Cobra: Fine-grained Malware Analysis using Stealth Localized-executions

Amit Vasudevan and Ramesh Yerraballi
Department of Computer Science and Engineering
University of Texas at Arlington
Box 19015, 416 Yates St., 300 Nedderman Hall, Arlington, TX - 76019, USA.
{vasudeva, ramesh}@cse.uta.edu

Abstract

Fine-grained code analysis in the context of malware is a complex and challenging task that provides insight into malware code-layers (polymorphic/metamorphic), its data encryption/decryption engine, its memory-layout etc., important pieces of information that can be used to detect and counter the malware and its variants. Current research in fine-grained code analysis can be categorized into static and dynamic approaches. Static approaches have been tailored towards malware and allow exhaustive fine-grained malicious code analysis, but lack support for self-modifying code, have limitations related to code-obfuscations and face the undecidability problem. Given that most if not all malware employ self-modifying code and code-obfuscations, poses the need to analyze them at runtime using dynamic approaches. However, current dynamic approaches for fine-grained code analysis are not tailored specifically towards malware and lack support for multithreading, self-modifying and/or self-checking code and are easily detected and countered by ever-evolving anti-analysis tricks employed by malicious code.

To address this problem we propose a powerful dynamic fine-grained malicious code analysis framework codenamed Cobra, to combat malware that are becoming increasingly hard to analyze. Our goal is to provide a stealth, efficient, portable and easy-to-use framework supporting multithreading, self-modifying/self-checking code and any form of code obfuscation in both user- and kernel-mode on commodity operating systems. Cobra cannot be detected or countered and can be dynamically and selectively deployed on malware specific code-streams while allowing other code-streams to execute as is. We also illustrate the framework utility by describing our experience with a tool employing Cobra to analyze a real-world malware.

1. Introduction

Malware — a term used for viruses, trojans, spywares or any malicious code – is widespread today. Malware analysis — a complex process employing various coarse- and fine-grained analysis methods – provides insight into malware structure and functionality and facilitates the development of an antidote. For example, the W32/MyDoom [38] trojan and its variants propagate via e-mail and download and launch external programs using the network and registry. Such behavior, which includes the nature of information exchanged over the network, the registry keys used, the processes and files created etc., is inferred by employing coarse-grained analysis pertaining to process, network, registry, file and other related services of the host operating system (OS). Once such behavior is known, fine-grained analysis is employed on the identified areas to reveal details such as the polymorphic and/or metamorphic code layers of the trojan, its data encryption and decryption engine, its memory layout etc.

Fine-grained malware analysis is a challenging task that provides important pieces of information that are key to building a blueprint of the malware core structure and functioning that aids in the detecting and countering the malware and its variants. As an example, the W32/MyDoom trojan with variants commonly known as W32/MyDoom.X-MM (where X can be A, B, R, S, G etc.) share the same metamorphic code layers, encryption/decryption engine and similar anti-analysis schemes. The Netsky [15], Beagle [24] and Sobig [35] worms are some other examples of how malware are coded in an iterative fashion to add more features while retaining their core structure. Thus, once the core structure of a malware is documented, it becomes easy to tackle the rest of its variants as well as other malware which share a similar structure. Also, with malware writers employing more complex and hard to analyze techniques, there is need to perform fine-grained analysis of malicious code to counter them effectively.

Current research in malware analysis can be broadly categorized into static and dynamic approaches. Static approaches allow exhaustive fine-grained analysis because they are not bound to a specific execution instance. They allow detection of malicious code without actually running the program, ensuring that the malices discovered will never be executed and incur no run-time overhead. In spite of such powerful properties, static analysis has some limitations. With static analysis there is the problem that the analyzed code need not be the one that is actually run; some changes could be made between analysis and execution. This is particularly true with polymorphism [56, 47] and metamorphism [48] that are techniques employed by most if not all current generation malware. Also it is impossible to statically analyze certain situations due to undecid-

ability (eg. indirect branches). Further, static code analysis also have limitations related to code obfuscation — a technique used by malware to prevent their analysis and detection. Dynamic approaches overcome these limitations by analyzing the code during run-time ensuring that the the analyzed code is the one that is actually run, without any further alterations. Though there have been several research on dynamic coarse-grained malware analysis [23, 42, 46, 21, 51], not much has been published about dynamic fine-grained malware analysis. Currently fine-grained dynamic malware analysis can be achieved by employing debuggers and/or fine-grained instrumentation frameworks .When using a debugger such as Softice [20], WinDBG [43] etc., the basic approach is to set breakpoints on identified areas and then trace the desired code-stream one instruction at a time to glean further information. Alternatively one could also employ a fine-grained instrumentation framework such as Pin [34], DynamoRIO [8] etc., for automated tracing of code-streams for a given range of code. However, these tools are not equipped to handle malicious code and have severe shortcomings in the context of malware.

Current debugging and fine-grained instrumentation techniques can be easily detected and countered by the executing malware code stream. As an example, the W32/HIV [37], W32/MyDoom [38], W32/Ratos [49], and their variants employ techniques such as code execution timing, where the malware time their executing code therby easily detecting that they are being analyzed (since debugging and/or automated code tracing incur latency that is absent during normal execution). Further they contain ad-hoc detection schemes against popular debuggers such as Softice, WinDBG, etc. Current debugging and fine-grained instrumentation techniques do not carry support for self-modifying and/or self-checking (SM-SC) code . Most if not all malware are sensitive to code modification with subtle anti-analysis techniques and code obfuscations that defeat breakpoints in debugging and the process of automated code tracing using fine-grained instrumentation frameworks. As an example, the W32/MyDoom and the W32/Ratos employ integrity checking of their code-streams with program-counter relative code modification schemes which render software breakpoints and current fine-grained instrumentation frameworks useless. Malware that execute in kernel-mode are even tougher to analyze using current dynamic fine-grained techniques, since they have no barriers in terms of what they can access. For example, the W32/Ratos employs multithreaded polymorphic/metamorphic code engine, running in kernel-mode and overwrites the interrupt descriptor table (IDT) with values pointing to its own handlers. Current fine-grained instrumentation frameworks do not carry support for kernel-mode code and do not support multithreading. Current debugging techniques provide kernel-mode code support but do not support multithreading in kernel-mode. Furthermore, recent trend in malware has been to employ debugging mechanisms supported by the underlying processor within their own code, thereby effectively preventing analysis of their code using current debugging techniques. Examples include W32/Ratos, which employs the single-step handler (used for code tracing) to handle its decryption in kernel-mode and W32/HIV which uses debug registers (used for hardware breakpoints) for its internal computation. This situation calls for a dynamic fine-grained code-analysis framework specifically tailored towards malware.

This paper presents the concept of *stealth localized-executions* and presents Cobra, a realization of this concept that enables dynamic fine-grained malware analysis in commodity OSs in a completely stealth fashion. Our goals are to provide a stealth, efficient, portable and easy-to-use framework that supports multithreading, SM-SC code and code obfuscations in both user- and kernel-mode while allowing selective isolation of malware code-streams. By *stealth* we mean that Cobra does not make any visible changes to the executing code and hence cannot be detected or countered. The framework employs subtle techniques such as slice-colascing and slice-skipping to provide a *efficient* supervised execution environment. Cobra currently runs under the Windows (9x, NT, 2K and XP) and Linux OSs with minimal dependency on the host OS and employs an architecture specific disassembler for its inner functioning. This makes the framework *portable* to other platforms (OS and architecture) with ease. Cobras API is simple yet powerful making the framework *easy-to-use*. Analysis tools are usually coded in C/C++ using the framework API. The API is architecture independent while allowing the tools to access architecture specific details when necessary. The framework allows what we call *selective-isolation* that allows fine-grained analysis to be deployed on malware specific code-streams while allowing normal code-streams to execute as is. The framework also allows a user to tie specific actions to events that are generated during the analysis process in real-time. To the best of our knowledge, Cobra is the first fine-grained malware analysis framework that provides a highly conducive environment to combat malware which are ever-evolving and increasingly becoming hardened to analysis.

This paper is organized as follows: We begin by considering related work on malware analysis and compare them with Cobra in Section 2. We then present an overview of Cobra in Section 3. We follow this with a detailed discussion on design and implementation issues in Section 4. In Section 5, we discuss our experience with one of our tools employing Cobra to analyze a real- world malware and present some performance numbers for the framework in Section 6. Finally, we conclude the paper in Section 7 summarizing our contributions with suggestions for future work.

2. Background and Related Work

A malware is a program that can affect, or let other programs affect, the confidentiality, integrity, the data and control flow, and the functionality of a system without explicit knowledge and consent of the user [4]. A classification of

malware according to its propagation method and goal can be found in [39, 7]. Given the fact that malware is widespread today and knowing the devastating effects that malware can have in the computing world, detecting and countering malware is an important goal. To successfully detect and counter malware, one must be able to analyze them in both coarse- and fine-grained fashion — a complex process that is termed Malware analysis. In this section we will discuss some of the existing research in the area of malware analysis and compare them with Cobra. A complete annotated bibliography of papers on malware analysis and dectection can be found in Singh and Lakhotia [45]. Current research in malware code analysis and detection can be broadly categorized into static and dynamic approaches [4]. Both methods have their advantages and disadvantages and are complimentary. Static approaches to malware analysis can be used first, and information that cannot be gleaned statically can then be dynamically acscertained.

Static approaches to malware analysis extend techniques related to verifying security properties of software at a source level [1, 6, 9, 10, 26, 32] to binary (since for a malware, in most if not all cases, there is no source-code availability). Bergerson et. al [5, 3] present techniques that disassemble the binary and pass it through a series of transformations that aid in getting a high-level imperative representation of the code. The binary is then sliced to extract the code fragments critical from the standpoint of security and malicious code. Giffin et al [22] disassemble a binary to remotely detect manipulated system calls in a malware. Many malware detection techniques are based on static analysis of executables. Kruegel et al [28] employ static binary analysis to detect kernel-level rootkits. SAFE [12] and Semantic-Aware Algorithm [14] are other examples of malware detection algorithms employing similar static analysis techniques. Static approaches allow exhaustive fine-grained analysis because they are not bound to a specific execution instance. They enable detection of malicious code without actually running the program. Therefore, the malices discovered will never be executed. On the performance side, there is no run-time overhead associated with a static analysis. After just one analysis, the program can run freely. In spite of these beneficial properties, there are some limitations. The main problem with static code analysis is that the analyzed code need not be the one that is actually run; some changes could be made between analysis and execution. This is particularly true in self-modifying techniques such as polymorphism [56, 47] and metamorphism [48] that are ubiquitous in most malware code streams. Static approaches also have limitations related to code obfuscation [18, 19, 55]. They employ a dissassembler as an essential step in their analysis procedure. Linn and Debray [31] and Christodorescu and Jha [13] demonstrate that simple obfuscations can thwart the dissassembly process. While Kruegel et al [27] present techniques for disassembling obfuscated executables, they are unable to handle situations such as indirect obfuscation [41, 53], instruction overlap [17] etc.

Dynamic approaches analyze malware code during runtime. Though dynamic approaches incur a run-time overhead and are non-exhaustive, they overcome the main limitation of static approaches in ensuring that the the analyzed code is the one that is actually run, without any further alterations thereby supporting self-modifying code and code obfuscations. Dynamic approaches can further be categorized into coarse-grained and fine-grained approaches. Coarse-grained dynamic approaches are very useful in capturing the behavior of a malware at a high level.Janus [23] provides a secure environment to execute untrusted applications. It intercepts and filters dangerous system calls under Solaris to reduce the risk of a security breach by restricting the program's access to the operating system. DaMon [21] is a dynamic monitoring system uses a similar technique to dynamically enforces a security policy to stop certain malicious actions on resources such as ports, registry, processes etc. SPiKE [51] is a stealth software framework that works on the principle of dynamic monitoring, to log activity of malware code streams. Most network intrusion detection systems [42] and honeypots [46] also hinge on dynamic coarse-grained analysis for their functioning.

In contrast, fine-grained dynamic approaches help to understand the inner structure of the malware in terms of its run-time code envelopes, its data encryption/decryption engine, memory layout, anti-analysis techniques etc. Though there have been several research on dynamic coarse-grained malware analysis, not much has been published about dynamic fine-grained malware analysis. Cohen [16] and Chess-White [11] propose a virus detection model that executes in a sandbox. However, their model is not generic and does not allow fine grained analysis at a level that can be used to document the internal workings of a malware. Debuggers such as Softice [20], WinDBG [43], GDB [33] etc. enable dynamic fine-grained analysis in both user- and kernel-mode. Though current debuggers to some extent, support self-modifying and code obfuscations, they are not tailored specifically towards malware analysis and fall prey to several anti-debugging tricks employed by them [50]. While code analysis using a debugger is manual (one has to trace instructions manually), tools such as Pin [34], Valgrind [8], DynamoRIO [40], Strata [44], Diota [36] etc. enable automated code tracing by employing a virtual machine approach. However, these tools are designed for normal program instrumentation and hence do not carry support for SM-SC code and code obfuscations. Further these tools do not carry adequate support for multithreading and cannot handle code in kernel-mode. Hypervisors such as VMWare [52], QEmu [2] etc. are able to handle multithreading in both user- and kernel-mode code efficiently, but do not carry support for SM-SC code. Also, they are not tailored towards malware and can be detected and countered [48].

In comparison Cobra is a dynamic fine-grained malware analysis framework that overcomes the shortcomings of current research in dynamic fine-grained malware analysis by providing a stealth supervised code execution environment

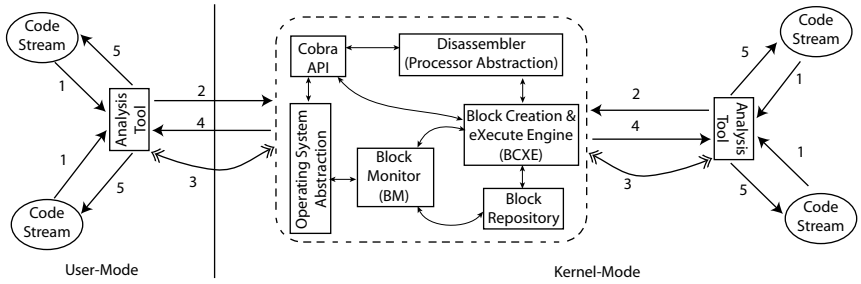

Figure 1. High Level Architecture of Cobra

that can handle multithreading, self-modifying and any form of code obfuscation in both user- and kernel-mode. Cobra cannot be detected or countered in any fashion and supports both manual and automated code tracing, providing insight into executing malware code streams at runtime. The framework supports selective isolation whereby one can deploy fine-grained analysis of malware specific code-streams while co-existing with normal code-streams in real-time.

3. Framework Overview

Fine grained malware analysis using Cobra is facilitated by a technique that we call *stealth localized-executions*. The basic idea involves decomposing (*slicing*) a target code-stream into several instruction blocks (*slices*) which are then executed, one block at a time, in a fashion so as to mimic the normal execution of the target code-stream. Each slice is implanted with various invisible Cobra specific code constructs (as applicable), ensuring that the framework has complete control over the executing code-stream while remaining stealth.

Figure 1 illustrates the current architecture of Cobra. The framework core consists of a code slice and execute engine (CSXE), a disassembler, a slice repository, a slice-monitor, and a framework API. The CSXE is responsible for decomposing a target code-stream into individual slices. The target code stream can be a regular or a malware specific code-stream. An architecture specific *disassembler* is employed to construct slices corresponding to the target code-stream, in a dynamic fashion. The *slice repository* functions as a local cache for storage of slices. Only code from the slice repository is executed and never any code from the target code stream. The CSXE begins slicing the target code stream at an *overlay point* and stops slicing at a *release point*, a user-defined range where fine-grained analysis is desired. This allows Cobra to be deployed and removed in a dynamic fashion on a given code-stream while allowing other code streams to execute as is, a technique we call *selective isolation*. As an example, the `W32/Ratos` trojan and its variants employ several kernel-mode threads for their functioning. An overlay point in this case could be `KiSwitchContext` (a Windows internal kernel function responsible for thread pre-emption) and a release point could be the return from this function. The *slice-monitor* employs subtle techniques involving virtual memory to protect critical memory regions during slice execution and is responsible for maintaining coherence between the target code-stream and its slices in case of self-modifying code.

As Figure 1 shows, there are typically three binary elements present during an analysis session: the target code-streams (residing in either user- and/or kernel-mode), the analysis tool employing Cobra (typically some sort of debugger), and Cobra itself. The analysis tool, for each overlay point in a target code-stream, invokes the framework for fine-grained analysis over a specified range of the code-stream (Steps 1 and 2, Figure 1). The analysis tool uses SPiKE [51] (a stealth coarse grained malware analysis framework) and/or VAMPiRE [50] (a stealth breakpoint framework) to gain control at specified overlay points in both user- and/or kernel-mode. The analysis tool then performs the required actions (processing) for specified events during the execution of the slices. An event can be: slice creation, execution of instructions within a slice, a system or library function invocation within a slice, acesss to critical memory regions within a slice etc. (Step 3, Figure 1). Cobra finally stops slicing the target code-stream at the specified release point and transfers control back to the analysis tool which then resumes normal execution of the target code-stream (Steps 4 and 5, Figure 1).

Cobra resides in kernel-mode and can capture multithreaded, SM-SC and any form of code obfuscations in both user- and kernel-mode code with ease. The framework is completely re-entrant, as it does not make use of any OS specific routines during the execution of slices and uses shared memory with its own isolation primitives for interprocess communication. The framework employs techniques such as *slice-skipping* (where standard and/or non-standard code-streams are excluded from slicing) and *slice-coalescing* (where multiple slices are composed together) for efficiency. The Cobra API is simple yet powerful to allow a tool writer to harness the complete power of the framework.

4. Design and Implementation

Our goal in designing and implementing Cobra was twofold. First, it should be able to provide a stealth supervised environment for fine-grained analysis of executing malware code-streams, supporting multithreading, self-modifying code and any form of code obfuscation in both user- and kernel-mode on commodity OSs. Second, one must be able to deploy the framework dynamically and selectively on malware specific code-streams while allowing other code-streams to execute as is. This section describes how Cobra achieves these capabilities.

4.1. Localized-Executions

Cobra decomposes a target code-stream into several groups of instructions and executes them in a fashion so as to mimic the code-stream's normal execution. This process is what we call *localized-executions* and the instruction groups are called *blocks*. A block is nothing but a straight-line sequence of instructions that terminates

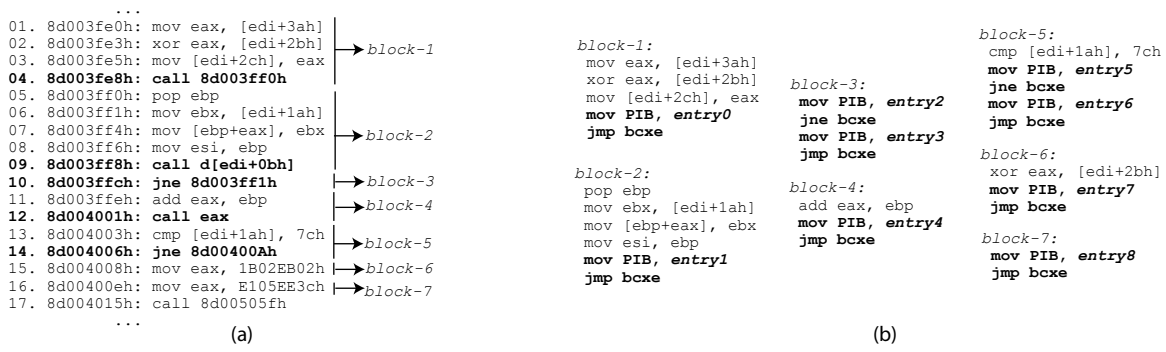

Figure 2. Block Creation: (a) Target Code-stream, and (b) Corresponding Blocks

in either of these conditions: (1) an unconditional control transfer instruction (CTI), (2) a conditional CTI, or (3) a specified number of non-CTIs. A *block-repository* contains a subset of the recently constructed blocks and acts as a framework local cache. Only blocks residing in the block-repository are executed — never the instructions in the target code-stream (hence the term *localized-executions*). Cobra's Block Creation and eXecution Engine (BCXE) is responsible for creating blocks from the target code-stream and executing them.

4.1.1. Block Creation The BCXE employs an architecture specific disassembler on the target code-stream, to discover instructions one at a time, and create the corresponding blocks. Figure 2a shows part of a code-stream of the W32/Ratos trojan and a typical block creation process. The code-fragment has been modified to remove details not pertinent to our discussion and the instructions are shown in the 32-bit assembly language syntax of the IA-32 (and compatible) processors [25].

Every block ends with a framework specific set of instructions — which we call a *Xfer-stub* — that transfers control to the BCXE. Xfer-stubs ensure that Cobra is always under control of the target code-stream being analyzed. When a block is executed, the BCXE gets control at the end of the block execution via the block xfer-stub, determines the target memory-address to create the next block from, dynamically generates a new block for the corresponding code-stream if it has not done before, and resumes execution at the newly generated block. Thus, execution of blocks follows a path which is the same as the normal execution of the target code-stream in absence of the framework. Figure 2(b) shows the blocks created by the BCXE for the code-stream shown in Figure 2(a).

The BCXE differs from VMs employed in current hypervisors [52, 2] and fine-grained instrumentation frameworks [34, 8, 40, 44, 36] in that: (a) it employs special treatment for CTIs thereby supporting SM-SC code and any form of code obfuscation (see Section 4.1.3), (b) it employs special treatment on privileged instructions and instructions that betray the real state of a code-stream and hence cannot be detected or countered in any fashion (see Section 4.2), (c) it achieves efficiency without recompiling the instructions of the target code-stream (see Section 4.3), and (d) it is completely re-entrant supporting multithreading under both user- and kernel-mode and allows tuning the level of fine-grained analysis desired.

4.1.2. Xfer-stubs Xfer-stubs, code constructs specific to Cobra, that terminates every block can be abstracted as a function that takes a single parameter (Figure 3(a)). The parameter is an index into a *Xfer-Table*, an internal data structure of the framework, which enables the BCXE to obtain run-time information on the supervised code-stream, which includes among other things, the address of the target code-stream to generate a new block from. A xfer-stub replaces the CTI (conditional or the unconditional) that terminates a block. In some cases, where block creation terminates because a predefined number of non-CTIs were reached, Cobra treats the block as ending with an unconditional branch/jump instruction and creates a corresponding xfer-stub. Figure 3(b) shows the xfer-stub implementations for conditional and unconditional CTIs on the IA-32 (and compatible) processors. For unconditional CTIs the corresponding xfer-stub simply performins an unconditional jump (JMP) into the BCXE. For conditional CTIs, the xfer-stub translates a conditional into a conditional and an explicit JMP. This ensures that the BCXE gets control for both situations where the conditional evaluates to a true and false. The parameter to a xfer-stub is passed via a Parameter Information Block (PIB) — a per-thread, framework internal memory area instead of the thread stack. This is required to prevent Cobra from being detected or countered by the malware being analyzed (see Section 4.2.2).

The xfer-table (shown in Figure 3(c)) is an array of structures, one element for each xfer-stub that is currently used by the framework. Every entry in the xfer-table consists of (1) a target-address type (TAT), (2) a target-address value (TAV), (3) the xfer-stub type (XST), and (4) additional xfer-stub parameters (if applicable). The TAT determines if the address at which the CXSE will create a new block from, is an immediate value (VALIMM) or an indirect expression (VALIND) whose value has to be evaluated by the BCXE at run-time upon entry from the xfer-stub. The TAV is a constant (when TAT is VALIMM) or an expression (when TAT is VALIND). The XST indicates whether the xfer-stub is for a standard CTI with no additional processing (XNORMAL) or a CTI that needs special processing (XSPECIAL). XST XNORMAL is used in the majority of cases while XST XSPECIAL is used to handle cases where: (1) the CTI uses the thread stack implicitly (CALL, INT etc.) (see Section 4.1.3), and (2) the framework needs to employ block specific xfer-stubs to remain stealth (see Section 4.2.1). In both cases the framework makes use of additional xfer-stub specific parameters.

Figure 3(c) shows the entries corresponding to the xfer-stubs for the blocks shown in Figure 2(b). As seen, entry0 has TAT set to VALIMM and TAV set to the constant 8d003ff0h since the corresponding xfer-stub is for a CALL instruction (Line 4, Figure 2(a)) which deals with a constant target address at which the CXSE generates the next block from. However, entry4 has TAT set to VALIND and TAV set to the expression EAX since the corresponding xfer-stub is for a CALL instruction (Line 12, Figure 2(a)) whose target

```
bcxe(entry_number);
```

(a)

```
unconditional
mov PIB, entry
jmp bcxe

conditional
mov PIB, entry_truepart
jxx bcxe
mov PIB, entry_falsepart
jmp bcxe

jxx = je, jne, jc, jnc, jb etc.
```

(b)

Entry Number	Target Address Type (TAT)	Target Address Value (TAV)	Xfer-Stub Type	Xfer-Stub Parameters
entry0	VALIMM	8d003ff0h	XSPECIAL	8d003ff0h
entry1	VALIND	[edi+0bh]	XSPECIAL	8d003ffch
entry2	VALIMM	8d003ff1h	XNORMAL	NULL
entry3	VALIMM	8d003ffeh	XNORMAL	NULL
entry4	VALIND	eax	XSPECIAL	8d004003h
entry5	VALIMM	8d00400ah	XNORMAL	NULL
entry6	VALIMM	8d004008h	XNORMAL	NULL
entry7	VALIMM	8d004010h	XNORMAL	NULL
entry8	VALIND	[edi+eax*4]	XNORMAL	NULL

(c)

Figure 3. (a) Xfer-Stub Abstraction, (b) Xfer-Stub Implementation on IA-32 (and compatible) processors, and (c) Xfer-Table

address depends on the runtime value of EAX at that point in execution. For both cases, the XST is set to XSPECIAL to indicate that the xfer-stubs require additional processing.

4.1.3. Obfuscated Code and SM-SC Code Cobra's model of employing xfer-stubs for every CTI (unconditional or conditional) enable the framework to support any form of code obfuscations, since obfuscated code rely on conditional and/or unconditional branches in between instructions for their functioning [31, 18, 19, 55]. Since every block generated by the BCXE terminates on exactly one CTI and the fact that the BCXE can handle both direct and indirect control transfers, it is guaranteed that the destination memory-address for the next block creation always points to an address from which a valid block can be constructed using the disassembler. Lines 14–16, Figure 2(a) and blocks 5–7, Figure 2(b), show an example of obfuscation in a code-stream and the corresponding blocks generated. Note how the blocks successfully un-obfuscate the code stream as it would happen during normal execution in the absence of the framework. A point to be noted is that if a block tried to include more than one CTI, the resulting block generated might escape the framework supervision during execution due to a CTI with indirect target address (e.g CTI in line 12, Figure 2(a)). However, the framework can be configured dynamically to construct blocks including multiple CTIs or coalesce existing blocks for performance enhancements (see Section 4.3) in cases where one can be certain that such blocks will not escape the framework supervision.

Cobra handles CTIs that employ the stack implicitly in a special fashion. As an example, on the IA-32 (and compatible) processors, the CALL instruction transfers control to a procedure unconditionally. The instruction pushes the return address on the stack as a part of its semantic which is then popped by a corresponding RET instruction to resume execution at the caller. This property is exploited by most if not all SM-SC code which, instead of using the RET instruction, pop the value into a register and use it to access their code in a position independent manner for modification and/or integrity checking. Cobra ensures that the program-counter of the target code stream is always reflected in the corresponding xfer-stub for such instructions thereby supporting SM-SC code. block 1, Figure 2(b) shows the cobra xfer-stub corresponding to the CALL instruction shown in line 4, Figure 2(a). As seen the corresponding xfer-table entry, entry0 (Figure 3(c)), sets the XST to XSPECIAL and stores the original program-counter as the parameter. The BCXE thus pushes the original program-counter on the stack before proceeding with block creation at the destination address.

4.1.4. Block Execution Localized-executions start from a user-defined point — which we call an *overlay point* — in a target code-stream. An overlay point is the memory-address (typically a OS and/or a library function address) in a target code-stream from where fine-grained analysis is desired. An overlay point under Cobra is defined by employing SPiKE [51] (a stealth coarse-grained malware analysis framework) and/or VAMPiRE [50] (a stealth breakpoint framework). Once execution reaches an overlay point, Cobra is invoked to start fine-grained analysis until a *release point* is reached. A release point is the memory-address in a target code-stream where Cobra relinquishes supervision and lets the code-stream execute in a normal fashion. A overlay point and its corresponding release point thus establish a fine-grained analysis range on a target code-stream under Cobra, while allowing other code-streams to execute as is — a technique we call *selective isolation*. Under Cobra, one can specify multiple overlapping and/or non-overlapping overlay and release points for a target code-stream. The framework also supports nesting of overlay and release points and allows release points to be infinite, in which case the complete thread containing the target code-stream is constantly run under Cobra's supervision, until the thread terminates or the framework is invoked to stop localized-executions.

As an example, the W32/Ratos trojan runs under the Windows OS and employs several kernel-mode threads for its inner functioning. One such kernel-mode thread replaces the default single-step handler in the Interrupt Descriptor Table (IDT) with a trojan specific handler. With Cobra, we employ *KiSwitchContext* (an internal Windows kernel function responsible for thread pre-emption) as the first overlay point to execute each of the trojan kernel-mode threads under the supervision of Cobra with infinite release points. Upon detection of an access to the single-step vector in the IDT via a Cobra generated event (see Section 4.1.5), we employ the destination address of the single-step handler as our second overlay point (with the corresponding release point being the return from exception), thereby allowing us to study the W32/Ratos single-step handler in further detail. All this is done while co-existing with other OS user- and kernel-mode threads and exception handlers.

Cobra's BCXE executes individual blocks in an unprivileged mode regardless of the execution privilege of the target code-stream. This ensures that Cobra has complete control over the executing instructions. The framework can also monitor any access to specified memory regions, the OS kernel and resources, dynamic libraries etc.

Cobra employs the virtual memory system combined with subtle techniques for memory access monitoring. On the IA-32 (and compatible) processors, for example, Cobra elevates the privilege level of specified memory regions and critical memory structures such as page-directories/page-tables, the IDT, the descriptor tables (GDT and LDT), task state segments (TSS) etc. by changing their memory page attributes and installs its own page-fault handler (PFH) to tackle issues involving memory accesses. The PFH also facilitates hiding framework specific details in the page-table/page-directories and the IDT while at the same time allowing a code-stream to install their own handlers and descriptors in these tables. Cobra employs stealth-implants (see Section 4.3.1) to support supervised execution of privileged instructions in the event that the target code-stream runs in kernel-mode.

Cobra does not make use of any OS specific functions within its BCXE. The disassembler employed by the framework is completely re-entrant. The framework employs a per-thread PIB for the block xfer-stubs, does not tamper with the executing stack and employs subtle techniques to remain stealth (see section 4.3.1). These features enable the framework to support multithreading since the executing threads see no difference with or without CORBA in terms of their registers, stack contents or time block. Cobra also supports automatic thread monitoring for a specified process or the OS kernel. This is a feature that automatically blocks every code-stream associated with a target process. Thus an entire process can be executed under Cobra by specifying the process creation API as an intial overlay point and allowing the framework to automatically insert overlay points thereafter on every new thread or process associated with the parent.

4.1.5. Events and Callbacks Cobra generates various events during block execution. These include block creations, stealth implants, begin/end execution of a whole block, execution of individual and/or specific instructions within a block, system calls and standard function invocations, access to user defined memory regions, access to critical structures such as page-directories/page-tables, IDT, GDT etc. An analysis tool employing Cobra can employ event specific processing by registering *callbacks* — functions to which control is transferred by the framework to process a desired event during block execution. Callbacks are passed all the information they need to change the target code-stream registers and memory (including the stack), examine the block causing the event and instructions within it. A callback can also establish a new overlay point during an analysis session. Events and Callbacks thus facilitate tuning the level of fine grained analysis from as fine as instruction level to progressively less finer levels.

A typical analysis process in our experience would employ events on block creations, begin/end of block executions, access to critical memory regions and any stealth implants before doing an instruction level analysis of blocks. As an example, if one considers the W32/Ratos, it overwrites the IDT single-step handler entry with a value pointing to its own single-step handler within a polymorphic code envelope. A IDT-write event can be used to obtain the trojan single-step handler address in the first place. The callback for the IDT-write event would use the single-step handler address as an overlay point to further analyze the trojan single-step handler in a fine-grained fashion. The events of block creation, begin/end block executions can then be used to build an execution model of the polymorphic code envelope. The inner working of the single-step handler can then be studied by an instruction level analysis on identified blocks.

4.2. Stealth Techniques

Block execution occurs at an unprivileged level which can cause problems with privileged instructions in the event that the target code-stream runs in kernel-mode. Also, certain situations can result in the betrayal of the real state of an executing code stream. Blocks can contain instructions which read the machine state but do not cause an exception and thus escape the BCXE. As an example the RDTSC instruction can be used to obtain the relative execution time of a region of code which will be more in the case of a block being executed than normal execution. Certain instructions silently change the behavior of the executing code. An example would be a POPF instruction which sets the trap flag resulting in single step exceptions for every instruction executed thereafter. A malware code stream could also employ detection schemes against the framework by accessing structures such as the stack, page-tables, descriptor tables (IDT, GDT and LDT) etc. Cobra employs a host of techniques to tackle issues involving privileged instructions and the framework stealthness.

4.2.1. Stealth-Implants Cobra scans a block for privileged instructions and instructions that betray the real state of the executing code-stream and replaces them with what we call *stealth-implants*. These are Cobra code constructs that aid in supervised execution of privileged instructions and the framework stealthness, while preserving the semantics of the original instructions in the target code-stream. Stealth-Implants only take place on blocks and never on the original code. Thus they are undetectable by any intergrity check as such checks always operate on the original code-stream. Cobra inserts stealth-implants for various instructions and employs a host of antidotes for various possible ways in which a malware could detect the framework. However, due to space constraints we will concentrate on the discussion of a few important instructions on the IA-32 (and compatible) processors and techniques that can be used to detect the framework and their antidotes.

IA-32 (and compatible) Processor Instruction	Stealth Implant
`rdtsc`	`mov eax,cobra_tcounter *`
`push segreg` `pop segreg`	`push cobra_segreg` `pop cobra_segreg *`
`mov destination,segreg` `mov segreg,source`	`mov destination,cobra_segreg` `mov cobra_segreg,source *`

`sidt destination`
`sgdt destination`
`sldt destination`
`str destination`
	`mov destination,cobra_idtclone`
	`mov destination,cobra_gdtclone`
	`mov destination,cobra_ldtclone`
	`mov eax,cobra_tssselector`

`mov drx/crx,source` `mov destination,drx/crx`	`mov cobra_drx/cobra_crx,source *` `mov destination,cobra_drx/cobra_drx`
`in al/ax/eax,port` `out port,al/ax/eax`	`mov al/ax/eax,cobra_valuefrom_port *` `mov cobra_valueto_port,al/ax/eax *`
`pushf` `popf`	`push cobra_eflag` `pop cobra_eflag *`

* Implant Executed via an Xfer-Stub

Figure 4. Stealth Implants

Localized-executions results in increased latency due to block creations and the xfer-stubs transferring control to and fro the BCXE during block execution. Such latency is not present during the normal execution of the target code-stream. A malware could use this fact to detect if its being analysed. As an example, a malware could use the RDTSC instruction to obtain the amount of clock-cycles that have elapsed since system-bootup and obtain a relative measurement of its code fragment execution. A malware could also use the real-time clock (RTC) to perform a similar latency detection in kernel-mode via the IN and the OUT I/O instructions. Fig-

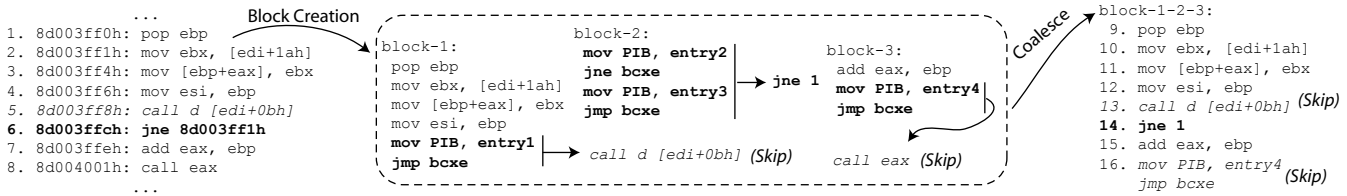

Figure 5. Skipping and Block-Coalescing

ure 4.2.1 shows the stealth-implants corresponding to such instructions. The RDTSC instruction stores the return value in the EAX register. The RTC detection makes use of the RTC I/O ports using the privileged IN instruction. Cobra replaces such privileged instructions with xfer-stubs that transfer control to the BCXE which then locally executes these instructions at a high privilege level and returns the result. For example, Cobra replaces the RDTSC instruction with a regular MOV instruction that stores the value of Cobra's internal processor-counter to the EAX register (Figure 4). A point to note is that not all stealth-implants transfer control to the BCXE. Most instructions which store a value into a destination memory operand can have a stealth-implant without an xfer-stub directly replacing the instruction.

Certain malware code streams can employ using debugging techniques for their own execution. A malware for example can employ the POPF instruction to set the processor trap-flag. This results in a single-step exception being invoked. The malware can then use the single-step handler to perform the actual functionality (eg. W32/Ratos). A stealth implant in this case will replace the POPF instruction with a xfer-stub that tranfers control to the BCXE. The BCXE will then examine the trap-flag and will automatically generate a single-step exception for every instruction thereafter until the trap-flag is clear. Some code-streams running in kernel-mode can also employ the hardware debugging registers themselves for computation. (eg. W32/HIV and W32/Ratos). The debugging registers can also be used by the malware to set breakpoints within its code-streams. Cobra handles such issues by replacing access to such debug registers with stealth-implants and can generate breakpoint exceptions by monitoring such registers.

Malware code streams can also use instructions such as PUSH, VERW and ARPL in both user- and kernel-mode to obtain the selector for the executing code segment. Since Cobra executes the blocks at an unprivileged level such instructions will reflect an unprivileged code selector and can be used as a detection mechanism against the framework. However, Cobra's stealth-implants in this case replace such instructions to reflect the actual value of the executing code segment selectors.

4.2.2. Cloning and Other Issues A malware can access critical system structures in order to detect that its being analyzed using Cobra. For example a malware might try to obtain the PFH address and compare it with the system default values (which for certain OSes lies within a fixed range regardless of their version) in order to detect the framework. Similarly it might try to check the page-attributes of certain memory regions (eg. its code and data) which can have their attributes elevated due to memory access monitoring by Cobra. Further a malware can also install its own fault handlers in the IDT for its functioning. Cobra uses a technique that we call *cloning* to hide the framework while at the same time allowing the malware to access such critical structures. The framework maintains a copy of critical memory regions such as the page-tables/page-directories, IDT, GDT etc. that reflect their system default contents initially. The framework PFH tackles issues such as reads and/or writes to such critical structures by presenting the clone of these memory regions, thereby fooling the malware into thinking that it is operating on the original memory regions. Stealth-Implants for certain instructions involving control registers such as CR3 (used to obtain the page-directory base address) and instructions such as SIDT, SLDT and SGDT present the addresses of the cloned memory regions instead of the original.

Localized-executions leads to a couple of issues a malware could exploit to detect the framework during run-time. A malware in a multithreaded fashion can use a thread context capture function (under Windows OSs the GetThreadcontext API and under Linux, the ptrace API) to obtain the current program-counter and stack contents for it executing threads. However, since the thread code-stream is being executed by Cobra, the values of the program-counter and the stack will be different than in the normal course of execution. Cobra instruments such APIs using a stealth coarse-grained instrumentation framework, SPiKE [51] and presents the original value of program-counter and the original thread stack. A point to be noted is that Cobra has no effect on the thread stack of the target code-stream. The framework employs a local stack (different from the currently executing thread stack), that is switched to upon entry from an xfer-stub. Also, the xfer-stubs make use of the PIB to pass parameters to the BCXE, thereby ensuring that the thread stack is left untouched. This prevents Cobra from being detected using stack overflow mechanisms.

Every instance of Cobra's deployment is different in the form of any privileged modules, environment variables, configuration files and code streams. Thus, no malware can detect the framework by searching these elements for a pattern. This also allows load/store instructions within blocks that access memory regions pertaining to the framework code/data, to be executed directly without Cobra's intervention.

4.3. Skipping and Block-Coalescing

A block may contain an instruction that transfers control to a code-stream following a standard execution semantic (e.g system calls, standard library calls etc.). Localized-executions of such standard code-streams result in increased latency and are in most cases undesirable (since they do not form a part of the malware being analyzed). For example, consider the code fragment of the W32/Ratos trojan as shown in Figure 5. The CALL instruction in line 8, Figure 5 uses an indirect target address, but is found to transfer control to a standard code-stream (in this case the VirtualProtect system call). Localized-execution of the system call in this case is meaningless since it does not contribute towards the analysis process. Note that the system call invocation does have a bearing on the malware

functionality at a coarse level, but does not have any implication on the fine grained analysis of the malware code stream. Cobra can identify such standard code-streams dynamically and exclude them from the slicing process thereby reducing the latency that might occur in trying to execute an OS or standard piece of code. We call this technique *block-skipping*. block-skipping can also be applied to non-standard code-streams during the analysis process. This might be used to exclude already analyzed code-streams from the slicing process thereby improving the performance. As an example, line 5 of Figure 5 shows a CALL instruction which performs an integrity check over a specified region of code. The code stream concealed behind this CALL never changes in its semantics and can be thus be skipped after analyzing it once.

Execution of blocks under Cobra involve control transfers to and fro the CXSE. These trasnsfers result in the saving and restoration of the processor registers which contribute to the framework latency. This is more important inr code-streams employing loops since every iteration of the loop will involve invoking the CXSE. The framework employs a technique that we call *block-coalescing* to minimize latency due to such code constructs. In this technique, a group of blocks are brought together to form a single block preventing multiple control transfers to the BCXE. Let us consider the code-fragment shown in Figure 5. Here we see a localized loop implementing some form of integrity check. The figure shows the blocks associated with the code fragment. From our analysis session, we found the loop to execute close to 50 times. Figure 5 shows the block-coalesced version where blocks 1–3 have been colaesed thus reducing the number of transfers to the BCXE. block-Coalescing is a powerful mechanism that produce blocks that are very similar to the original code-stream and can be locally executed with minimal latency while ensuring that Cobra is still under control of the executing code-stream. block-Coalescing is performed by a user-defined callback that chooses the blocks to participate in the coalescing process. With the above example, the first few instance of the loop executes normally producing blocks 1–3, Figure 5. Once the instructions in the blocks are analyzed the blocks can be coalesced together so that future iterations execute with minimal latency. Note from line 16, Figure 5 that the BCXE gets control once again after the loop has been executed without any intervention.

Cobra supports block-coalescing for self-modifying code by employing a subtle technique involving virtual memory. The idea is to set the memory page attributes of the executing code-streams to read-only. Thus, when a write occurs to such code regions due to self-modification, Cobra's PFH gets control and the blocks corresponding to such code regions are purged and re-created subsequently — a process we call *block-purging*.

4.4. Framework API

Anaylsis tools that employ Cobra are usually written in C/C++ using the framework API. The API is easy-to-use and is designed to be platform independent whenever possible allowing tool code to be re-usable while allowing them to acess platform specific details when necessary. In Figure 6, we list a partial code of a plugin for WiLDCAT, our prototype malware analysis environment employing Cobra. The plugin is specific to the analysis of the W32/Ratos trojan and its variants and aids in gleaning information about the malware internals.

The main interface to Cobra is provided in the form of four APIs: cobra_init, cobra_action, cobra_start and cobra_comm. As seen from Figure 6, an analysis tool using Cobra, first initializes the framework from within its main routine using the cobra_init API. The analysis tool also initializes support frameworks used by Cobra such as VAMPiRE [50] and/or SPiKE [51] at this stage and establishes the first overlay point using their API (in our example in Figure 6, SPiKE is used to setup an overlay point on KiSwitchContext, a Windows internal kernel function). When the overlay point gets control, the tool uses the cobra_action API to setup callbacks for various events to be handled once slicing begins. This is shown in Figure 6 where events for IDT access (AC_IDTREAD and AC_IDTWRITE) and block creation (AC_blockCREATE) are selected. With Cobra, a single callback can handle multiple events. Finally, the analysis tool invokes the cobra_start API to establish the range in which fine grained analysis is desired. The starting address of slicing depends upon the overlay point and in our example happens to be the parameter to the KiSwitchContext function which is the thread address to switch context to. This is retrieved from the DRIFTERINFO structure for the instrument under SPiKE [51]. The release point in the example is the code label release and is the point where the thread returns. A point to be noted is that most kernel-mode threads never return, but the establishing such a release point ensures that the framework stops slicing in case the thread returns.

During block execution, Cobra invokes the callback (actionhandler in Figure 6) when the desired events occur. This is in the form of a single parameter of type ACTIONINFO to the callback. Among other fields, the ACTIONINFO structure contains the event-type, the event parameters and a pointer to a buffer containing the instructions for the current block. The callback can then take further steps to process the desired event. In our example, the IDT write event (AC_IDTWRITE) is used to obtain the single-step handler address and to setup another overlay point on the single-step handler. The Cobra API cobra_comm is used to communicate between a call-

```
--- w32ratos_plugin.c ---
#include <cobra.h>
#include <spike.h>
...
...
// main function for a plugin under WiLDCAT
void plugin_main(){
  ...
  spike_init(); //initialize SPiKE
  cobra_init(); //initialize Cobra
  ...
  //establish overlay point
  addr1= spike_addr("ntoskrnl.exe", "KiSwitchContext");
  spike_insertprobe(addr1, overlaypoint);
  ...
}
...
...
void actionhandler(ACTIONINFO *a){
  ...
  //process desired event
  switch(a->eventtype){
    case EVENT_IDTWRITE: //write to IDT
       ...
       //new overlay point on single-step handler
       spike_insertprobe(a->eventparam[0], overlaysshandler);
  ...
}
...
void overlaypoint(DRIFTERINFO *d){
  ACTIONCHAIN ac.
  ...
  //select events to be processed
  cobra_action(ac, AC_BLOCKCREATE, actionhandler);
  cobra_action(ac, AC_IDTREAD | AC_IDTWRITE, actionhandler);
  ...
  //start localized-execution at overlay point
  spike_cleanupstack(&d);
  cobra_start(d->origfunc, &release, ac);

release:
  spike_longjmp(&d); //resume normal execution
}
...
```

Figure 6. W32/Ratos Plugin for WiLDCAT, a tool employing Cobra

back and other modules of the analysis tools, in situations where the analysis tool is constructed to be different processes. In our example, WiLDCAT runs the plugin in the address space of the target process/thread while the tool itself runs as a separate process.

5. Experience

This section will discuss our experience with Cobra in analyzing a real-world malware, W32/Ratos [49], thereby illustrating the framework utility. We chose the W32/Ratos trojan for our analysis and discussion since it operates in both user- and kernel-mode containing complex code envelopes and a variety of anti-analysis tricks that one would typically encounter in recent malware. Before we proceed to discuss our experience with the W32/Ratos trojan, a few words regarding the analysis environment are in order. To validate Cobra, we use our prototype malware analysis environment codenamed WiLDCAT. The current version of WiLDCAT runs under the Windows OSs (9x, 2K and XP) on the IA-32 (and compatible) processors. It makes use of Cobra (apart from other frameworks such as SPiKE [51] and VAMPiRE [50]) for real-time malware analysis in both coarse- and fine-grained fashions. For analysis purposes, an Intel Xeon 1.7 GHz 32-bit processor with 512 MB of memory running the Windows XP OS and WiLDCAT was used.

The W32/Ratos with variants known as W32/Ratos.A, Backdoor.Ratos.A, Backdoor.Nemog etc. is a trojan that runs under the Windows OSs and is usually deployed as a second stage malware after being downloaded as a payload of other worms such as W32/MyDoom [38] and its variants. The W32/Ratos and its variants (hereon referred to as W32/Ratos collectively) includes a kernel-mode component that executes as a service and a user-mode process. Once in place, the trojan will allow external users to relay mail through random ports, and to use the victim's machine as an HTTP proxy. The trojan also has the ability to uninstall or update itself, and to download files by connecting to various predefined list of IP addresses and ports on various public file sharing networks.

The internal structure of the W32/Ratos is as shown in Figure 7. The malware consists of a user-mode process (named DX32HHLP.EXE) and a kernel-mode component (named DX32HHEC.SYS) running as a service. DX32HHLP.EXE is responsible for a bulk of the malware functionality with the kernel-mode component aiding encryption/decryption and initial deployment. The W32/Ratos and its variants cannot be analyzed using current static approaches. The trojan employs a complex multithreaded metamorphic code envelope for both its user- and kernel-mode components. The W32/Ratos employs a multilevel encryption/decryption scheme employing algorithims which resemble the TEA [54] and IDEA [29] ciphers. It employs a windowed mechanism where the code is decrypted in a preallocated region of memory thus allowing only a small portion of it to be decrypted in memory at a given time.

The metamorphic code envelopes access code and data using relative pointers for self-modifications and integrity checks (see Section 5.1). Both the user- and the kernel-mode components employ several anti-analysis tricks to prevent themselves from being analyzed. Such checks include among other things, code execution time profiling and privilege level detections. These techniques are not handled by current dynamic fine-grained code-analysis frameworks and the malware upon such detections will remain dormant or put the system into an unstable state. The trojan code envelop is multithreaded in both user- and kernel-mode and in certain cases achieve decryption via a single-step exception (see Section 5.2)

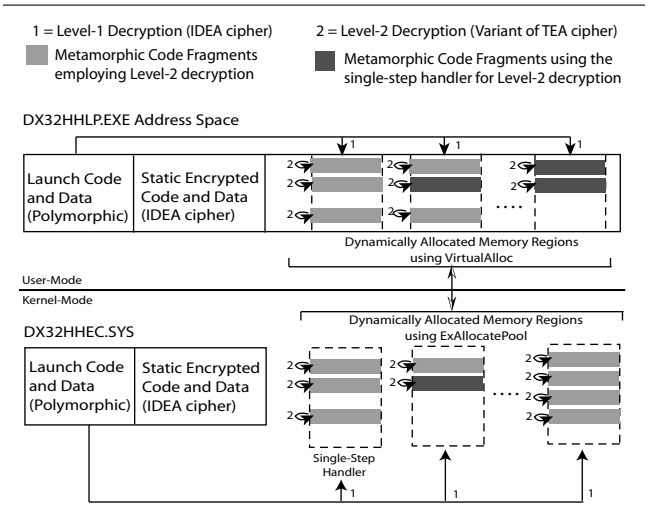

Figure 7. W32/Ratos Internal Structure

and also employ debugging registers for their internal computation, thereby defeating current debuggers. The kernel-mode component of the trojan achieves a stealth profile by hooking several Windows OS kernel functions such as ZwQueryDirInformation and ZwQwerySystemInformation.

The following paragraphs discuss in further detail, our experience in analyzing the W32/Ratos employing Cobra. The discussion serves as the basis on which we were able to document the inner structure (shown in Figure 7) and operation of the malware, thus illustrating the utility of the framework. For purposes of discussion, we will proceed to look at some simplified code fragments of the W32/Ratos under different analysis sessions with WiLDCAT, our prototype malware analysis environment. The code fragments are shown in the 32-bit assembly language syntax of the IA-32 (and compatible) processors.

5.1. Metamorphism and Memory Layout

Our first step in analyzing the W32/Ratos started with a coarse-grained analysis of the malware, thereby documenting its behaviour at a high level. We used SPiKE [51], a stealth coarse-grained malware analysis framework, to obtain the different system calls that were issued by the malware. It is found that the W32/Ratos issues several calls to ExAllocatePool, VirtualAlloc, VirtualProtect and MMProbeandLockPages, functions which allocate a range of memory outside of the existing code and/or data and change the attributes of the allocated memory range. The malware also creates several threads in both kernel- and user-mode as exempified by calls to PsCreateSystemThread and CreateThread APIs. We proceed to do a deeper investigation by using KiSwitchContext as our first overlay point and invoking Cobra for fine grained analysis of the threads created by the malware. KiSwitchContext is an internal windows kernel function that is used for thread pre-emption. We also instruct Cobra to generate events on memory accesses (reads, writes and/or executes) to any memory region allocated with the ExAllocatePool and VirtualAlloc APIs.

Consider the code fragment shown in Figure 8a. This (and several other) code fragments show the trojan metamorphic envelop in action. The code fragment shown here was obtained on events generated as a result of a write to the memory regions allocated by the malware using the ExAllocatePool and VirtualAlloc APIs and is a

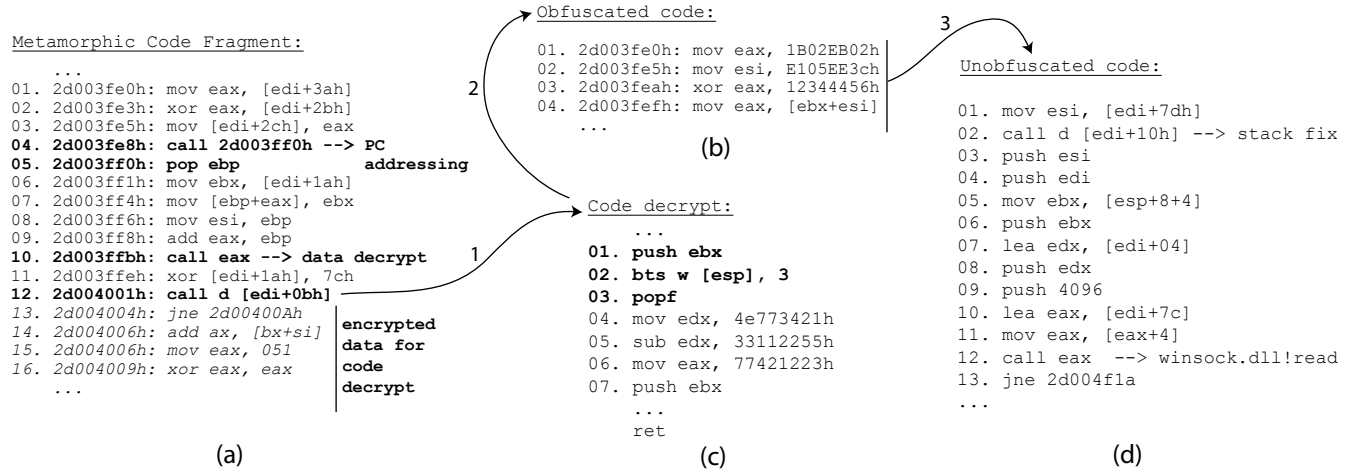

Figure 8. (a)-(d) W32/Ratos Metamorphic Code Fragment

coalesced version of the blocks that were executed by Cobra during the analysis process.

The W32/Ratos employs a multilevel decryption mechanism and metamorphic code engine to execute its internal code. The first level of decryption results in code fragments that are generated on the fly and executed in the memory regions allocated via the VirtualAlloc/ExAllocatePool APIs. Though the actual instructions of the decrypted code can vary from session to session of analysis due to the metamorphic nature, they have a regular pattern as shown in an example code fragment in Figure 8a. Here we see a fragment of a trojan subroutine that is responsible for its update feature.

Every code fragment generated by the metamorphic engine have two parts: a self-modifying second level decryption and an encrypted data that is responsible for the actual functioning of the fragment. The self-modifying section uses a program counter relative addressing (lines 4–5, Figure 8a) and modifies the code fragment by employing a second level decryption. Some of the metamorphic code fragments employ a second level decryption via a single-step handler in the kernel-mode component of the trojan (see Section 5.2). The second level of decryption changes the actual instructions at the start of the code fragment and creates an obfuscated code section by employing the encrypted data of the metamorphic code fragment. It is also responsible for setting up the data for the new instructions that are decrypted. (lines 10 and 12, Figure 8a). The memory regions allocated via the VirtualAlloc/ExAllocatePool APIs thus act as a runtime window for code and data decryption on the fly. This technique ensures that not all of the malware code and/or data are in the decrypted form at a given time thereby making the analysis process harder. With Cobra however, it is relatively easy to document such techniques by using events corresponding to memory region accesses. Figure 7 shows the memory layout of the trojan as a result.

The code generated by the second layer of decryption overwrites the start of the metamorphic code fragment and looks like a sequence regular instructions but are in fact obfuscated (Figure 8b). The actual instructions that are executed depend on the constants to the various instructions. Cobra's block execution events quickly reveal the actual code behind the obfuscated instructions as shown in Figure 8d. Upon further investigations it is found that the obfuscated instructions of the code fragment is actually a part of the trojan update feature that results in downlaods from certain public file sharing networks.

A metamorphic code fragment in certain cases also include an integrity check on the code fragment to see if there has been tampering. In such cases, the trojan simply ensures that the second level decryption is voided, which results in the obfuscated fragment not being decrypted leading to spurious faults during the analysis process. Manual patching of such detections is a tedious process since the malware employs several such integrity checks throught its functioning. With Cobra however this is not of any consequence since the framework ensures that the original code is left untouched in memory.

5.2. Decryption and Anti-Analysis Tricks

The W32/Ratos employs a multilevel decryption scheme. The first level of decryption results in the generation of metamorphic code envelopes as discussed in the previous section. The metamorphic code fragments themselves employ a second level of decryption for their code and data. In certain cases the second level of decryption is performed using a sublte technique involving the single-step handler.

Consider line 12 of the code fragment shown in Figure 8a. This is responsible for the decryption of the code for the particular metamorphic code fragment. Now consider the code fragment shown in Figure 8c which is obtained when the call is executed by Cobra. As seen from lines 1–3, Figure 8c, the trojan uses the PUSHF instuction and sets the trap-flag thereby invoking the single-step handler from that point on in execution. The single-step handler is present in the kernel-mode component, concealed within a polymorphic code envelope, and is responsible for the actual second level decryption.

We proceed to analyze the single-step handler using Cobra, by setting an overlay point on the IDT entry for the single-step exception upon encountering block execution events containing the instructions shown in Figure 8c. We also setup Cobra for generating memory read and write events on the instructions shown in lines 4–7, Figure 8c and the encrypted data section (line 13 onwards, Figure 8a) of the metamorphic code fragment, since we suspected that that single-step handler might overwrite those with new instructions on the fly.

Figure 9 shows a part of the single-step handler that was recon-

structed from the blocks by handling memory read and write events on the metamorphic code fragment in Figure 8a and the range of instructions shown in lines 4–7 of Figure 8c.

```
    ...
01. mov ebp, [edi+04h]
02. mov eax, [ebp]
03. mov [edi+18h], eax      -> instructions following
04. mov eax, [ebp+4]           single-step exception
05. mov [edi+1Ch], eax         form the decryption key
    ...
06. mov eax, [ebp-24h]
07. mov ebx, eax
08. shr ebx, 5
09. shl eax, 4
10. xor eax, ebx            -> Variation of the TEA
11. add eax, [ebp-24h]         employed for second
12. mov edx, [edi+1Ch]         layer decryptinon
13. and edx, 03h
14. mov ebx, [edi+18h]
15. add ebx, edx
16. mov edx, [ebx]]
17. add edx, [edi+1Ch]
    ...
```

Figure 9. W32/Ratos Single-Step Handler

As seen from lines 1–5, Figure 9 (documented via a memory read event on the instructions shown in lines 4–7, Figure 8c), the single-step handler makes use of a 64-bit value following the single-step exception from the instruction sequence shown in Figure 8c. A memory write event on the encrypted data section of the metamorphic code fragment in Figure 8a, later reveals code fragments of the single-step handler (lines 6–17, Figure 9) which resembles a variant of the TEA encryption algorithm using a 64-bit key which is borrowed from the two instructions following the single-step exception (lines 5–6, Figure 8c). Thus, the encryption key is stored within the metamorphic code fragment itself, albeit camouflaged in instructions, which is used to decrypt the obfuscated section for that particular metamorphic code fragment. Similar investigations reveals the IDEA encryption algorithm being employed as a first layer of encryption used to generate a metamorphic code fragment.

The W32/Ratos trojan employs a couple of other techniques apart from its metamorphic code envelopes, to prevent its analysis. These techniques are scattered in various areas of its internal code both in user- and kernel-mode. Consider the code fragment shown in Figure 10a. The W32/Ratos employs a technique which we call code execution timing. The trojan makes use of the RDTSC instruction to obtain the value of the processor time stamp counter at various intervals within a block of instructions comprising its code (lines 6 and 12, Figure 10a). If the difference in the execution time of the instructions for a block is greater than a particular pre-calibrated threshold for that block of instructions (lines 13–17, Figure 10a), the W32/Ratos sets certain specific flags which cause the decryption engine to cease working in a normal fashion (line 18, Figure 10a). Thus, when the trojan is executed in a virtual or supervised environment (resulting in an increase in the execution time beyond the the normal range) the metamorphic code generated by the trojan is erroneous and thus results in spurious faults.

Another technique employed by the W32/Ratos is what we call privilege-level checks. The trojan obtains the privilege level of its currently executing code, data and/or stack segment during execution of instructions in kernel-mode. This is done using varions instructions that access segment selectors (PUSH, MOV etc.). Figure 10b shows a fragment of code employing the PUSH instruction to obtain the code segment privilege level within its decryption engine. The W32/Ratos employs a very intelligent mechanism in that it uses the privilege level indicator in the segment selectors as a part of its internal computations (lines 3–9, Figure 10b). Thus, if

```
    ...
01. mov esi, [edx]
02. lea esi, [esi+22h]
03. mov esi, [esi+40h]
04. mov edx, [esi+44h]
05. xor ebx, ebx
06. rdtsc
07. push eax
08. call d [edi+32h]
09. jne 2150D010h
10. call d [edi+44h]
    ...
    jne 03
    call d [edi+12c]
    ...
```

```
    ...
11. pop [edi+4ch]
12. rdtsc
13. sub [edi+4ch], eax
14. lea ebx, [edi+4dh]
15. mov ebx, [ebx+24h]
16. sub [edi+4ch], ebx
17. jc 19
18. call d [edi+2dh]
19. mov eax, [esi]
    ...
```

(a)

```
...
01. mov eax, [ebp-20h]02.
02. mov ebx, eax
03. push cs
04. pop ecx
05. and ecx, 0Fh
06. add ecx, 5
07. shr ebx, ecx
08. dec ecx
09. shl eax, ecx
10. xor eax, ebx
...
```

(b)

Figure 10. W32/Ratos Anti-Analysis Tricks: (a) Code Execution Timing and (b) Privilege Level Detection

the code is run under a different privilege level to capture the behavior of the trojan in kernel-mode for example (this might be the case when running under a VM such as VMWare [52], Bochs [30] etc.), the decryption algorithm in this particular case would be erroneous as the values for the instructions would be completely different. However, Cobra handles such anti-analysis techniques with ease. The framework employs stealth-implants on blocks containing such instructions thereby fooling the malware into thinking its being executed without any form of supervision.

As seen, features provided by current fine-grained code-analysis frameworks do not suffice in the analysis of the W32/Ratos and other similar malware. However, with Cobra this task is greatly simplified. The framework cannot be detected or countered and allows fine-grained analysis to be dynamically and selectively deployed on desired code-streams. With Cobra it is relatively easy to glean malware details such as the malware code envelopes, its encryption/decryption engine, memory layout etc. – important pieces of information that facilitate the development of an antidote to combat the malware and it variants.

6. Performance Measurements

The performance of a fine-grained malware analysis framework such as Cobra depends on a number of factors, chief among them being: (a) the nature of the code being analyzed such as self-modifying, obfuscated, self-checking etc. and, (b) the style of analysis employed by an individual such as selecting the code streams to be analyzed, the analysis ranges, the blocks to be coalesced, the code blocks to be skipped etc. These factors are not easy to characterize and hence it is difficult to come up with a representative analysis session for performance measurements. This is further complicated by the fact that the same individual can adopt a different style of analysis at different times for a given code stream. Therefore, we will concentrate on presenting the performance of Cobra based on analysis sessions with a Windows based trojan, W32/Ratos (see

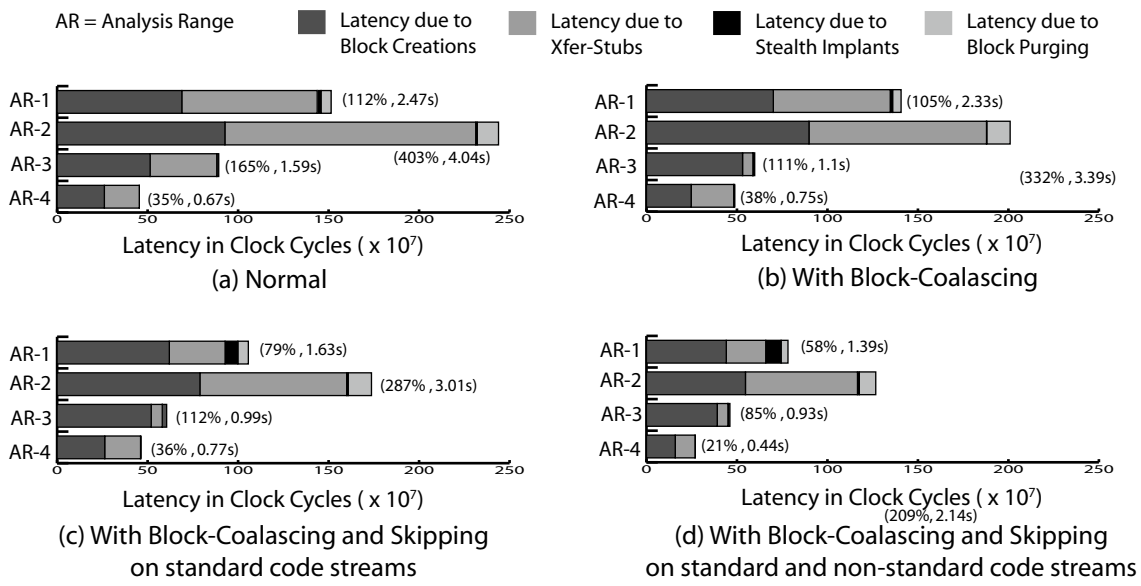

Figure 11. (a)-(d) Performance of Cobra on Analysis Sessions with the W32/Ratos

Section 5). The performance of the framework for other analysis sessions can be estimated in a similar fashion.

Before we proceed to present the performance measurements of Cobra, a few words regarding the test-bench are in order. To validate Cobra, we make use of our prototype malware analysis environment, WiLDCAT. The current version of WiLDCAT employs Cobra for its functioning and runs under the Windows OS (9x and XP) on the IA-32 (and compatible) processors. For test purposes, an Intel 1.7 GHz processor with 512 MB of memory was used.

We divide the total run-time latency of Cobra into: (a) latency due to block creations, (b) latency due to xfer-stubs, (c) latency due to block purging, and (d) latency due to stealth implants. Readings were taken at various points within WiLDCAT and Cobra to measure these overheads. We use processor clock cycles as the performance metric for the runtime latency. This metric is chosen, as it does not vary across processor speeds and also since it is a standard in literature related to micro benchmarks. The processor performance counter registers were used to measure the clock cycles by using the RDMSR instruction.

Figure 11 shows the performance of Cobra under different analysis methods for various analysis ranges. The analysis ranges were chosen from our analysis sessions involving the encryption and decryption engine of the W32/Ratos. The choice of the analysis ranges (single-step handler, parts of first and second decryption layers) were such that their semantics are relatively constant and they occur for every instance of the trojan deployment. This allows us to obtain a deterministic performance measure of various aspects of the framework. For the graphs in Figure 11, the y-axis (category axis) represents the analysis ranges the x-axis is the amount of extra clock cycles that are incurred as opposed to the native run-time of that particular range. Also, the data label next to each of the stacked bar in all the graphs represent the percentage of normalized latency and its corresponding time in seconds for a 1.7GHz Intel processor.

Figure 11a shows the performance of Cobra when run normally (without applying performance enhancement techniques such as block-coalescing and/or skipping). As seen from the graph of Figure 11a, latency due to block-creations and xfer-stubs are present in every analysis range (analysis ranges 1–4 in this case) and form a major portion of Cobra's overall latency since these elements form the backbone of the framework. Latency due to block-purging only comes into effect when an analysis range involves self-modifying code (analysis ranges 1–3 in this case) and is due to the fact that the framework invalidates the blocks corresponding to the modified code regions. Latency due to stealth-implants occur when Cobra needs to patch a block in order to prevent its detection. This is shown in analysis ranges 1, 2 and 4 which contain W32/Ratos anti-analysis code fragments. In general Cobra incurs lower overall latency when the ratio of straight line instructions to branches and loops is greater over a localized code region as exemplified by analysis range 4. In other cases the overall latency of the framework depends upon the number and nature of the branches encountered in the code stream. As an example, analysis range 2 incurs a latency as high as 5 times the normal execution time due to a high amount of code obfuscation via jumps. This is due to the increased block creations and xfer-stub overheads for such blocks, to ensure that the obfuscation is tackled completely during block execution (see Section 4.1.2).

Figure 11b shows the latency of Cobra employing block-coalescing on the same analysis ranges. Block-coalescing helps in reducing the latency due to xfer-stubs when analyzing code involving loops over instruction blocks. As seen from the graph in Figure 11b, analysis ranges 2 and 3 which contain a large number of loops incurs a much lower overall latency with block-coalescing when compared to its overall latency without block-colascing (Figure 11a). However, for analysis range 1 there is negligible gain in performance with block-coalescing, since the number of code loops is very less in its case. A point to note is that the latency of analysis range 4 with block-coalescing is more than its latency without block-coalescing in Figure 11a. This is due to the fact that the W32/Ratos generates varying amount of code for a given functionality and embeds a random amount of anti-analysis code fragments in different instances of its deployment due to its metamorphic nature.

block-skipping helps to further reduce the overall latency by excluding standard and/or already analyzed code streams from the analysis process. Figure 11c shows the performance of Cobra with

block-coalescing and block-skipping applied to standard kernel code streams. As seen from the graph of Figure 11c, the latency of analysis ranges 1 and 2 are reduced further when compared to their latency without block-skipping in Figure 11b. This is because, the code streams in analysis ranges 1 and 2 invoke standard kernel functions such as VirtualProtect, KeSetEvent, KeRaiseIrql etc. which are excluded from the slicing process with block-skipping. However, analysis range 4 has negligible improvement since it does not involve any calls to standard code-streams. Figure 11d shows the performance of Cobra with block-coalescing and block-skipping on standard as well as already analyzed malware code streams. As an example the single-step handler always invokes the code block handling a variant of the TEA decryption algorithm several times within its code stream. This code block never changes in its semantics and can be thus be skipped after analyzing it once. As seen from the graph of Figure 11d, analysis ranges 1–4 have a reduced latency from their counterparts in Figure 11c. Note that analysis range 4, which showed a negligible change with block-skipping of standard code streams, shows a noticable latency reduction with block-skipping applied to already analyzed code blocks.

Thus we can conclude that the performance of the framework is highly dependent on the nature of code being analyzed and the style of analysis employed by an individual (in terms of selecting analysis ranges, colascing blocks, choosing code blocks to skip etc.). However, the measurements give a gross indication of the latency one might expect. As seen from Figure 11, even the worst case latency (without block-coalescing and block-skipping) of the framework is found to be within the limits to suit interactive analysis.

7. Conclusions

We have presented Cobra, a stealth, efficient, portable and easy-to-use dynamic fine-grained malicious code analysis framework that overcomes the shortcomings in current research involving fine-grained code-analysis in the context of malware. Cobra facilitates the construction of powerful fine-grained malware analysis tools which are required to combat malware that are increasingly becoming hardened to analysis. Cobra does not make any visible changes to the executing code and hence cannot be detected or countered. The framework can capture multithreaded , SM-SC and any form of code obfuscations in both user- and kernel-mode while incurring a performance latency that is suitable for interactive analysis. The framework supports selective isolation – a technique that enables fine-grained analysis of malware specific code-streams while co-existing with normal code-streams in real-time. Cobra currently runs on the Windows (9x, NT, 2K and XP) and Linux OSs on the IA-32 (and compatible) processors. Cobra's architecture has minmal OS dependency and abstracts platform specific details thus enabling the framework to be ported to other platforms. We show Cobra's easy-to-use APIs enable construction of powerful fine-grained malware analysis tools with ease and discuss one of our own tools that we have used for fine-grained analysis of various malware. In our belief, Cobra is the first of its kind in tailoring a fine-grained code-analysis strategy specifically targetted at analyzing malware code-streams. Future works include improving the performance of the framework and integrating Cobra into a full fledged malware analysis environment currently being developed by us.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, May 2002.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Symposium on Requirements Engineering for Information Security (SREIS'01)*, March 2001.

[4] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salios, N. Tawbi, R. Charpentier, and M. Patry. Detection of malicious code in cots software: A short survey. *First International Software Assurance Certification Conference (ISACC'99)*, March 1999.

[5] J. Bergeron, M. Debbabi, M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the IEEE 4th International Workshop on Enterprise Security (WETICE'99)*, June 1999.

[6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[7] V. Bontchev. Methodology of computer anti-virus research. *Ph.D. Thesis, Faculty of Informatics, University of Hamburg*, 1998.

[8] D. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. *Ph.D. Thesis. Massachusetts Institute of Technology.*, 2004.

[9] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, November 2002.

[10] B. Chess. Improving computer security using extending static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 160–173, May 2002.

[11] D. Chess and S. White. An undetectable computer virus. *Virus Bulletin Conference*, 2000.

[12] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security03)*, pages 169–186, Aug 2003.

[13] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA04)*, pages 34–44, July 2004.

[14] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantic aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[15] M. Ciubotariu. Netsky: a conflict starter? *Virus Bulletin*, pages 4–8, May 2004.

[16] F. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6:22–35, 1987.

[17] F. Cohen. Operating system protection through program evolution. February 1998. Available online at URL http://all.net/books/IP/evolve.html. Last Accessed: 01 November 2005.

[18] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, August 2002.

[19] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *Technical Report 148, Department of Computer Science, University of Auckland*, July 1997.

[20] Compuware Corporation. Debugging blue screens. *Technical Paper*, September 1999.

[21] M. Debbabi, M. Girard, L. Poulin, M. Salois, and N. Tawbi. Dynamic monitoring of malicious activity in software systems. *Symposium on Requirements Engineering for Information Security (SREIS'01)*, March 2001.

[22] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of 11th USENIX Security Symposium (Security'02)*, 2002.

[23] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[24] J. Gordon. Lessons from virus developers: The beagle worm history through april 24, 2004. *Security Focus*, May 2004. Available online at URL http://downloads.securityfocus.com/library/BeagleLessons.pdf. Last accessed 01 November 2005.

[25] Intel Corp. Ia-32 intel architecture software developers manual. vols 1–3. *Intel Developers Guide*, 2003.

[26] T. Jensen, D. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[27] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, August 2004.

[28] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC04)*, 2004.

[29] X. Lai and J. Massey. A proposal for a new block encryption standard. In *Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 389–404, 1991.

[30] K. Lawton. Bochs: The open source ia-32 emulation project. Available Online at URL http://bochs.sourceforge.net, Last Accessed: 04 November, 2005.

[31] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.

[32] R. Lo, K. Levitt, and R. Olsson. Mcf: A malicious code filter. *Computers and Society*, 14(6):541–566, 1995.

[33] M. Loukides and A. Oram. Getting to know gdb. *Linux Journal*, 1996.

[34] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwoo. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[35] LURHQ. Sobig.e - evolution of the worm. *Technical Report. LURHQ*, 2003. Available online at URL http://www.lurhq.com/sobig-e.html. Last accessed 01 November 2005.

[36] J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. *Compendium of Workshops and Tutorials held in conjunction with PACT02'*, 2002.

[37] McAfee. W32/hiv. *Virus Information Library*, October 2000. Available online at URL http://vil-origin.nai.com/vil/. Last accessed 28 Oct. 2005.

[38] McAfee. W32/mydoom@mm. *Virus Information Library*, 2004. Available online at URL http://vil-origin.nai.com/vil/. Last accessed 28 Oct. 2005.

[39] G. McGraw and G. Morrisett. Attacking malicious code: Report to the infosec research council. *IEEE Software*, 17(5):33–41, October 2000.

[40] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *3rd Workshop on Runtime Verification*, 2003.

[41] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals*, E86-A(1), 2003.

[42] V. Paxon. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[43] J. Robbins. Debugging windows based applications using windbg. *Miscrosoft Systems Journal*, 1999.

[44] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Reconfigurable and retargetable software dynamic translation. In *1st Conference on Code Generation and Optimization*, pages 36–47, 2003.

[45] P. Singh and A. Lakhotia. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices*, 37(2):29–35, February 2002.

[46] L. Spitzner. Honeypots: Tracking hackers. *Addison-Wesley*, 2003. ISBN: 0-321-10895-7.

[47] Symantec. Understanding and managing polymorphic viruses. Available online at URL http://www.symantec.com/avcenter/whitepapers.html. Last Accessed: 28 October 2005.

[48] P. Szor. The art of computer virus research and defense. *Addison Wesley in collaboration with Symantec Press*, 2005.

[49] TrendMicro. Bkdr.surila.g (w32/ratos). *Virus Encyclopedia*, August 2004. Available online at URL http://www.trendmicro.com/vinfo/virusencyclo/. Last accessed 28 Oct. 2005.

[50] A. Vasudevan and R. Yerraballi. Stealth breakpoints. *21st Annual Computer Security and Applications Conference (ACSAC'05)*, December 2005.

[51] A. Vasudevan and R. Yerraballi. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. *29th Australasian Conference in Computer Science (ACSC'06)*, January 2006.

[52] VMWare Inc. Accelerate software development, testing and deployment with the vmware virtualization platform. *Technical Report, VMWare Technology Network*, June 2005.

[53] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proceedings of International Conference of Dependable Systems and Networks*, 2001.

[54] D. Wheeler and R. Needham. Tea, a tiny encryption algorithm. In *Proceedings of the 2nd International Workshop on Fast Software Encryption*, pages 97–110, 1995.

[55] G. Wroblewski. General method of program code obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, June 2002.

[56] T. Yetiser. Polymorphic viruses, implementation, detection and protection. *VDS Advanced Research Group, P.O. Box 9393, Baltimore, MD 21228, USA*. Available online at URL http://vx.netlux.org/lib/ayt01.html. Last accessed 28 Oct. 2005.