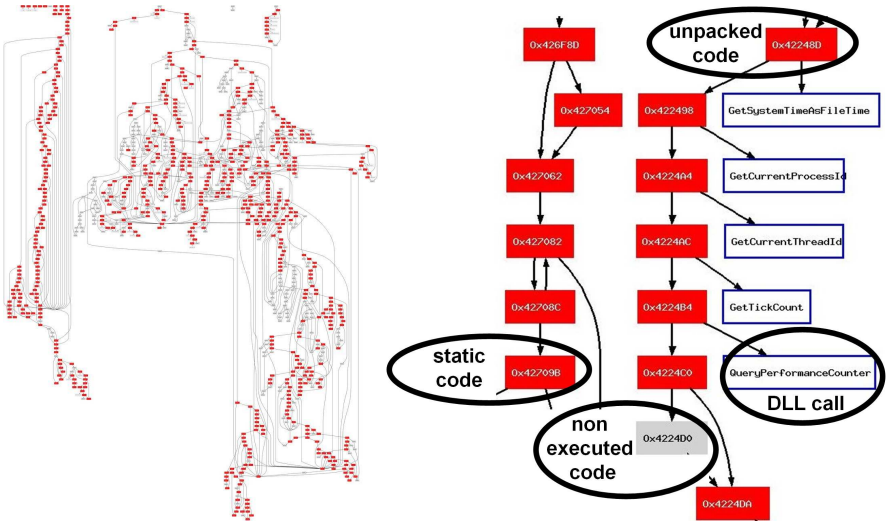
Accomplishing an analysis task using SD-Dyninst requires no more skill from the analyst than performing the same task with Dyninst on a conventional binary. We wrote a malware analysis factory that uses SD-Dyninst to perform code-coverage of malicious program executions by instrumenting every basic block in the program, (both statically present and dynamically unpacked blocks) and removing the instrumentation once it has executed. This instrumentation code consists of only fifty lines. Our factory halts the malware at the point that it attempts its first network communication, exits, or reaches a 30-minute timeout. At this time, the factory prints out a traversal of the program's call stacks and outputs a CFG of the binary that identifies calls to Windows DLL functions and is annotated to distinguish between blocks that have executed and those that have not. If the malware fails to send a network packet, we verify our analysis by comparing against a trace of the malware's execution to ensure that our analysis reaches the same point.

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. We then analyzed 200 malware samples that were collected by Offensive Computing [3] 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 identified by dynamic analysis. For the malware samples that attempted to communicate with the network, our analysis factory walked their call-stacks to identify the code in the malicious executable that triggered the network communication.

As an example of the kinds of results produced by our factory, in Figures 5 and 6 we show two of its analysis products for the Conficker A malware binary. Our factory created similar analysis products for all the other malware binaries that we analyzed, and these can be viewed online at `http://www.paradyn.org/SD_Dyninst/`. In Figure 5a we show the annotated CFG of the Conficker A binary in its entirety, while Figure 5b shows an excerpt of that graph, highlighting the fact that SD-Dyninst has captured static and dynamic code, both code in the executable and code in Windows DLL's, and both code that has executed and code that has not executed but that may be of interest to the analyst. Figure 6 shows our traversal of Conficker's call stacks at its first call to the `select` routine. As seen in this stack trace, we are able to identify the stack frames of functions that lack symbol information, an important benefit of our analysis capabilities. While existing stackwalking techniques are accurate only for statically analyzable code [26], our hybrid analysis enables accurate stackwalking by virtue of having analyzed all of the code that could be executing at any given time.

9 Conclusion

We create a hybrid analysis algorithm that makes it possible to analyze and control the execution of malicious program binaries in a way that is both more



(a) The entire CFG.

(b) CFG excerpt showing Conficker's basic blocks and its calls to system libraries. Basic blocks that have executed are shown in red blocks, non-executed blocks are in light-grey, and API functions are shown as white rectangles.

Fig. 5: 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).

```
top  pc=0x7c901231 DbgBreakPoint_0x7c901230 in ntdll.dll   [Win DLL]
     pc=0x10003c83 DYNbreakPoint_0x10003c70 in dyn_RT.dll  [Instrument.]
     pc=0x100016f7 DYNstopThread_0x10001670 in dyn_RT.dll  [Instrument.]
     pc=0x71ab2dc0 select_0x71ab2dc0        in WS2_32.dll  [Win DLL]
base pc=0x401f34   func=nosym1f058_0x41f058 in cf.exe      [Conficker]
```

Fig. 6: An SD-Dyninst stack walk taken when the Conficker A binary executes Winsock's `select` routine. The stack walk includes frames from our instrumentation, `select`, and Conficker.

intuitive and more efficient than existing methods. Our combination of static and dynamic analysis allows us to provide analysis-guided instrumentation on obfuscated, packed, and self-modifying program binaries for the first time. We implemented these ideas in SD-Dyninst, and demonstrated that they can be applied to most of the packing tools that are popular with current malware. We demonstrated the usefulness of our techniques by applying SD-Dyninst to produce analysis artifacts for the Conficker A binary that would have required substantial manual effort to produce through alternative methods.

Ongoing research in the Dyninst project promises to address the two primary limitations of this work. First, our instrumentation's changes to the program's address space can be detected through anti-tampering techniques such as self-checksumming. The second problem is that our parsing techniques assume that

the targets of conditional control transfers always represent real code, meaning that the obfuscation proposed by Collberg et al. [14] could be used to pollute our parse with non-code bytes, potentially causing us to instrument data and causing the program to malfunction. Both of these problems are being addressed by a piece of ongoing research in Dyninst that will ensure that the presence of instrumentation will not impact a program's data accesses.

Two additional analysis-resistance techniques that merit discussion are anti-debugging and timing checks. Several Windows API and system calls can be used to detect the presence of a debugger. Invocations of such functions are easily detected by SD-Dyninst instrumentation and we have disabled those we have come across, but from the literature [18] we know there are additional anti-debugging methods that our current implementation does not disable. Timing checks similarly depend on Windows API calls, system calls, and special-purpose instructions that are easily detected by SD-Dyninst. However, we have not had to disable timing checks, as most existing checks are tuned to detect the significant slowdowns incurred by single-step debugging techniques and are not triggered by our more efficient instrumentation-based techniques. It is possible that future timing checks could detect the slowdown caused by our algorithm's online analysis, in which case we could adapt Ether's [17] clock emulation techniques to hide the slowdown from the program.

Acknowledgments

This work is supported in part by Department of Energy grants DE-SC0004061, 08ER25842, 07ER25800, DE-SC0003922, Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), and National Science Foundation Cybertrust grants CNS-0627501, and CNS-0716460.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Computer economics 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code (2007)
2. Darkparanoid virus (1998)
3. Offensive computing, `http://www.offensivecomputing.net`
4. Anckaert, B., Madou, M., Bosschere, K.D.: A model for self-modifying code. In: Information Hiding. pp. 232–248. Alexandria, VA (2007)
5. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: International Conference on Compiler Construction. pp. 5–23. New York, NY (2004)
6. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology 2(1), 66–77 (2006)
7. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference. pp. 41–46. Anaheim, CA (2005)
8. BitDefender: BitDefender anti-virus technology. White Paper (2007)
9. Bustamante, P.: Malware prevalence. Panda Research web article (2008)

10. Bustamante, P.: Packer (r)evolution. Panda Research web article (2008)
11. Bustamante, P.: Personal correspondence (2009)
12. Chiang, K., Lloyd, L.: A case study of the rustock rootkit and spam bot. In: First Conference on Hot Topics in Understanding Botnets. Cambridge, MA (2007)
13. Cifuentes, C., Emmerik, M.V.: UQBT: adaptable binary translation at low cost. Computer 33(3), 60–66 (2000)
14. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Symposium on Principles of Programming Languages. pp. 184–196. San Diego, CA (1998)
15. Coogan, K., Debray, S., Kaochar, T., Townsend, G.: Automatic static unpacking of malware binaries. In: Working Conference on Reverse Engineering. Antwerp, Belgium (2009)
16. Danehkar, A.: Inject your code into a portable executable file (2005), `http://www.codeproject.com/KB/system/inject2exe.aspx`
17. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Conference on Computer and Communications Security. Alexandria, VA (2008)
18. Ferrie, P.: Anti-unpacker tricks. In: International CARO Workshop. Amsterdam, Netherlands (2008)
19. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed System Security Symposium. San Diego, CA (2003)
20. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: RAID. pp. 98–115. Springer Berlin / Heidelberg, Cambridge, MA (2008)
21. Hind, M., Pioli, A.: Which pointer analysis should I use? In: International Symposium on Software Testing and Analysis. pp. 113–123. Portland, OR (2000)
22. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic program instrumentation for scalable performance tools. In: Scalable High Performance Computing Conference. Knoxville, TN (1994)
23. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Workshop on Recurring Malcode. Alexandria, VA (2007)
24. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: USENIX Security Symposium. San Diego, CA (2004)
25. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Conference on Computer and Communications Security. pp. 290–299. Washington, D.C. (2003)
26. Linn, C., Debray, S., Andrews, G., Schwarz, B.: Stack analysis of x86 executables (2004), manuscript
27. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation. pp. 190–200. Chicago, IL (2005)
28. Madou, M., Anckaert, B., de Sutter, B., Bosschere, K.D.: Hybrid static-dynamic attacks against software protection mechanisms. In: ACM workshop on Digital Rights Management. pp. 75–82. Alexandria, VA (2005)
29. Maebe, J., Bosschere, K.D.: Instrumenting self-modifying code. In: International Workshop on Automated and Algorithmic Debugging. Ghent, Belgium (2003)
30. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Annual Computer Security Applications Conference. Miami Beach, FL (2007)

31. Mirgorodskiy, A.V., Miller, B.P.: Autonomous analysis of interactive systems with self-propelled instrumentation. In: International Conference on Parallel Computing. San Jose, CA (2005)
32. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Symposium on Security and Privacy. pp. 231–245. Oakland, CA (2007)
33. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Annual Computer Security Applications Conference. Miami Beach, FL (2007)
34. Nanda, S., Li, W., Lam, L.C., cker Chiueh, T.: Bird: Binary interpretation using runtime disassembly. In: International Symposium on Code Generation and Optimization (CGO 2006). pp. 358–370. New York, NY (2006)
35. Neumann, R.: Exepacker blacklisting part 2. Virus Bulletin pp. 10–13 (2007)
36. Nguyen, A.M., Schear, N., Jung, H., Godiyal, A., King, S.T., Nguyen, H.: Mavmm: A lightweight and purpose-built vmm for malware analysis. In: Annual Computer Security Applications Conference. Honolulu, HI (2009)
37. Perriot, F., Ferrie, P.: Principles and practise of x-raying. In: Virus Bulletin Conference. pp. 51–66. Chicago, IL (2004)
38. Popov, I., Debray, S., Andrews, G.: Binary obfuscation using signals. In: USENIX Security Symposium. pp. 275–290. Boston, MA (2007)
39. Porras, P., Saidi, H., Yegneswaran, V.: A multi-perspective analysis of the storm (peacomm) worm. SRI International Technical Report (2007)
40. Porras, P., Saidi, H., Yegneswaran, V.: An analysis of conficker's logic and rendezvous points. SRI International Technical Report (2009)
41. Quist, D., Ames, C.: Temporal reverse engineering. In: Blackhat USA. Las Vegas, NV (2008)
42. Quist, D.A., Liebrock, L.M.: Visualizing compiled executables for malware analysis. In: Workshop on Visualization for Cyber Security. Atlantic City, NJ (2009)
43. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: Conference on Artificial Intelligence. Chicago, IL (2008)
44. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: Annual Computer Security Applications Conference. pp. 289–300. Miami Beach, FL (2006)
45. Security, P.: Annual report Pandalabs 2008
46. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Network and Distributed System Security Symposium. San Diego, CA (2008)
47. Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., Robinson, S.G.: Binary translation. Communications of the ACM 36(2), 69–81 (1993)
48. Srivastava, A., Edwards, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50 (2001)
49. Srivastava, A., Eustace, A.: ATOM: a system for building customized program analysis tools. In: Programming Language Design and Implementation. Orlando, FL (1994)
50. Trilling, S.: Project green bay–calling a blitz on packers. CIO Digest: Strategies and Analysis from Symantec p. 4 (2008)
51. Vigna, G.: Static disassembly and code analysis. In: Malware Detection. Advances in Information Security, vol. 35, pp. 19–42. Springer (2007)
52. Yegneswaran, V., Saidi, H., Porras, P.: Eureka: A framework for enabling static analysis on malware. Technical Report SRI-CSL-08-01 (2008)