Detecting Call Obfuscations in x86 Executables

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Michael P. Venable

Fall 2005

© Michael P. Venable

2005

All Rights Reserved

Detecting Call Obfuscations in x86 Executables

Michael P. Venable

APPROVED:

Arun Lakhotia, Chair Associate Professor of Computer Science	Vijay Raghavan Professor of Computer Science
William R. Edwards, Jr. Associate Professor of Computer Science	C. E. Palmer Dean of the Graduate School

TABLE OF CONTENTS

1 Introduction 1
  1.1 Motivation 1
  1.2 Research Objective 3
  1.3 Research Contribution 3
  1.4 Organization of Thesis 4

2 Related Work 5

3 Background 13
  3.1 Disassembly Overview 13
  3.2 Obfuscation Overview 16

4 Definitions 21
  4.1 Reduced Interval Congruence 21
  4.2 Stack-Location 22
  4.3 Value 22
  4.4 State 22

5 Operations 24
  5.1 Arithmetic Operations 24
  5.2 Memory Operations 29
  5.3 Miscellaneous Operations 30

6 Evaluation Function 32

7 Analysis 35

8 Examples 37
  8.1 Using Push/Jmp 37
    8.1.1 Interpretation 38
    8.1.2 Analysis 40
  8.2 Using Push/Ret 41
    8.2.1 Interpretation 41
    8.2.2 Analysis 42
  8.3 Using Pop to Return 43
    8.3.1 Interpretation 43
    8.3.2 Analysis 44
  8.4 Modifying Return Address 44
    8.4.1 Interpretation 44
    8.4.2 Analysis 44

9 Prototype 46
9.1 Phases 46
9.1.1 Disassembly 46
9.1.2 Parsing 48
9.1.3 Interpretation 48
9.1.4 Analysis 50
9.2 Implementation Results 51
10 Results 53
11 Conclusion 57
Bibliography 58
Abstract 60
Biographical Sketch 61

List of Figures

Figure 3-1. Example obfuscation of a call instruction 19

Figure 5-1. Possible abstract stack (a) before adding to esp (b) after adding to esp 24

Figure 5-2. Possible abstract stack (a) before subtracting from esp (b) after subtracting from esp 26

Figure 7-1. ASG annotated with address of instruction that created it 35

Figure 8-1. Obfuscated call using push/jmp 37

Figure 8-2. Contents of the state at various points in the example program .... .40

Figure 8-3. Call obfuscation using push/ret 42

Figure 8-4. Obfuscation using pop to return 43

Figure 8-5. Obfuscation by modifying return address 45

Figure 9-1. The four phases of the detection process. 46

Figure 9-2. Control-flow graph illustration of (a) a simple junction node and (b) a loop junction node 50

Figure 9-3. Prototype user interface 52

Figure 10-1. Manipulation of the abstract stack graph 55

List of Abbreviations

ASG	abstract stack graph
AST	abstract syntax tree
CFG	control flow graph
DLL	dynamic link library
ELF	executable and linking format
PE	portable executable
RIC	reduced interval congruence
SEH	structured exception handling
VSA	value set analysis

1 Introduction

1.1 Motivation

Programmers obfuscate their code with the intent of making it difficult to discern information from the code. Programs may be obfuscated to protect intellectual property and to increase security of code (by making it difficult for others to identify vulnerabilities) [1-3]. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners [4]. The focus of this thesis is detecting obfuscated malicious code.

Malicious code writers have many obfuscating tools at their disposal, such as Mistfall and CB Mutate (provided by the BlackHat community) as well as commercially available tools such as Cloakware and PECompact. They may also develop their own tools. Some known obfuscation techniques include variable renaming, code encapsulation, code reordering, garbage insertion, and instruction substitution [1]. We are interested in instruction substitution performed at the assembly level, particularly for *call* obfuscations.

A common obfuscation observed in malicious programs is obfuscation of *call* instructions (i.e., call obfuscation) [4]. For instance, the *call addr* instruction may be replaced with two *push* instructions and a *ret* instruction, the first *push* pushing the address of the instruction after the *ret* instruction (the return address of the procedure call), the second *push* pushing the address *addr* (the target of the procedure call). The third instruction, *ret*, causes execution to jump to *addr*, simulating a *call* instruction. No actual *call* statement is present in the code. The code may be further obfuscated by spreading the instructions and by further splitting each instruction into multiple instructions.

Some analysis methods draw conclusions about a program's intent by observing the system calls made by the program [5]. If the pattern of system calls matches a known malicious pattern of calls, then the file is deemed malicious. Symantec's Bloodhound technology, for example, uses classification algorithms to compare the system calls made by the program under inspection against a database of calls made by known viruses and clean programs [6].

The challenge, however, is in detecting the operating system calls made by a program. The Portable Executable (PE) and Executable and Executable and Linking Format (ELF) file formats for binaries include a mechanism for informing the linker about the libraries used by a program, but there is no requirement that this information be present. For instance, in Windows, the entry point address of various system functions may be computed at runtime via the Kernel32 function GetProcAddress. The Win32.Evol worm uses precisely this method for obtaining the addresses of kernel functions and also uses call obfuscation to further deter reverse engineering.

Obfuscation of call instructions breaks most virus detection methods based on static analysis since these methods depend on recognizing call instructions to (a) identify the kernel functions used by the program and (b) to identify procedures in the code. This obfuscation also takes away important cues that are used during manual analysis. We are then left with only dynamic analysis, i.e., running a suspect program in an emulator and observing the kernel calls that are made. Such analysis can easily be thwarted by what is termed as a "picky" virus—one that does not always exhibit malicious behavior. In addition, dynamic analyzers must use some heuristic to determine when to stop analyzing a program, for it is possible the virus may not terminate without user input. Virus writers can bypass these

heuristics by introducing a delay loop that simply wastes cycles. It is therefore important to detect obfuscated calls for both static and dynamic analysis of viruses.

To address this situation, this thesis incorporates the work of Balakrishnan and Reps [2] with the work presented by Lakhotia and Kumar [4]. In particular, the notion of Reduced Interval Congruence (RIC) will be employed to approximate the values that a register may hold. However, unlike the techniques of Balakrishnan and Reps [2], register *esp* will hold a value that specifically represents some node or group of nodes in an abstract stack graph. Since graph nodes are not suitable to be represented by RIC, we maintain both the stack location and RIC information when performing our analysis.

1.2 Research Objective

The goal of this research is to design an algorithm for detecting call obfuscations embedded within x86 executable images and to design a tool that applies the algorithm to actual executables. The hope is that such a tool may find a purpose in assisting malware analyzers, either manual or automated, by helping to quickly identify potentially malicious programs and to provide more reliable analysis results.

1.3 Research Contribution

This research combines work from various authors to form a new, unified model for the detection of obfuscated calls. Two previous pieces—value-set analysis (VSA) and abstract stack graph (ASG)—are used as the foundation for this work. VSA is a relatively new approach towards abstract interpretation of program values while the goal of recent ASG work is the detection of obfuscated calls. The ASG work, however, suffers from a critical

weakness—it does not know the values of program variables. This limitation means several x86 instructions, such as an indirect jump, cannot be properly handled. Many other examples of attacks against the ASG approach can be derived. To counter such attacks, a method of determining program values is needed, prompting the merger of the VSA and ASG methods.

The unique approach to static analysis presented in this thesis uses the algorithm given in the ASG work to detect static calls and joins this method with the VSA work for determining program values. A formal definition of this newly created domain is presented along with a formal specification of the analysis process. This new approach allows for the proper handling of a much larger set of instructions, and the hope is that it will result in a more accurate and robust analysis process.

1.4 Organization of Thesis

This thesis is organized as follows. Chapter 2 discusses work related to the area of static analysis. Chapter 3 discusses background material that may be helpful in understanding this work. Chapters 4, 5, and 6 provide formal specifications of the analysis domain, while Chapter 7 describes how the elements in the domain are utilized to uncover call obfuscations. Chapter 8 contains examples illustrating the various phases in the analysis process, and a prototype resulting from this work is presented in Chapter 9. Chapter 10 discusses the results and insights gained by this work, and Chapter 11 concludes this thesis.

2 Related Work

Fred Cohen, in his exemplary work “Computer Viruses—Theory and Experiments,” defines a computer virus as “a program that can ‘infect’ other programs by modifying them to include a possibly evolved copy of itself” [7]. It is possible for each infected program to infect other programs, resulting in increases in infection rates. The main contribution of Cohen’s work is his argument that there is no algorithm that can detect all possible viruses. To prove this, he creates a program containing a function D that returns true if the program is a virus and returns false if it is not. Take, for instance, the following pseudo-code:

```
Program P := if D(P) then exit; else spread.
```

The program does the following: if D determines that P is indeed a virus, then P will simply exit and not exhibit any virus-like behavior; if D determines that P is not a virus, then P will infect other files, a prime characteristic of a virus. Thus, if one assumes D can correctly identify a virus, a contradiction arises. This simple program shows that generic detection of viruses is undecidable, thanks to the contradiction.

Cohen’s work exposes nine problems that can be proven to be undecidable: (1) detection of a virus by its appearance; (2) detection of a virus by its behavior; (3) detection of an evolution of a known virus; (4) detection of a triggering mechanism by its appearance; (5) detection of a triggering mechanism by its behavior; (6) detection of an evolution of a known triggering mechanism; (8) detection of a viral detector by its behavior; (9) detection of an evolution of a known vial detector.

Cohen notes that, while he has shown that, in general, detection of viruses is undecidable, it is still possible to detect any particular virus. This is pleasant news in comparison to his previous discoveries, yet researchers Chess and White [8] have shown that even this statement may be too optimistic.

Chess and White show that there are computer viruses that no algorithm can detect, at least not perfectly. For this proof, the following virus is constructed:

```
Program P :=
  if S(P) then
    exit
  else
    replace the text of subroutine S with a random program;
    spread;
    exit;
  S := return false;
```

From the code above, one can see that this virus is capable of mutating its own code. Let's assume there is an algorithm C that can detect this virus. For all possible variations of P, there will be at least one containing the code:

```
Program P :=
  if S(P) then
    exit
  else
    replace the text of subroutine S with a random program;
    spread;
    exit;
  S := return C(P)
```

Like the program presented by Cohen, this program is contradictory, because it uses the detection algorithm C to decide whether to spread or not. This approach can be extended to show that for any detection algorithm, there is some mutation of P that cannot be detected.

In effect, Cohen's work shows that

$\forall$ A, $\exists$ V s.t. A does not detect V (for every algorithm, there is some virus that it does not detect) [8],

while Chess's and White's work shows

$\exists$ V s.t. $\forall$ A, A does not detect V (there exists a virus which no algorithm perfectly detects) [8].

Thus, when trying to detect viruses, we are forced to either (a) settle for imperfect results or (b) design the detector for a particular virus (which, according to Chess's and White's findings, may also be imperfect). In order for Chess's and White's findings to be successful, the virus must be able to mutate its code. While there are no viruses (aware to the author) that successfully mutate code to alter their behavior (not intentionally anyway), there are many that alter their appearance through code mutation. These are commonly referred to as metamorphic viruses.

Metamorphic viruses alter their instructions before spreading to a new host [9-11]. The altered instructions are structurally different, but semantically equivalent. Traditional methods of virus detection involve obtaining a signature, typically a sequence of bytes, that is specific to a particular virus and can be used to identify the virus. Since the body of a metamorphic virus changes with each new infection, it is no longer possible to use a static signature as a means of identification.

Some earlier, and simpler, metamorphic viruses contained features such as swapping register names in various locations of the code. Code such as `MOV eax, 5` can be replaced with `MOV`

`ecx, 5`, assuming that the value in ecx is not needed at that point. More complicated viruses may swap whole instructions with equivalent, but different, instructions.

Work by Christodorescu and Jha [12] tries to defeat metamorphic viruses by recognizing patterns in the code. They have found that even simple changes to code, such as inserting *nop* instructions, are enough to avoid detection from many commercial anti-virus products. To prevent becoming prey to such tactics, the authors present a language for specifying malicious patterns in code in a way that is abstract enough to be resistant to code changes, but specific enough to identify the virus effectively. For instance, instead of specifying specific register names, one can use X to represent any registers. Thus, the instruction `pop X` may represent `pop eax`, `pop ebx`, etc. Additionally, a set of instructions can be specified along with a set of conditions that must hold, such as "this instruction uses X" or "this instruction does not kill X." This form is abstract enough so that new instructions can be inserted between the original instructions and still be detected. It cannot, however, handle code changes that replace whole instructions with different, but equivalent, instructions.

Newer work by the same team aims to improve on the previous work and remove its shortcomings [13]. To accomplish this, instead of specifying actual instructions, they specify the intent or meaning of the instruction. Arithmetic operations such as "A = A + c" match the instruction `eax = eax + 4`; a statement like "mem[B] = f(mem[A])" can match the instruction `mem[ebx] = mem[edx-3]`; the statement "B = const_addr" may match `eax = 0x403000`. These are just a few possibilities. By looking only at the meaning of code, it is possible to detect more complicated code changes than previous approaches. Code such as `pop eax` is semantically equivalent to the code sequence `mov eax, [esp]; add esp, 4`;

analysis that looks only at code structure would not detect equivalence and may result in false negatives. Looking at semantics may be the answer.

Lakhotia and his students have also been working at defeating metamorphic viruses [9, 10, 14]. Their approach involves reversing the code transformations performed by the virus with the hope of reaching a stable, common form that can be used as a basis for comparison to other files infected with a mutation of the same virus. To do this, Lakhotia recovers a transformation function *f* from the virus. Given a metamorphic virus A, new mutated generations are generated (by the virus) by applying the transformation function, resulting in mutations: $f(A)=B$, $f(A)=C$, $f(A)=D$, etc. Lakhotia proposes applying the reverse of the transforms to each mutation, resulting in $f^{-1}(A)=A'$, $f^{-1}(B)=A'$, $f^{-1}(C)=A'$, etc. The A' in the example is called the zero-form, and the goal is to construct the inverse function in such a way that, when applied to a mutation of a virus, it always results in the same zero-form. Once done, one can create a static signature on the zero-form (similar to how signatures have traditionally been created) and new mutations can be detected by first applying the inverse function and then checking for the signature. The difficult part is ensuring the inverse function terminates and always generates the same zero-form when given a new mutated variant of a virus.

In addition to mutating code, many virus writers turn to code obfuscation as a means to defeat analysis, and, as one would expect, much research has been done on the effects obfuscation has on analysis. Linn and Debray describe several code obfuscations that can be used to thwart static analysis [3]. Specifically, they attack disassemblers by inserting junk statements at locations where the disassembler is likely to expect code. A junk statement is a

statement that never gets executed and is not necessary for the code to function properly (similar to dead code). It may contain real instructions or may simply be bytes containing random data. Of course, in order to maintain the integrity of the program, these junks bytes must not be reachable at runtime.

Linn and Debray take advantage of the fact that most disassemblers are designed around the assumption that the program under analysis will behave "reasonably" when function calls and conditional jumps are encountered. In the normal situation, it is safe to assume that, after encountering a *call* instruction, execution will eventually return to the instruction directly following the call. However, it is easy for an attacker to construct a program that does not follow this assumption, and by inserting junk bytes after the call, many disassemblers will incorrectly process the junk bytes as if they were actual code. Another obfuscation technique involves using indirect jumps to prevent the disassembler from recovering the correct destination of a jump or call, thereby resulting in code that is not disassembled.

The authors show that, by using a combination of obfuscation techniques, they are able to cause, on average, 65% of instructions to be incorrectly disasassembled when using the popular disassembler IDA Pro from DataRescue [15]. To counter these obfuscations, it would be necessary to (1) determine the values of indirect jump targets and (2) correctly handle call obfuscations. Doing so will help avoid the junk bytes that confound many disassemblers.

Balakrishnan and Reps [2] show how it is possible to approximate the values of arbitrary memory locations in an x86 executable. Their paper introduces the Reduced Interval

Congruence (RIC), a data structure for managing intervals while maintaining information about stride. Previous work in this area discuss how intervals can be used to statically determine values of variables in a program [16], but the addition of stride information makes it possible to determine when memory accesses cross variable boundaries, thus increasing the usefulness of such an approach.

Their paper, however, assumes that the executable under analysis conforms to some standard compilation model and that a control-flow graph can be constructed for the executable. Incorrect results may arise when applied to an executable consisting of obfuscations typically found in malicious programs [2, 17].

Kumar and Lakhotia [4] present a method of finding call obfuscations within a binary executable. To accomplish this, they introduce the abstract stack graph, a data structure for monitoring stack activity and detecting obfuscated calls statically. The abstract stack associates each element in the stack with the instruction that pushes the element. An abstract stack graph is a concise representation of all abstract stacks at every point in the program. If a return statement is encountered where the address at the top of the stack (the return address) was not pushed by a corresponding call statement, it is considered an obfuscation attempt and the file is flagged as possibly malicious.

The limitation of this approach is that the stack pointer and stack contents may be manipulated directly without using push and pop statements. Doing so bypasses the mechanisms used in Kumar and Lakhotia's approach for detecting stack manipulation and may result in an incorrect analysis. Also, indirect jumps cannot be properly analyzed, since

there is no mechanism for determining jump targets of indirect jumps. These limitations may be overcome by combining their stack model with the work by Balakrishnan and Reps for analyzing the content of memory locations.

3 Background

3.1 Disassembly Overview

At the heart of static analysis for binaries lies disassembly. Disassembly is the process of reverse-engineering an executable to recover the set of assembly instructions stored within the executable image. Many disassemblers also perform additional functions, such as reconstructing the control-flow graph from the disassembled output. Many static analysis tools use this information in conjunction with a set of analysis rules to identify a file as either benign or malicious. Disassembly, however, is not an exact science, and there are many different approaches, each with its own set of trade-offs.

A common and simple disassembly approach, known as linear sweep, requires the disassembler to start processing from the beginning (of the .text segment) and disassemble one instruction at a time until the end is reached [18, 19]. This simplistic method fails when data is embedded within the code, as shown below.

```
Main:
  …
  JMP FUNC
  DB  0E8h
FUNC:
  …
```

A linear sweep disassembler would simply disassemble the data as if it were code. If the data happens to match a real assembly instruction, then the disassembler would continue without a clue about the error. Otherwise, the disassembler may skip over the byte that is known not to be code and try the next byte. An interesting artifact from this is that the disassembler tends to eventually get "back on track." That is, it will begin disassembling actual instructions

correctly at some later point in time [3]. In the meantime, many instructions will be incorrectly disassembled, and the disassembler will not backtrack to correct the problem.

Another popular disassembly method is known as recursive traversal [3, 18, 19]. This method tries to overcome the weaknesses of linear sweep by following the control-flow of the program when disassembling, instead of disassembling in a straight line from start to finish. Thus, if the programmer inserts data in the middle of the instruction stream (as in the above example), the disassembler will jump over the data and will not be fooled. There are other situations that do cause problems, however. In order to know where to resume disassembly after a jump, the disassembler must be able to deduce the possible jump targets statically, which is not always an easy task. Indirect jumps, in particular, pose a challenge, and if the target of an indirect jump cannot be determined, there is the possibility of code not being disassembled (that is, the disassembler believes the code is data).

Another tricky situation arises when one of the jump targets is data. Take, for instance, the code below:

```
Main:
  CMP   eax, eax
  JE    FUNC+1
  JMP   FUNC
  …
FUNC:
  DB    0E8h
  PUSH  0
  CALL  ExitProcess
```

Mentally tracing through the execution of the code reveals that the instruction `JE FUNC+1` (jump if equal) always transfers control to *FUNC+1*, because the two operands being compared (*eax* and *eax*) are always equal. Thus, the instruction `JMP FUNC` is not reachable

and will never be executed. This is a good thing too, since there is no code at *FUNC*, only data. (This style of trickery is known as an opaque predicate. See Chapter 3.2 for information on opaque predicates.)

As a test, we fed this program into the debugger OllyDbg. OllyDbg assumed that *FUNC* and *FUNC+1* were both valid jump targets and thus must both contain code. After disassembly, Ollydbg produced the following output (modified slightly for readability):

```
Main:
  CMP  eax,eax
  JE   FUNC+1
  JMP  FUNC
FUNC:
  CALL 01281075
  ADD  byte ptr ds:[eax],al
  ADD  bh,bh
  AND  eax,403030
```

This code is very different from the original executable's code. Ollydbg incorrectly interpreted the data at location *FUNC* as code, resulting in a sequence of incorrectly parsed instructions. One might expect the disassembler to recognize that it is processing data instead of code and identify it as such. However, in this case, the data at *FUNC* consist of the byte E8h, which happens to match the opcode for the *call* instruction. It is therefore not easily possible for the disassembler to recognize that it is processing data (to OllyDbg's credit, it did place question marks next to the incorrectly disassembled instructions). For this obfuscation to work, it is important that the byte located at *FUNC* is a valid opcode. Replacing 0E8h with an invalid opcode, say 0h, will allow OllyDbg to correctly disassemble the executable.

Other disassembly methods have been introduced and some variations of the previous techniques have been proposed. Worth noting is interactive disassembly, which requires a

human to make key decisions to improve the disassembly [19]. The disassembler may perform an initial disassembly, and the user may override some of the decisions made by the disassembler or may instruct the disassembler to disassemble areas of code that the disassembler misinterpreted for data. Also possible is running the suspect program in a debugger while allowing the user to guide the disassembly process. With this sort of dynamic analysis, a more precise disassembly can be achieved, but at the expense of more human intervention.

Understanding the possible pitfalls of disassembly is important, since, like most other forms of static analysis, this project relies on the disassembly to perform its work. Any disassembly method we choose will have an impact on the results obtained. The specifications in this thesis can, fortunately, be used with any disassembler, thus it is left up to the implementer to decide which tradeoffs are acceptable and choose an appropriate disassembly method.

3.2 Obfuscation Overview

Code obfuscation is a technique for altering the structure of a program's instruction set in order to make the meaning less apparent and thus harder for someone to reverse engineer. Obfuscation has found a purpose in many legitimate applications where it is sometimes used to deter reverse-engineering by competitors who may be interested in learning proprietary formats [1, 3]. Simple encryption of the code is not enough, since the code must be decrypted before it can be executed and would then become vulnerable to reverse

engineering. Thus, obfuscation has proven to be a useful technique to hide important information within code.

On the other hand, like any technology, obfuscation can be used in a more malicious context. Specifically, malicious code writers frequently turn to code obfuscation to prevent analysis by researchers. Code obfuscation may be particularly effective at deterring static analysis, since dynamic analysis analyzes behavior and behavior is typically left unchanged by obfuscators. Taken from [20], the static analysis process can generally be broken down into five phases: disassembly, procedure abstraction, control-flow graph generation, dataflow analysis, and property verification. It is possible (and can perhaps be an insightful exercise for the malware analyst) to use obfuscation to attack any of the five phases. Since understanding these obfuscations may help us see where this work is situated in the big picture, it may be worthwhile to review various forms of code obfuscation and see how they can be applied to attack static analysis.

Many obfuscation techniques use what is referred to as opaque predicates. An opaque predicate is a program variable whose value is known before run-time and is used in such a way that the obfuscator can predict the flow of control, but an analyst may not (or cannot do so easily) [1]. The obfuscator uses the opaque predicate as an argument to a conditional branch and is then able to decide which path is chosen based on whether the predicate is true of false. A good opaque predicate is one that contains a value that is computationally expensive to determine. If an analyst cannot determine the value of the predicate in a conditional, then the only option is to assume both paths are possible. Among other things, opaque predicates provide a way to attack the control-flow graph generation phase of static

analysis. By using these contrived branch statements, one can force an analyzer to add unnecessary edges to the control-flow graph, thus degrading the results.

Another useful tactic is to insert irrelevant code or data, frequently with the help of opaque predicates. Take the following piece of code as an example:

```
if x < 0 then
  y = true
else
  y = false
```

It is obvious that, after execution, *y* may have two possible values, either *true* or *false*. If the variable *x*, however, is an opaque predicate (it's value is known before run-time), then clearly *y* may have only one value after execution—*true* if x is less than zero, *false* otherwise. Such an attack is aimed at the dataflow analysis phase and can decrease the precision of the analysis results.

A variation of the above theme is to insert junk data, an example of which is shown in Chapter 3.1. In this attack, the goal is to trick the disassembler into incorrectly disassembling data as if it were code. This attack is clearly aimed at the disassembly phase of static analysis.

Another interesting obfuscation is to break the common assumptions surrounding branch instructions. In most compiler generated code, a *call* instruction is paired with a *ret* instruction, and the *ret* instruction transfers control to the instruction following the *call*, but one can choose not to follow this convention. Doing so obscures the flow of execution in a program and is thus an attack on the control-flow generation phase. It can also, however,

cause problems during the disassembly phase for certain methods of disassembly (recursive-traversal for instance).

Violating the common branching conventions can be also be used to remove information about library calls, the focal point of this project. One can effectively hide library calls by replacing the *call* statement with a different, but equivalent, instruction or set of instructions. Figure 3-1 presents three different ways to call the function *DeleteFileA*. An analyzer designed to search for *call* instructions would overlook the second two sets of instructions in the figure, which would have an effect on the property verification phase.

One last method worth discussing is to aggregate or otherwise do away with the abstractions put in place by the programmer. Programmers typically use abstractions to make the translation from design to code simpler, but these abstractions give vital clues about the code's purpose. In assembly, the key abstraction is the procedure. At procedure boundaries, then, is the logical place to obfuscate. By obfuscating *call* instructions (similar to the process of obfuscating calls to system functions), one can blur the boundaries between procedures.

```
PUSH  eax                    ; push function parameter
CALL  DeleteFileA            ; transfer control to function

PUSH  eax                    ; push function parameter
PUSH  $+10                   ; push return address
JMP   DeleteFileA            ; transfer control to function

PUSH  eax                    ; push function parameter
PUSH  $+11                   ; push return address
PUSH  offset DeleteFileA     ; push address of function
RET                          ; transfer control to function
```

Figure 3-1. Example obfuscation of a call instruction. Each set of instructions are equivalent in function (`$` is the current byte within the executable, thus `$+10` means 10 bytes after this position).

Also, code from various procedures can be interleaved resulting in code where multiple procedures are "woven" together visually, but are still two separate and independent procedures semantically (clearly an attack on the procedural abstraction phase). Indeed, one virus, Zmist, takes this idea to the extreme:

> The virus supports a unique new technique: code integration. The Mistfall engine contained in the virus is capable of decompiling Portable Executable files to its smallest elements, requiring 32MB! of memory. Zmist will insert itself into the code: it moves code blocks out of the way, inserts itself, regenerates code and data references, including relocation information, and rebuilds the executable. This is something that has not been seen in any previous virus [11].

This chapter presents only a portion of the possible attacks using obfuscation. The work in this project is primarily focused on call obfuscations, though some of the material might also apply to other forms of obfuscations. For instance, this thesis presents a method for abstract interpretation of program values that could be used to uncover opaque predicates. Once the opaque predicates are identified, irrelevant branches could be removed much like a compiler removes dead code. At present, we focus our attention only on call obfuscations simply to keep the breadth of the project at a reasonable level.

4 Definitions

We now begin a formal specification of this project's analysis method. The domain of our method consists of RICs (reduced interval congruence), stack-locations, values, and a state.

4.1 Reduced Interval Congruence

A Reduced Interval Congruence is a hybrid domain that merges the notion of interval with that of congruence. Since an interval captures the notion of upper and lower bound [16] and a congruence captures the notion of stride information, one can use RIC's to combine the best of both worlds. An RIC is a formal, well-defined, and well-structured way of representing a finite set of integers that are equally apart.

For example, say we need to over-approximate the set of integers {3,5,9}. An interval over-approximation of this set would be the interval [3,9] which contains the integers 3, 4, 5, 6, 7, 8, and 9; a congruence representation would note that 3, 5, and 9 are odd numbers and over-approximate {3,5,9} with the set of all odd numbers 1,3,5,7,…. Both of these approximations are probably much too conservative to achieve a tight approximation of such a small set. The set of odd numbers is infinite and the interval does not capture the stride information and hence loses some precision.

In the above example, the RIC 2[1,4] +1, which represents the set of integer values {3, 5, 7, 9} clearly is a tighter over-approximation of our set.

Formally written, an RIC is defined as:

$RIC := a \times [b,c]+d = \{x \mid x = aZ+d \text{ where } Z \in [b,c]\}$

4.2 Stack-Location

A stack-location is an abstract way of distinguishing some location on the stack. It is "abstract" in the sense that no attempt is made to determine the location's actual memory address. Instead, each stack-location is represented by a node in an abstract stack graph. Each stack-location stores a value, discussed next.

4.3 Value

Each stack-location and register stores a value. A value is an over approximation of the location's run-time content and may be a stack-location, RIC, or both. If an RIC or stack-location is $\top$, its value is either not defined or cannot be determined. Also, a stack-location may be given the value **B**, which represents the bottom of the stack.

More formally,

$\text{VALUE} := \text{RIC}_{\top} \times P(\text{STACK_LOCATION})_{\top}$

4.4 State

The state represents the overall configuration of the memory and registers at a given program point. The state consists of a mapping of registers to values, a mapping of stack-locations to values, and the set of edges in the stack graph.

Formally,

$$\begin{aligned}\text{STATE} := (&\text{REGISTER} \rightarrow \text{VALUE},\\ &\text{STACK_LOCATION} \rightarrow \text{VALUE},\\ &\text{STACK_LOCATION} \times \text{STACK_LOCATION})\end{aligned}$$

Some explanation about the notation surrounding the state may be in order before proceeding to the next chapters. Throughout the remainder of this text, we use the symbol "state↓X" to denote the Xth element of the state, thus "state↓2" denotes the 2nd element, which is the store of STACK_LOCATION-VALUE mappings. The symbol "a ↦ b" denotes assignment of the value b to a. Lastly, the symbol "[a]b" denotes appending a to b. As an example, the statement "[*eax* ↦ (⊤,⊤)]state↓1" maps *eax* to the value (⊤,⊤) and adds this new mapping to the store of register mappings. Also, the statement "a→(b ▯ c)" can be read as "if a then b else c."

5 Operations

5.1 Arithmetic Operations

Operations are defined that can be applied to values. The result of each operation depends on whether the value represents a stack-location or RIC. For instance, adding two RICs results in a new RIC, where the new RIC is an over-approximation of the sum of the two RICs given as input. Addition of an RIC and a stack-location outputs a set of stack-locations. These stack-locations are obtained by traversing the abstract graph, starting from the stack-location given as input, and stopping after n nodes have been traversed, where n is a number included in the RIC given as input. This is equivalent to adding some number to a stack address and getting some other stack address as output (Figure 5-1). Adding two stack-locations is the same as adding two stack addresses, and since we make no attempt to determine the addresses of locations on the stack, we are unable to perform the addition. Thus, addition of two stack-locations results in an undefined value. The $\sqcup$ operator, seen in the definition of +, returns the union of two values.

add: VALUE $\times$ VALUE $\times$ STATE $\rightarrow$ VALUE

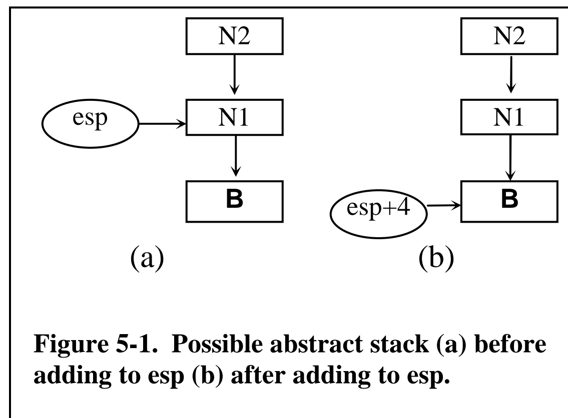

Figure 5-1. Possible abstract stack (a) before adding to esp (b) after adding to esp.

add: ((a, ⊤), (c, ⊤), s) → (+(a, c), ⊤)

add: ((a, ⊤), (⊤, d), s) → (⊤, +(a, d, s))

add: ((a, ⊤), (c, d), s) → (+(a,c), +(a, d, s))

add: ((⊤, b), (c, ⊤), s) → (⊤, +(c, b, s))

add: ((a, b), (c, ⊤), s) → (+(a,c), +(c, b, s))

add: anything else → (⊤,⊤)

$+: \text{RIC} \times \text{RIC} \rightarrow \text{RIC}$

$+(\text{R1}, \text{R2}) = \sqcup \text{R1} \boxplus a$, where $a \in \text{R2}$

$+: \text{RIC} \times \text{STACK_LOCATION} \times \text{STATE} \rightarrow P(\text{STACK_LOCATION})$

$+(\text{R}, s, \text{state}) = \sqcup r^{th}$ successor of s, where $r \in \text{R}$

The ⊞ operator shifts an RIC by a specified amount and, in effect, adds a number to an RIC.

$\boxplus: \text{RIC} \times \mathbb{N} \rightarrow \text{RIC}$

$(a[b,c]+d) \boxplus x = (a[b,c]+(d+x))$

Subtraction is similarly defined. Two RICs can be subtracted to produce another RIC. A stack-location minus an RIC results in new un-initialized nodes being added to the graph (Figure 5-2). Also, since an RIC can represent multiple numbers, the subtraction operation may result in multiple stack-locations as the result. This means that there is more than one possible stack configuration at that program point.

Adding new nodes, however, may not always be the best approach. For instance, if some number is subtracted from register *esp*, then it is referencing some location above the top of the stack. In this case, adding new un-initialized nodes to the stack graph is probably the correct approach. However, if some number is subtracted from register *eax* and *eax* points to

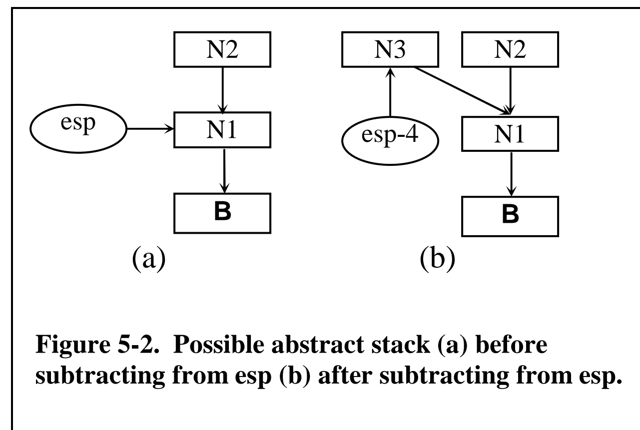

Figure 5-2. Possible abstract stack (a) before subtracting from esp (b) after subtracting from esp.

some stack-location, should new nodes be added to the graph or is it simply trying to access some stack-location that has been previously created?

Our solution to this problem is to add new nodes only when the node being subtracted from is “below” or equal to the stack pointer in the abstract stack graph. Thus, subtracting from *esp* will always result in new nodes being added, which is the desired result. If, on the other hand, *ebp* equals *esp + 4,* then subtracting four from *ebp* will result in *ebp* being set equal to *esp*. Again, this is the behavior we want, since clearly, we do not wish to create a new unitialized node in such a case, but would rather reference the already created node.

Subtracting an RIC minus a stack-location is undefined, since this would require knowing the actual address of the stack-location, something that we do not know. For similar reasons, a stack-location minus a stack-location results in an undefined value.

The function -* is provided to assist in adding multiple nodes to the abstract stack graph. It takes as input a stack-location, RIC, and state and recursively adds nodes, starting from the given stack-location and stopping once the specified number of nodes have been added. The

function also tracks the set of stack-locations that arise as a result of the subtraction. For example, *esp* minus the RIC 4[2,3] is equivalent to *esp* minus 8 and *esp* minus 12, and would cause three nodes to be added: *esp* - 4, *esp* - 8, *esp* - 12. Of these nodes, *esp* - 8 and *esp* - 12 are in the set of stack-locations resulting from the subtraction.

sub: VALUE $\times$ VALUE $\times$ STATE $\rightarrow$ VALUE $\times$ STATE

sub: $((a, \top), (c, \top), s) \rightarrow$	$((-(a, c), \top), s)$
sub: $((\top, b), (c, \top), s) \rightarrow$	$((\top, r), s_2)$, where $(r, s_2) = -(b, c, s)$
sub: $((a, b), (c, \top), s) \rightarrow$	$((-(a, c), r), s_2)$, where $(r, s_2) = -(b, c, s)$
sub: anything else $\rightarrow$	$((\top, \top), s)$

-: RIC $\times$ RIC $\rightarrow$ RIC

$-(R1, R2) = \sqcup\ R1 \boxplus -a$, where $a \in R2$

-: STACK_LOCATION $\times$ RIC $\times$ STATE $\rightarrow$ *P*(STACK_LOCATION) $\times$ STATE

-(s, R, state) = -*(s, R, state, $\emptyset$)

-*: STACK_LOCATION $\times$ RIC $\times$ STATE $\times$ *P*(STACK_LOCATION) $\rightarrow$
P(STACK_LOCATION) $\times$ STATE

-*(s, R, state, result) = let (s2, state2) = add-node(s, state) in

–*(s2, -(R,1), state2, $(1 \in R) \rightarrow (result \cup \{s2\})$ □ result) if some member of R is > 0

result $\times$ state if no member of R is > 0

The add-node function, which appears in the definition of -*, assists other functions by providing an easy mechanism to add new nodes to the stack. The nodes added are not initialized. This is useful in situations where some number is subtracted from *esp*. In these

cases, new nodes are added to the stack with undefined values. The add-node function returns a new state along the newly created stack-location.

add-node: STACK_LOCATION $\times$ STATE $\rightarrow$ STACK_LOCATION $\times$ STATE

add-node(loc, state) = m $\times$ (state$\downarrow$1, [m $\mapsto$ $(\top,\top)$]state$\downarrow$2, (m, loc) $\cup$ state$\downarrow$3))

Multiplication of two RICs results in an RIC that over-approximates the multiplication of each number expressed by the two RICs. Clearly, without knowing the actual address of a stack-location, it is not possible to multiply an RIC by a stack-location or multiply two stack-locations. Thus, these operations result in an undefined value.

mult: VALUE $\times$ VALUE $\rightarrow$ VALUE

mult: ((a, $\top$), (c, $\top$)) $\rightarrow$ (*(a,c), $\top$)
mult: anything else $\rightarrow$ $(\top,\top)$

*: RIC $\times$ RIC $\rightarrow$ RIC

*(R1, R2) = $\sqcup$ R1 $\times$ r, where r $\in$ R2

Division is even more restricted than multiplication. Any division attempt results in an undefined value, regardless of input. This is because division may result in a floating-point number, and the RIC structure does not yet handle floating-point numbers.

div: VALUE $\times$ VALUE $\rightarrow$ VALUE

div: anything $\rightarrow$ $(\top,\top)$

5.2 Memory Operations

The contents of arbitrary locations on the stack may be accessed and manipulated using the load, store, top, pop, and push functions.

The load function takes as input a stack-location and a state and returns the value that is located at the given stack-location. A future extension to this work will add a similar function for retrieving values stored at arbitrary memory locations such as the heap.

load: $\text{STACK_LOCATION} \times \text{STATE} \rightarrow \text{VALUE}$

load(location, state) = $\text{state}{\downarrow}2(\text{location})$

The store function takes as input a stack-location, a value, and a state and returns an updated state that holds the new value at the specified stack-location. Like the load function, this function will be improved to also update arbitrary memory locations in future versions.

store: $\text{STACK_LOCATION} \times \text{VALUE} \times \text{STATE} \rightarrow \text{STATE}$

store(loc, value, state) = $(\text{state}{\downarrow}1, [\text{loc} \mapsto \text{value}]\text{state}{\downarrow}2, \text{state}{\downarrow}3)$

The top function can be used to easily retrieve the value stored at the top of the stack. Since there may be more than one stack-location at the top of the stack at any given time, the union of these locations is returned as the result.

top: $\text{STATE} \rightarrow P(\text{VALUE})$

top(state) = $\sqcup\ \text{state}{\downarrow}2(\text{m})$, where $\text{m} \in \text{state}{\downarrow}1(\text{esp})$

Push and pop behave as one would expect. Push adds a value to the top of the stack and returns an updated state. Pop removes the value from the top of the stack and updates the state.

push: VALUE $\times$ STATE $\rightarrow$ STATE

push(value, state) = ([esp $\mapsto$ m]state$\downarrow$1, [m $\mapsto$ value]state$\downarrow$2, [$\cup$ (m, n)] $\cup$ state$\downarrow$3)

where n $\in$ (state$\downarrow$1(esp))$\downarrow$2

pop: REGISTER $\times$ STATE $\rightarrow$ STATE

pop(reg, state)= ([reg $\mapsto$ top(state), esp $\mapsto$ $\sqcup$ succ(1, n, state$\downarrow$3)]state$\downarrow$1, state$\downarrow$2, state$\downarrow$3)

where n $\in$ state$\downarrow$2(esp)

5.3 Miscellaneous Operations

The following operations have been created to perform various necessary tasks or to work as helper functions.

Reset is provided to easily create a new stack. In some cases, the analysis may not be able to determine which stack-location is the correct stack top. In these cases, a new stack is created. This involves simply setting the stack top (the *esp* register) equal to **B** (the bottom of the stack).

reset: STATE $\rightarrow$ STATE

reset(state) = ([esp $\mapsto$ ($\top$, {**B**})]state$\downarrow$1, state$\downarrow$2, state$\downarrow$3)

The make-value function provides an easy way to convert some input, such as a constant, into a value.

make-value: $\mathbb{N} \to \text{VALUE}$

make-value(c) = $(0\times[0,0]+c, \top)$

6 Evaluation Function

The evaluation function, $\mathcal{E}$, formally specifies how each x86 instruction is processed. It takes as input an instruction and a state and outputs a new state.

$\mathcal{E}$: INST × STATE → STATE

Processing a push or pop instruction is fairly easy. For push, a new value is created that represents the value being pushed and the state is modified such that the stack top points to the new value. Pop modifies the state such that the stack top points to the next node(s) in the abstract stack graph, effectively removing the old stack top.

$\mathcal{E}$ [m: push c], state = $\mathcal{E}$ (next(m), push(make-value(c), state))

$\mathcal{E}$ [m: push reg], state = $\mathcal{E}$ (next(m), push(state↓1(reg), state))

$\mathcal{E}$ [m: pop reg], state = $\mathcal{E}$ (next(m), pop(reg, state))

Anytime a hard-coded value is moved into register *esp*, the abstract stack graph is reset. Since the analysis does not track the addresses of stack-locations, we are unable to determine where the hard-coded value may point. Thus, analysis continues from this instruction with a new stack graph.

$\mathcal{E}$ [m: mov esp, c], state = $\mathcal{E}$ (next(m), reset(state))

Encountering an add or sub instruction requires performing the requested operation and updating the specified register in the state. Mult and div instructions are handled similarly.

$\mathcal{E}$ [m: add reg, c], state = let v = add(state↓1(reg), make-value(c), state) in

$\mathcal{E}$ (next(m), ([reg ↦ v]state↓1, state↓2, state↓3))

$\mathcal{E}$ [m: add reg1, reg2], state = let v = add(state↓1(reg1), state↓1(reg2), state) in

$\mathcal{E}$ (next(m), ([reg1 ↦ v]state↓1, state↓2, state↓3))

$\mathcal{E}$ [m: sub reg, c], state = let (v, state2) = sub(state↓1(reg), make-value(c),state) in

$\mathcal{E}$ (next(m), ([reg ↦ v]state2↓1, state2↓2, state2↓3))

$\mathcal{E}$ [m: sub reg1, reg2], state = let (v, state2) = sub(state↓1(reg1), state↓1(reg2), state) in

$\mathcal{E}$ (next(m), ([reg1 ↦ v]state2↓1, state2↓2, state2↓3))

When a call instruction is encountered, the address of the next instruction (the return address) is pushed onto the stack and analysis continues at the target of the call. In the case of an indirect call, the target of the call is determined by using value set analysis.

$\mathcal{E}$ [m: call c], state = $\mathcal{E}$ (inst(c), push(next(m), state))

$\mathcal{E}$ [m: call reg], state = $\mathcal{E}$ (inst(state↓1(reg)), push(next(m), state))

Jump instructions are handled in a manner similar to calls. When processing conditional jumps, each branch is analyzed and the results are merged. In the presence of indirect jumps, the value of the register being jumped to is retrieved and used as the target.

$\mathcal{E}$ [m: jmp c], state = $\mathcal{E}$ (inst(c), state)

$\mathcal{E}$ [m: jmp reg], state = $\mathcal{E}$ (inst(state↓1(reg)), state)

$\mathcal{E}$ [m: conditional jump to c], state = $\mathcal{E}$ (next(m), state) $\cup$ $\mathcal{E}$ (inst(c), state)

$\mathcal{E}$ [m: conditional jump to reg], state = $\mathcal{E}$ (next(m), state) $\cup$ $\mathcal{E}$ (inst(state↓1(reg)), state)

Processing a ret instruction involves retrieving the return address from the top of the stack and continuing analysis from there. Since the value retrieved from the stack may represent multiple addresses, each possible address is analyzed and the results are merged.

$\mathcal{E}$ [m: ret], state = $\sqcup$ $\mathcal{E}$ (inst(x), pop(state)), where x $\in$ top(state)

Handling a mov instruction is relatively straightforward. In all cases, some value needs to be stored at some location. That value is either immediately available in the instruction or must first be retrieved from some other location.

$\mathcal{E}$ [m: mov reg, c], state = $\mathcal{E}$ (next(m), ([reg $\mapsto$ make-value(c)]state$\downarrow$1, state$\downarrow$2, state$\downarrow$3))

$\mathcal{E}$ [m: mov [reg], c], state = $\mathcal{E}$ (next(m), store(state$\downarrow$1(reg), make-value(c), state))

$\mathcal{E}$ [m: mov reg1, reg2], state = $\mathcal{E}$ (next(m), ([reg1 $\mapsto$ state$\downarrow$1(reg2)]state$\downarrow$1, state$\downarrow$2, state$\downarrow$3))

$\mathcal{E}$ [m: mov reg1, [reg2]], state=

$\mathcal{E}$ (next(m), ([reg1$\mapsto$load(state$\downarrow$1(reg2))]state$\downarrow$1, state$\downarrow$2, state$\downarrow$3))

7 Analysis

After interpretation, the input file is analyzed, with the help of the ASG, to uncover any obfuscations. To recap, the abstract stack graph is a data structure that represents the actual stack by storing abstract values, instead of actual values. To assist in discovering obfuscations, each node in the ASG is modified to hold the address of the instruction that caused the node to be created. For instance, the sequence of instructions in Figure 7-1a would result in the graph shown in Figure 7-1b. Constructing the graph in this way allows for easy detection of modifications to the return address located on the stack.

Two different types of obfuscations can be detected with this approach: (a) obfuscations involving the replacement of a call instruction with some other combination of instructions, (b) obfuscations involving the replacement of the return address with a new address.

To detect obfuscations that involve simulating a call instruction with some combination of other instructions, the ASG at each *ret* instruction is examined. The instruction that created the node that is pointed to by register *esp* is retrieved. Note that this node is the top of the

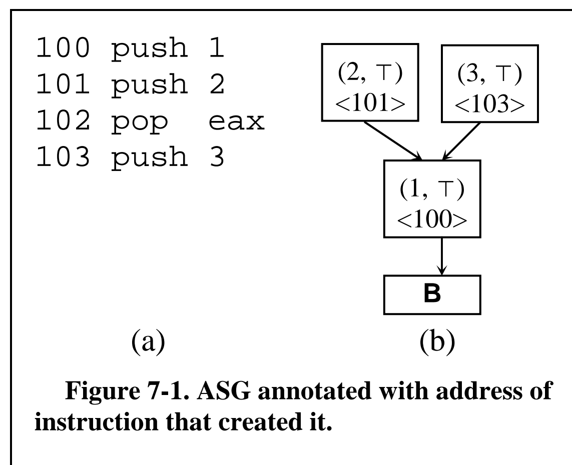

Figure 7-1. ASG annotated with address of instruction that created it.

stack and should hold the return address at this point. Since the current instruction is a return instruction, the top of the stack should have been created by a corresponding call instruction (the call instruction pushes the return address onto the stack). If any other instruction is responsible for placing the return address onto the stack, then a call obfuscation has been detected. That is, some other tactic was used to simulate a call instead of using the call instruction directly.

The other type of obfuscation involves removing the return address from the stack and replacing it with some other value. This type of obfuscation is a useful attack against most dissassemblers, since many of them typically assume that control will transfer back to the instruction that made the call, and thus will construct an incorrect control-flow graph.

To detect this obfuscation, we look for situations where the return address is popped off the stack. At each *pop* instruction, if the instruction that created the node at the top of the stack (the node to be popped) is a *call* instruction, then it is evident that the return address is being removed from the stack. Thus, the pop instruction is flagged as an obfuscation.

8 Examples

The following sections contain examples demonstrating the analysis of various obfuscation techniques. Section 8.1 contains a rather detailed example intended to explain the analysis process. The remaining sections briefly describe how this approach can be used to analyze other obfuscations.

8.1 Using Push/Jmp

Figure 8-1 contains a sample assembly program that will be used as an example for the remainder of this section. The program consists of two functions: *Main* and *Max*. *Max* takes as input two numbers and returns as output the larger of the two numbers.

The function *Main* pushes the two arguments onto the stack, but instead of calling *Max* directly, it pushes the return address onto the stack and jumps to *Max*. This technique can effectively hide the boundary between the two procedures and result in a less accurate CFG. Analysis methods relying on the flow graph may, in effect, produce less accurate results as well.

```
Main:                         Max:
L1:  PUSH 4                   L7:  MOV  eax, [esp+4]
L2:  PUSH 2                   L8:  MOV  ebx, [esp+8]
L3:  PUSH offset L5           L9:  CMP  eax, ebx
L4:  JMP  Max                 L10: JG   L11
L5:  PUSH 0                   L11: MOV  eax, ebx
L6:  CALL ExitProcess         L12: RET  8
```

Figure 8-1. Obfuscated call using push/jmp.

8.1.1 Interpretation

Upon entry, all registers are initialized to ⊤, signaling that their values have not yet been determined. The stack is currently empty as is the mapping of stack-locations to values, since there is no stack content yet (Figure 8-2a).

Instruction L1 pushes a value onto the stack. The value pushed is the RIC 0[0,0] + 4, or simply 4. A new stack-location is created to hold this value and is added to the set of edges in the abstract stack graph that connects the new stack-location to the bottom of the stack (Figure 8-2b). Notice that register *esp* is modified so that it references the stack-location that is the new top of the stack.

Instructions L2 and L3 perform in a manner similar to L1. L3, however, pushes an instruction address onto the stack. In this example, we will represent the addresses of instructions by using the instruction's label. However, in practice, the actual address of the instruction is used instead and can easily be represented using an RIC (Figure 8-2c).

L4 is an unconditional jump. Control is transferred to the destination of the jump and the state is left unchanged.

The next instruction evaluated is the target of the jump, or L7 in this case. L7 is a mov instruction that moves the value located at *esp*+4 into register *eax* (Figure 8-2d).

Instruction L8 performs in a manner similar to L7. Instruction L9 has no effect on the state.

Instruction L10 is a conditional jump and does not change the state. During evaluation, each possible target will be processed and each resulting state is joined once the two execution paths meet.

Instruction L11 copies the value from *ebx* to *eax* (Figure 8-2e).

The *ret 8* instruction at L12 implicitly pops the return address off the top of the stack and continues execution at that address. It also adds 8 bytes to *esp*. This causes *esp* to be incremented by 2 stack-locations (since each stack-location holds 4 bytes). However, since L12 can be reached from L10 and L11, the results of evaluating the two paths must be joined before processing L12. Creating the union of the two states is easy in this case. The only difference between the two is the value of *eax*. At instruction L10, *eax* is 2, whereas at instruction L11, *eax* is 4. The union of the two is the set {2, 4}, or the RIC 2[1,2]+0 (Figure 8-2f).

Evaluation continues at L5, which ends the program.

Looking at the final state, we see that *eax* may hold either 2 or 4 and *ebx* equals the constant 4. Note that a quick scan of the code reveals that *eax* will actually always equal 4 at L5. The analysis assumed that the jump at L9 might pass execution to instruction L10 or L11. However, execution will always continue at L10, because *eax* is always less than *ebx* at L8. This does not mean the analysis is incorrect. One goal of VSA is to play it safe and over-approximate the actual values, hence the discrepancy. Applying techniques used in compilers, such as dead code elimination, may assist in providing more accurate results.

8.1.2 Analysis

Let's first look at the case of detecting call obfuscations (i.e. obfuscations that simulate a call). To do this, the algorithm searches for all *ret* instructions. There is one at L12. The

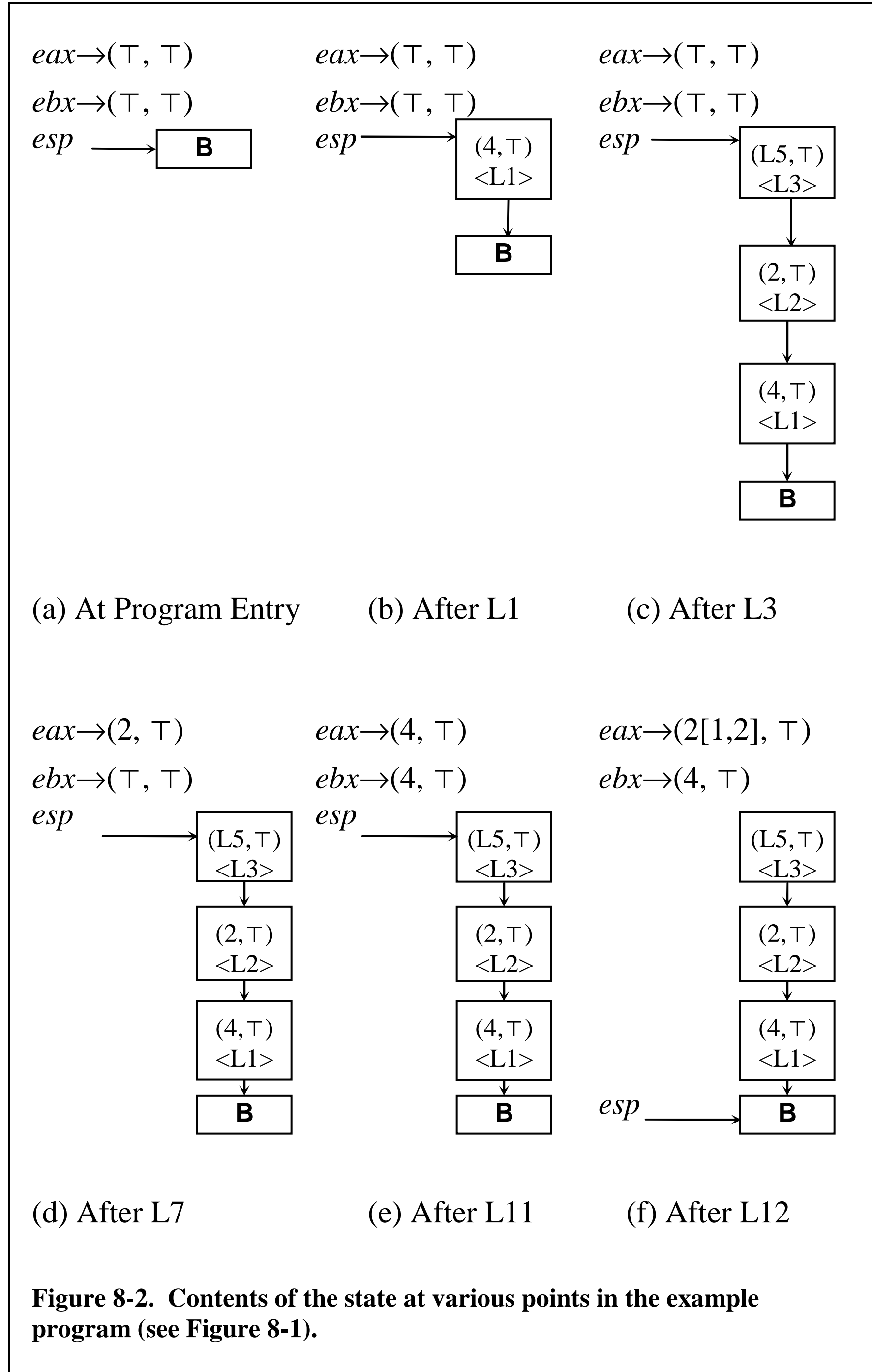

Figure 8-2. Contents of the state at various points in the example program (see Figure 8-1).

state just before L12 is executed is shown in Figure 8-2e. We see that the top of the stack was created by instruction L3, which happens to be a *push* instruction, not a call. We therefore conclude that the return address was placed on the stack via unconventional means and no *call* was ever made. Indeed, a quick review of the code reveals that a *jmp* was used instead.

To detect return obfuscations (i.e. obfuscations that remove the return address), the algorithm searches for all *pop* instructions. In this sequence of code, there are none, so nothing needs to be done.

8.2 Using Push/Ret

Figure 8-3a shows the same code, but using the push/ret obfuscation. Instructions L3 and L4 push the return address and the target address onto the stack. L5 consists of a *ret* that causes execution to jump to the function *Max*. Analysis methods that rely on the correctness of a CFG will surely fail when analyzing such code.

8.2.1 Interpretation

During the interpretation, at instruction L5, there are four nodes in the abstract stack, as shown in Figure 8-3b. At the top of the abstract stack is the address of the function *Max*. When the *ret* is encountered, analysis continues at this address and *esp* is incremented so that it points to the node containing (L6, $\top$). Thus, L6 becomes the return address of the *Max* procedure (Figure 8-3c).

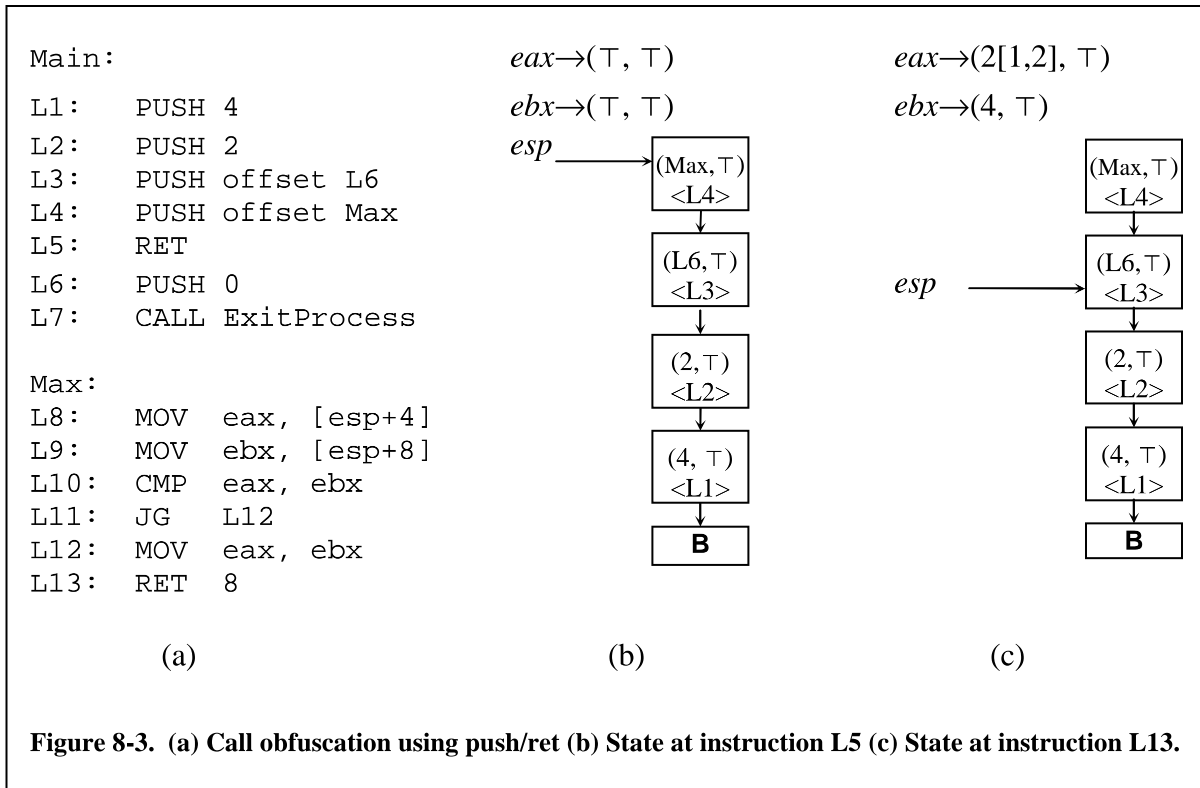

Figure 8-3. (a) Call obfuscation using push/ret (b) State at instruction L5 (c) State at instruction L13.

8.2.2 Analysis

There are two *ret* instructions in the code, one at L5 and the other at L13. The state just before executing instruction L5 is shown in Figure 8-3b. The node at the top of the stack was created by instruction L4, a *push* instruction, not a *call*. It is therefore evident that the *ret* is used as a *call*.

The second *ret* occurs at L13 and the corresponding state is shown in Figure 8-3c. The node at the top of the stack was created by instruction L3. Again, this node is not a *call*, but a *push*, and is identified as an obfuscation attempt.

8.3 Using Pop to Return

In Figure 8-4a, the function Max is invoked in the standard way, however it does not return in the typical manner. Instead of calling *ret*, the function pops the return address from the stack and jumps to that address (lines L11-L13).

8.3.1 Interpretation

At instruction L11, the stack contains four nodes, as shown in Figure 8-4b. L11 removes the value from the top of the stack and places it in *ebx*. L12 adds eight to *esp*, which causes *esp* to point to the bottom of the stack. L13 is an indirect jump to the address in *ebx*. Looking at the stack at instruction L13 (Figure 8-4c), *ebx* contains (L4, ⊤), and thus analysis continues at

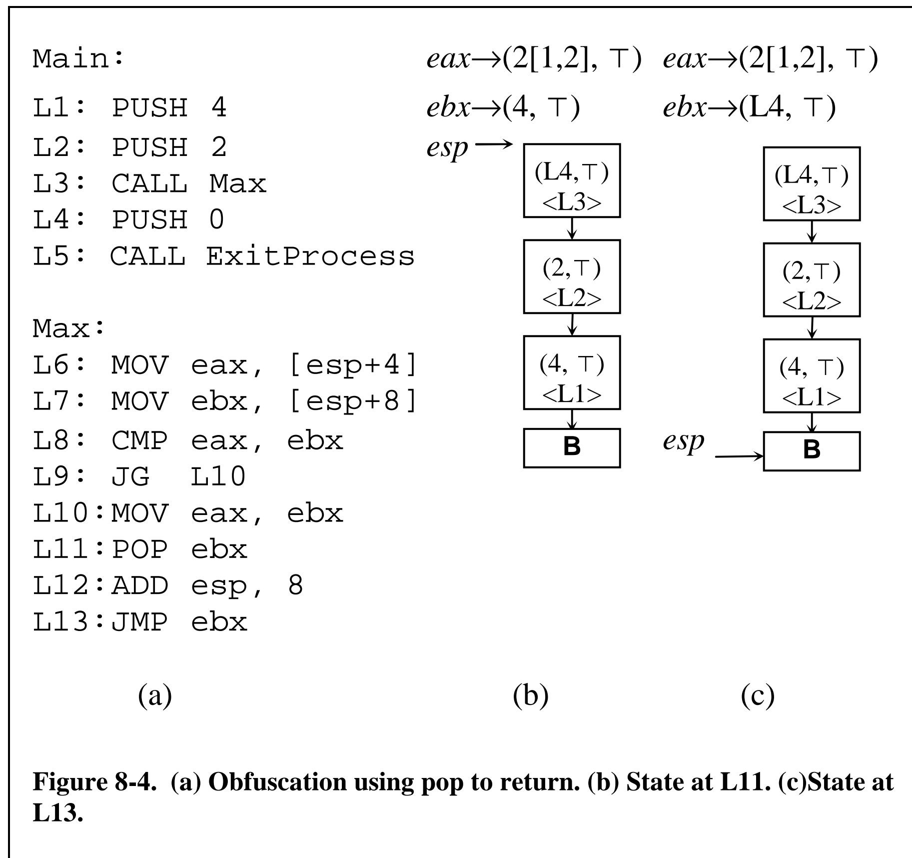

Figure 8-4. (a) Obfuscation using pop to return. (b) State at L11. (c)State at L13.

instruction L4, the original return address.

8.3.2 Analysis

The code in Figure 8-4a does not contain any *ret* instructions, but does contain one *pop* at L11. Figure 8-4b shows the state just before executing instruction L11. By examining the state, we see that the node to be popped at L11 was created by instruction L3, which is a *call* instruction. Thus, instruction L11 pops off the return address and is flagged as an obfuscation.

8.4 Modifying Return Address

In Figure 8-5a, the procedure *Max* pops the original return address and replaces it with an alternate address to transfer control to a function other than the caller. In this example, control transfers to L30, which is not shown.

8.4.1 Interpretation

At instruction L11, the top of the stack originally contains (L4, $\top$) (Figure 8-5b). L11 removes this value from the stack and L12 pushes the value (L30, $\top$) onto the stack. Figure 8-5c shows the resulting state. The ret statement at L13 causes interpretation to continue at instruction L30.

8.4.2 Analysis

In this example, there are two instructions of interest: a *pop* at L11 and a *ret* at L13.

At instruction L11 (see Figure 8-5b for the state), the value at the top of the stack is L4, the return address. This value was placed onto the stack by instruction L3, a *call*. Thus, the program is removing the return address from the stack, something only the *ret* instruction should do. This instruction would be flagged as an obfuscation.

Instruction L13 is a *ret* and the top of the stack was created by instruction L12, a *push*. Again, we would expect the return address to have been created by a *call* instruction, but that is not the case. Thus, the instruction is flagged.

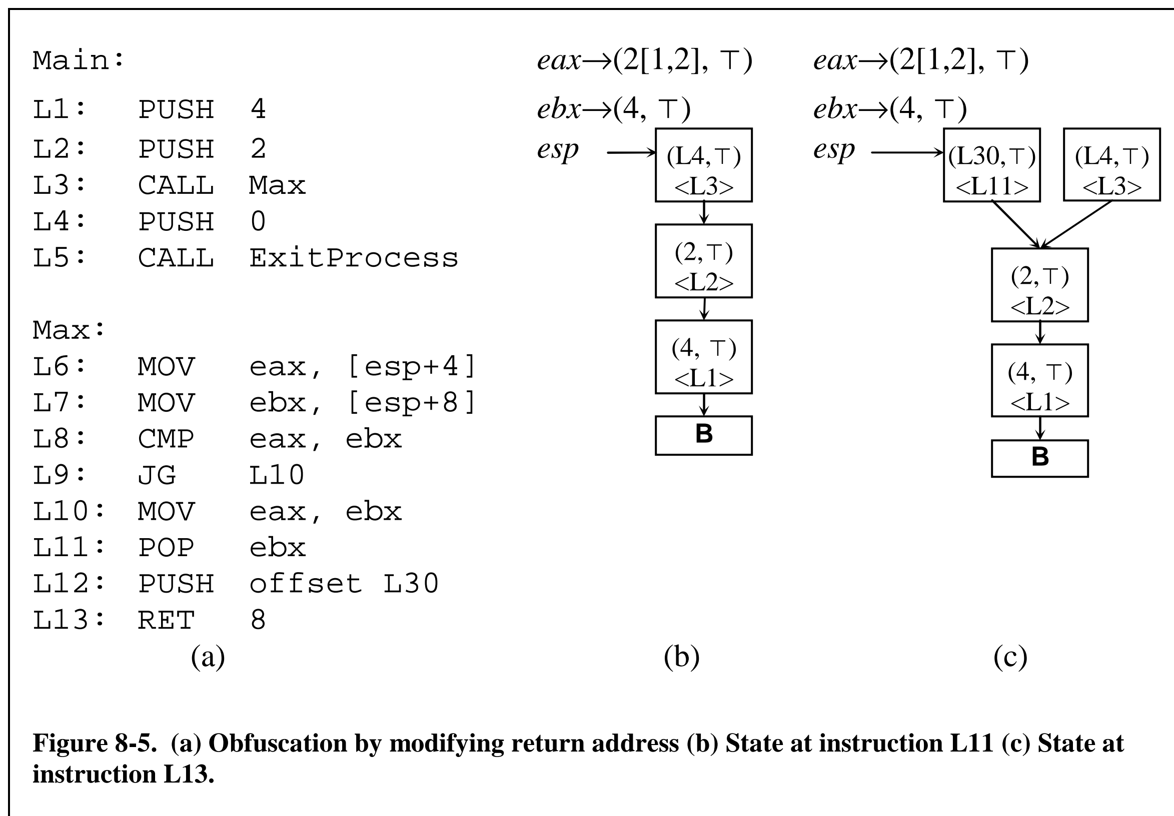

Figure 8-5. (a) Obfuscation by modifying return address (b) State at instruction L11 (c) State at instruction L13.

9 Prototype

A prototype has been constructed that uses the previous ideas to detect obfuscations in programs. The prototype is written in Java utilizing the Eclipse framework. Eclipse is an extensible development environment with a rich set of tools to aid in development [21]. Programs developed on Eclipse are written as plugins to the Eclipse platform and can take advantage of the robust Eclipse framework to decrease development time.

9.1 Phases

The entire detection process consists of four phases: disassembly, parsing, interpretation, and analysis. All phases, with the exception of disassembly, were written anew specifically for this project. For disassembly, however, the debugger/disassembler OllyDbg is used. An illustration of the process is shown in Figure 9-1.

9.1.1 Disassembly

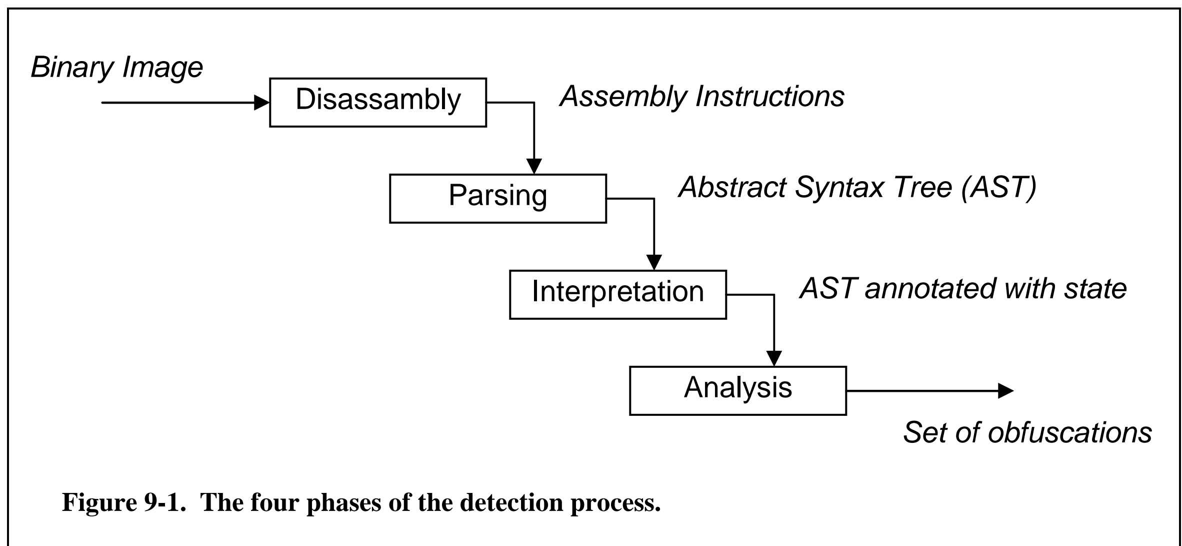

Figure 9-1. The four phases of the detection process.

Since writing a disassembler from scratch is a monumental task of itself, we've decided to utilize the OllyDbg debugger for this purpose. A binary x86 executable is fed into OllyDbg which outputs an ASCII text file containing the executable instructions as assembly code. This text file is then fed to the next phase, the parser.

It is important to understand any assumptions put forth by the disassembler of choice. Any disassembler we use, unless specifically designed for the disassembly of malware, will likely contain key assumptions made by the programmers that translate into attack points for malicious code writers.

As already mentioned, the prototype uses the Ollydbg disassembler/debugger for disassembly. OllyDbg appears to use the recursive-traversal method of disassembly (see Chapter 3.1 for an explanation of the recursive-traversal method) and is thus susceptible to the same shortcomings as any recursive-traversal disassembler. While the weaknesses of the disassembly are indeed important, they are much less important in the context of this project. We are primarily interested in showing the feasibility of using abstract interpretation and the abstract stack graph as a means to discover obfuscated function calls, a task that requires, but is separate from, disassembly. Therefore, we narrow the scope of this project to exclude issues concerning disassembly and have chosen to focus our efforts on executables for which a reliable disassembly can be obtained using OllyDbg. This does not indicate a weakness of our approach, but instead is a weakness of the chosen disassembly method and can be improved by using an alternate method. In fact, the design of the system allows for the easy exchange of one disassembler for another, provided a suitable parser exists for the disassembler. Thus, one can find a disassembler that minimizes the appropriate weaknesses

relevant to the task at hand or even incorporate a custom-written disassembler if necessary. The only chore needed would be to specify the grammar describing the output of the disassembler.

9.1.2 Parsing

Once the binary executable has been disassembled into assembly instructions, it is parsed by the program and translated into a suitable internal representation to be used for the next phase: interpretation. Parsing is a relatively simple task requiring that only a grammar describing the input to be parsed be provided. This grammar can be swapped with other grammar descriptions, opening the possibility of handling other input formats such as disassembled code from other disassemblers.

9.1.3 Interpretation

Interpretation is, by far, the most complicated phase of the entire process. The goal of interpretation is to compute abstract values where concrete values ordinarily reside [16]. The abstract value may over-approximate the corresponding concrete value, but should never under-approximate it. After interpretation, it should be possible to observe the state at any given instruction and see what values some particular register or piece of memory may hold. Interpretation can provide answers to questions like "what values may register *eax* hold at instruction two?"

To accomplish this, an implementation of the evaluation function discussed in Chapter 6 was developed along with implementations of the state and operations applicable to the state as

described in Chapters 4 and 5. The definitions given in Chapters 4, 5, and 6 were followed very closely, but special care is needed to ensure the program terminates.

To ensure termination, an instruction is not interpreted if (a) it has been interpreted before and (b) the state previously used to interpret the instruction over-approximates the current state. To handle branches and loops, a control-flow graph (CFG) is constructed while interpreting and two different types of CFG nodes are defined, similar to the definitions introduced by Cousot and Cousot [16]. These nodes are simple junction nodes and loop junction nodes.

Simple junction nodes are the result of two or more paths of a conditional jump joining back together (Figure 9-2a). When this situation arises, we merge the two states obtained from the two possible paths, where merging states requires merging each value contained within the two states. Doing so ensures that, at the instruction in question, the state will contain an over-approximation of all possible paths that lead to that instruction.

Loop junction nodes result from loops in the code (Figure 9-2b). In this case, we would like to minimize the number of times the loop is interpreted. Merging the new state with the previous state would provide the correct result, but would require that we still interpret the loop many times. Instead, we must widen the state, which requires that each value in the state be widened relative to the values in the new state. For example, if register *eax* is initialized to 1 and is changed to 2 inside the loop, the loop junction node would have been encountered twice, once for eax=0[0,0]+1 and once for eax=[0,0]+2. At this node, we widen register *eax* which results in eax=1[1,∞]+0. It may be possible to result in a more accurate

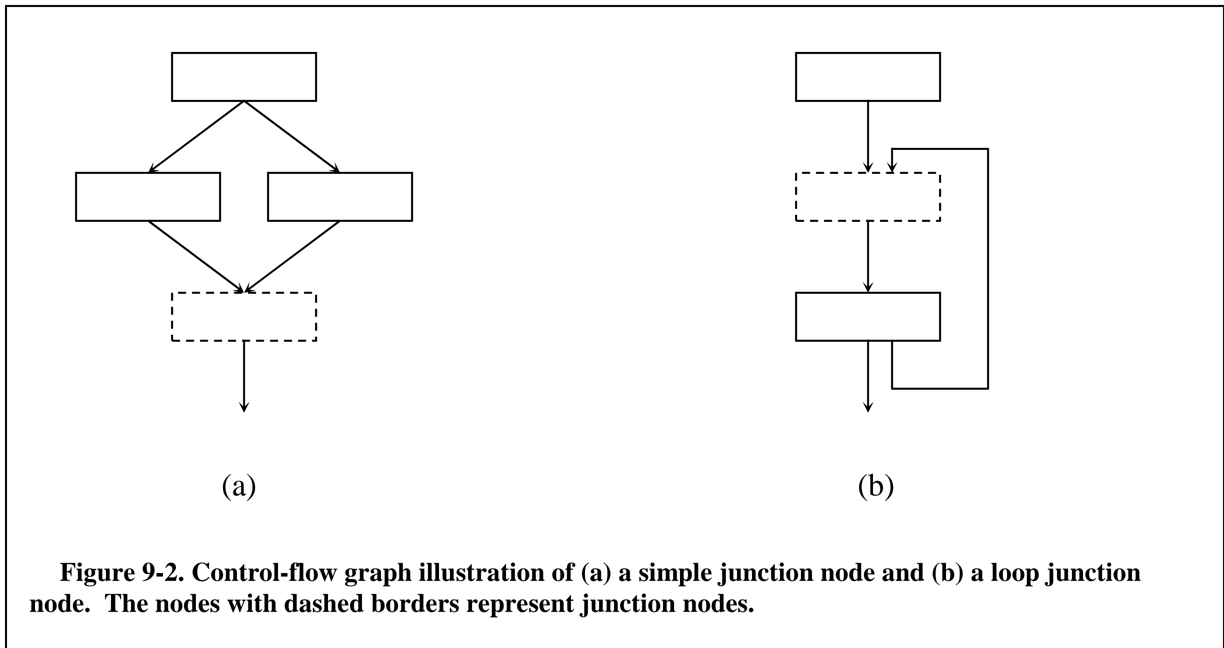

Figure 9-2. Control-flow graph illustration of (a) a simple junction node and (b) a loop junction node. The nodes with dashed borders represent junction nodes.

value. For instance, if, during the loop, *eax* ranges from 1 to 10, then widening *eax* should ideally result in the value 1[1,10]+0. The current implementation does not take this information into account and will always set the range to infinity. While this is a large over-approximation, it is still a valid result.

9.1.4 Analysis

The final phase, and perhaps the simplest, is the analysis. The goal of analysis is to find all call obfuscations hidden in the executable. The analysis phase has at its disposal the disassembled executable where each instruction is annotated with the state at that particular instruction. With the state readily available, the software can use the abstract stack graph to carry out the analysis as described in Chapter 7. Once completed, the results of the analysis are displayed to the user by highlighting the obfuscated instructions.

9.2 Implementation Results

The resulting application is able to detect the various forms of obfuscation mentioned previously. A screenshot of the resulting application is shown in Figure 9-3. The prototype places markers on the ruler next to each obfuscation call, and obfuscated calls are also displayed as a list. Also shown are two panes containing the register and stack contents at the selected instruction.

One limitation of the current implementation involves efficiency. In particular, a more efficient form of interprocedural analysis is needed and, perhaps, better management of the abstract stack graph. In the meantime, we have confined ourselves to running the software on smaller files that contain the same obfuscations found in larger malware samples.

Another limitation worth noting is the lack of support for structured exception handling (SEH). SEH is a programming mechanism useful for detecting serious errors and transferring control to code designed to handle the errors. Malicious programmers sometimes use SEH as another way to control the flow of execution by intentionally causing an error to occur (such as a divide by zero) and placing the malicious code at the location intended for code that will handle the error. The interpreter does not implement SEH and therefore is vulnerable to such attacks. This weakness can result in reachable code that is never processed by the interpreter.

Non-obfuscated call site

Register and stack contents displayed on right

Obfuscations are shown on the side ruler and near the bottom as a list

Figure 9-3. Prototype user interface.

10 Results

We began this project with the aim of creating an application capable of detecting call obfuscations in malicious software. The intended users include malware analysts who would likely use the tool as a means of identifying an executable as potentially malicious and for finding system calls that may otherwise have been concealed. The tool could also potentially find use in a malware scanner that uses heuristics; since many heuristic scanners rely on knowledge of system calls to determine malicious activity, having a module capable of identifying obfuscated system calls could prove to be valuable.

Previous work in this area has been performed by Kumar and Lakhotia [4, 22] which resulted in the development of a similar tool. Their tool also uses an abstract stack graph with positive results. Our tool extends on this previous work by using abstract values to keep track of program registers and memory contents. Our hope was to create a system that is capable of finding call obfuscations via the stack graph even in the face of instructions such as *sub esp, eax*. Such an instruction proves challenging for previous work, since no effort was made to determine the value stored in *eax*. When faced with such an instruction, the previous work simply resorted to setting register *esp* to the bottom of the stack, essentially creating a new stack graph. Doing so, no doubt, reduces the precision of the results.

Including abstract values also makes it possible to overcome another tricky situation, namely understanding system calls made via the functions *GetModuleHandle* and *GetProcAddress*. A program can use this pair of functions to determine the address of any system function at runtime. The function address is placed in register *eax* and the function can be called by the

instruction *call eax*. Since this technique is commonly found in viruses, it would be ideal to have support for it.

The work proposed in this thesis provides what appears to be one way of supporting functions like *GetModuleHandle*/*GetProcAddress*. The current implementation, however, does not. The lack of support is partly due to the function *GetModuleHandle*, which takes as a parameter a pointer to a string containing the name of the DLL (dynamic link library) to retrieve a handle for. In an attempt to keep the size of the project to a reasonable level, we have decided not to support arbitrary memory locations in the current system (only registers and the stack are supported). Thus, it is not possible for us to handle this style of system call invocation until memory is fully supported, something to be added in a future version.

A somewhat surprising limitation that was encountered relates to the reduced-interval congruence (RIC) values and certain types of instructions. Instructions such as *add eax, ebx* are easily interpreted using RICs, since adding two RICs is a somewhat trivial task. However, a much more complicated instruction is *or eax, ebx*, which requires the *or* operation to be applied to each bit in *eax* and *ebx*. RICs do not appear to be designed for this particular situation, and, indeed, finding an efficient algorithm for handling instructions such as *or*, *and*, *xor*, etc. has proven to be challenging. As of now, these instructions result in undefined values. From experimentation, we have found that on large programs, many values end up becoming undefined due to such functions. This is an area that needs to be improved.

Having looked at how the abstract stack is used, one can construct new forms of obfuscations that can circumvent this approach. For instance, the code shown in Figure 10-1a pushes the value five onto the stack and removes that value from the stack immediately after. Instruction L3 subtracts from the stack pointer, which effectively places the five at the top of the stack again. At L4, the value five is placed into *eax*.

The stack graph that would be created is shown in Figure 10-1b. At instruction L4, *esp* points to a value that has not been initialized. It is this value that is placed into *eax*, not the value five. Thus, the analysis is incorrect for this piece of code. The cause is the assumption that subtracting from register *esp* implies a new node should be created in the stack graph. While this assumption may be correct for compiler-generated code, hand-crafted assembly need not follow this convention. Other variations of this theme exist.

Another possible attack is in the over-approximation of the values. If the analysis over-approximates a value too much, the analysis is less useful. Code can be crafted to intentionally force the analysis to over-approximate important values, such as the targets of indirect jumps. In future work, we will study these attack vectors and determine how these obstacles can be overcome.

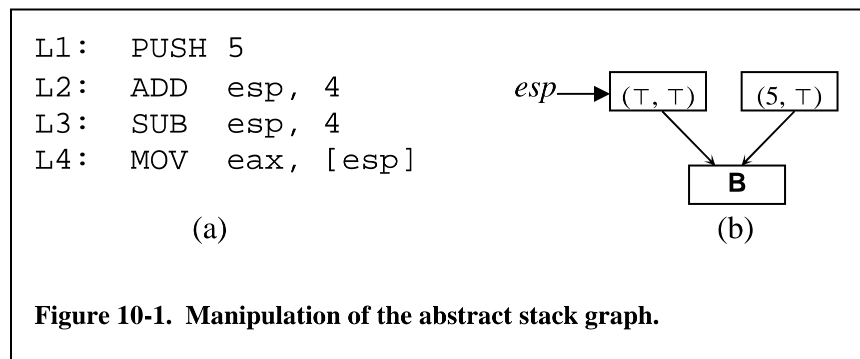

Figure 10-1. Manipulation of the abstract stack graph.

Despite its limitations, we did find the program to perform its task well. Having abstract values allows for a larger variety of instructions to be correctly processed without sacrificing the integrity of the abstract stack graph (we resort to resetting the stack graph on fewer occasions). With the addition of arbitrary memory support, functions like *GetProcAddress* should be an easy task to resolve, and by defining more operations to be applied to RICs, there should be fewer occasions where an RIC is given an undefined value. Finally, a more efficient implementation of the stack graph would allow for larger files to be processed quickly.

11 Conclusion

By using an abstract stack graph as an abstraction of the real stack, we are able to analyze a program without making any assumptions about the presence of activation records or the correctness of the control-flow graph.

The method presented here can be used to statically determine the values of program variables. The method uses the notion of reduced interval congruence to store the values, which allows for a tight approximation of the true program values and also maintains stride information useful for ensuring that memory accesses do not cross variable boundaries. The reduced interval congruence also makes it possible to predict the destination of jump and call instructions.

The potential benefit of this approach is in statically detecting obfuscated calls. Static analysis tools that depend on knowing what system calls are made are likely to report incorrect results when analyzing a program that contains call obfuscations. The consequences of falsely claiming a malicious file as benign can be extremely damaging and equally expensive to repair. Thus it is important to locate system calls correctly during analysis. The techniques discussed in this thesis can be applied to help uncover obfuscated calls and provide for a more reliable analysis.

Bibliography

[1] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Department of Computer Science, The University of Auckland, 148, July 1997.

[2] G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in X86 Executables," in *Proc. International Conference on Compiler Construction*, Barcelona, Spain, 2004. pp. 5-23.

[3] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Conference on Computer and Communications Security*, Washington D.C., USA, 2003. pp. 290-299.

[4] A. Lakhotia and E. U. Kumar, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries," in *Fourth IEEE International Workshop on Source Code Analysis and Manipulation(SCAM'04)*, Chicago, Illinois, 2004. pp. 17-26.

[5] J. Bergeron and M. Debbabi, "Detection of Malicious Code in Cots Software: A Short Survey," in *First International Software Assurance Certification Conference (ISACC'99)*, Washington DC, 1999. pp. 275-284.

[6] Symantec, "Understanding Heuristics: Symantec's Bloodhound Technology," http://www.symantec.com/avcenter/reference/heuristc.pdf, Last accessed July 1, 2004.

[7] F. Cohen, *Computer Viruses*, University of Southern California Thesis, 1985.

[8] D. M. Chess and S. R. White, "An Undetectable Computer Virus," in *Virus Bulletin Conference*, 2000.

[9] A. Lakhotia, A. Kapoor, and E. U. Kumar, "Are Metamorphic Viruses Really Invincible? - Part I," *Virus Bulletin*, 2004, pp. 5-7.

[10] A. Lakhotia, A. Kapoor, and E. U. Kumar, "Are Metamorphic Viruses Really Invincible? - Part II," *Virus Bulletin*, 2005, pp. 9-12.

[11] P. Ször and P. Ferrie, "Hunting for Metamorphic," in *Virus Bulletin Conference*, Prague, Czech Republic, 2001. pp. 123-144.

[12] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *12th USENIX Security Symposium*, Washington, D.C, 2003. pp. 169-186.

[13] C. M. Jha, et al., "Semantics-Aware Malware Detection," *IEEE Symposium on Security and Privacy*, pp. 32-46, 2005.

[14] M. Mohammed, *Zeroing in on Metamorphic Computer Viruses*, Center for Advanced Computer Studies, University of Louisiana at Lafayette, M.S. Thesis, 2003.

[15] "Idapro," Data Rescue, Liege, Belgium, http://datarescue.com, Last accessed January 31, 2005.

[16] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs," in *2nd Int. Symp. on Programming*, Dumod, Paris, France, 1976. pp. 106-130.

[17] G. Balakrishnan, et al., "Codesurfer/X86 -- a Platform for Analyzing X86 Executables," in *Conference on Compiler Construction*, Edinburgh, Scotland, 2005. pp. 250-254.

[18] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *Ninth Working Conference on Reverse Engineering (WCRE'02)*, Richmond, VA, 2002. pp. 45-54.

[19] L. Vinciguerra, et al., "An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java," in *10th Working Conference on Reverse Engineering*, 2003. pp. 14-23.

[20] A. Lakhotia and P. K. Singh, "Challenges in Getting 'Formal' with Viruses," *Virus Bulletin*, 2003, pp. 15-19.

[21] "The Eclipse Project," The Eclipse Foundation, http://eclipse.org, Last accessed February 7, 2005.

[22] E. U. Kumar, A. Kapoor, and A. Lakhotia, "Doc--Answering the Hidden 'Calls' of Virus," *Virus Bulletin*, 2005, pp. 7-10.

ABSTRACT

Programmers obfuscate their code to defeat manual or automated analysis. Obfuscations are often used to hide malicious behavior. In particular, malicious programs employ obfuscations of stack-based instructions, such as call and return instructions, to prevent an analyzer from determining which system functions they call. Instead of using these instructions directly, a combination of other instructions, such as PUSH and POP, are used to achieve the same semantics. This thesis presents an abstract interpretation based analysis to detect obfuscation of stack instructions. The approach combines Balakrishnan's and Reps's value set analysis (VSA) and Lakhotia's and Kumar's abstract stack graph, to create an analyzer that can track stack manipulations where the stack pointer may be saved and restored in memory or registers. This analysis technique may be used to determine obfuscated calls made by a program, an important first step in detecting malicious behavior.

Biographical Sketch

Michael Venable, son of Jeffery and Patti Venable, was born in Lafayette, Louisiana, on August 20, 1980. He received his bachelor's degree in computer science from the University of Louisiana at Lafayette in 2002, and after a brief stint as a computer programmer at a local software company, he began working on his master's degree at UL Lafayette. After receiving his master's degree, Michael will continue to work at UL Lafayette as a software engineer with a focus in the area of computer security.