Analyzing Memory Accesses in Obfuscated x86 Executables

Michael Venable, Mohamed R. Chouchane, Md Enamul Karim, and Arun Lakhotia

Center for Advanced Computer Studies, University of Louisiana at Lafayette, LA
{mpv7292, mohamed, mek, arun}@louisiana.edu

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 it calls. Instead of using these instructions directly, a combination of other instructions, such as PUSH and POP, are used to achieve the same semantics. This paper presents an abstract interpretation based analysis to detect obfuscation of stack instructions. The approach combines Reps and Balakrishnan's value set analysis (VSA) and Lakhotia 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. The analysis technique may be used to determine obfuscated calls made by a program, an important first step in detecting malicious behavior.

1 Introduction

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, 2]. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners [3]. The concern here 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 tool. Some known obfuscation techniques include: variable renaming, code encapsulation, code reordering, garbage insertion, and instruction substitution [2]. We are interested in instruction substitution of codes performed at the assembly level, particularly for call obfuscations.

A common obfuscation observed in malicious programs is obfuscation of call instructions [3]. 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

K. Julisch and C. Kruegel (Eds.): DIMVA 2005, LNCS 3548, pp. 1 – 18, 2005.

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

Fig. 1. Sample use of call obfuscation

pushing the address addr (the target of the procedure call). The third instruction, ret, causes execution to jump to addr, simulating a call instruction. Fig. 1 illustrates another form of obfuscation. In this example, Line L3 pushes the return address onto the stack and line L4 jumps to the function entry. No call statement is present. The code may be further obfuscated by spreading the instructions and by further splitting each instruction into multiple instructions.

Metamorphic viruses are particularly insidious because two copies of the virus do not have the same signature. A metamorphic virus transforms its code during each new infection in such a way that the functionality is left unchanged, but the sequence of instructions that make up the virus is different [4]. As a result, they are able to escape signature-based anti-virus scanners [5, 6]. Such viruses can sometimes be detected if the operating system calls made by the program can be determined [7]. For example, Symantec's Bloodhound technology 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 [8].

The challenge, however, is in detecting the operating system calls made by a program. The PE and ELF 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. The obfuscation also takes away important cues that are used during manual analysis. We are then left only with 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 execute its malicious payload. 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 paper incorporates the work from [9] with the work discussed in [3]. In particular, the notion of Reduced Interval Congruence (RIC) will be employed to approximate the values that a register may hold. However, unlike in [9], registers such as *esp* will hold values that specifically represent 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.

This paper is organized as follows. Section 2 discusses work related to the area of static analysis. Section 3 defines the domain that encompasses this work. Sections 4 and 5 consist of formal specifications of various functions used during the analysis. Section 6 contains an example demonstrating the analysis process. Section 7 describes our future goals in this area and section 8 concludes this paper.

2 Related Work

In [1], Linn and Debray describe several code obfuscations that can be used to thwart static analysis. Specifically, they attack disassemblers by inserting junk statements at locations where the disassembly is likely to expect code. 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 following 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 jmp 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 diasassembled when using the popular disassembler IDA Pro from DataRescue. 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 [9] 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, such as [10], discuss how intervals can be used to statically determine values of variables in a program, 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.

The 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 under analysis. Incorrect results may arise when applied to an executable consisting of obfuscations typically found in malicious programs.

Kumar and Lakhotia [3] 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 [3] 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 in [9] for analyzing the content of memory locations.

3 Definitions

The domain of our analysis method consists of RICs, stack-locations, values, and a state. They are briefly discussed below.

3.1 RIC

A Reduced Interval Congruence (RIC) 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 [10] 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 [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 [3,9] 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:

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

3.2 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.

3.3 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 $\bot$, which represents the bottom of the stack.

More formally,

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

3.4 State

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

Formally,

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

4 Operations

4.1 Arithmetic Operations

Functions are defined for performing various arithmetic operations on 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 (Fig. 2). 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.

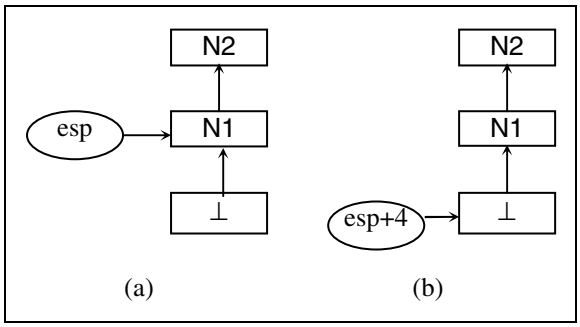

Fig. 2. Possible abstract stack (a) before adding to *esp* (b) after adding to *esp*

add: VALUE × VALUE × STATE → VALUE

INPUT		RESULT
(a, ⊤) × (c, ⊤) × s	→	(+(a, c), ⊤)
(a, ⊤) × (⊤, d) × s	→	(⊤, +(a, d, s))
(a, ⊤) × (c, d) × s	→	(+(a,c), +(a, d, s))
(⊤, b) × (c, ⊤) × s	→	(⊤, +(c, b, s))
(a, b) × (c, ⊤) × s	→	(+(a,c), +(c, b, s))
Anything else	→	(⊤,⊤)

+: RIC × RIC → RIC

+(R1, R2) = ⊔ R1 ⊞ a, where a ∈ R2

+: RIC × STACK_LOCATION × STATE → P(STACK_LOCATION)

+(R, s, state) = ⊔ r^{th} successor of s, where r ∈ R

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

⊞: RIC × ℕ → RIC

(a[b,c]+d) ⊞ 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 (Fig. 3). 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.

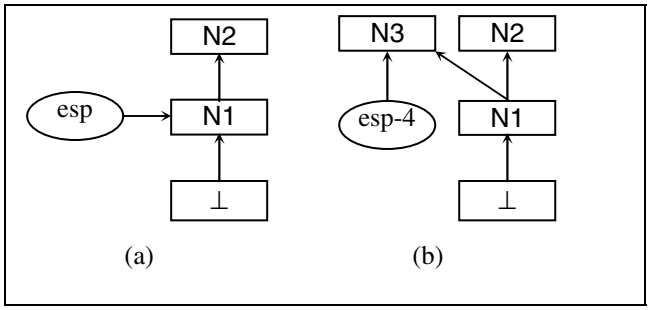

Fig. 3. Possible abstract stack (a) before subtracting from *esp* (b) after subtracting from *esp*

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 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? Further work in this area will help determine the best answer.

Moving on, 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 × VALUE × STATE → VALUE × STATE

INPUT		RESULT
(a, ⊤) × (c, ⊤) × s	→	(-(a, c), ⊤) × s
(⊤, b) × (c, ⊤) × s	→	(⊤, r) × s_2, where (r, s_2) = -(b, c, s)
(a, b) × (c, ⊤) × s	→	(-(a, c), r) × s_2, where (r, s_2) = -(b, c, s)
Anything else	→	(⊤,⊤) × s

-: RIC × RIC → RIC

-(R1, R2) = ⊔ R1 ⊞ -a, where a ∈ R2

-: STACK_LOCATION × RIC × STATE → P(STACK_LOCATION) × STATE
-(s, R, state) = -*(s, R, state, ∅)

-*: STACK_LOCATION × RIC × STATE × P(STACK_LOCATION) → P(STACK_LOCATION) × STATE
-*(s, R, state, result) = let (s2, state2) = add-node(s, state) in
–*(s2, -(R,1), state2, (1∈R) → (result ∪ {s2}) ▯ result)
if some member of R is > 0
result × 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 with the stack-location that is at the top of the stack.

add-node: STACK_LOCATION × STATE → STACK_LOCATION × STATE
add-node(loc, state) = m × (state↓1, [m ↦ (⊤,⊤)]state↓2, (m, loc) ∪ state↓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 × VALUE → VALUE

INPUT		RESULT
(a, ⊤) × (c, ⊤)	→	(*(a,c), ⊤)
Anything else	→	(⊤,⊤)

*: RIC × RIC → RIC
*(R1, R2) = ⊔ R1 × r, where r ∈ 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 × VALUE → VALUE

INPUT		RESULT
Anything	→	(⊤,⊤)

4.2 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: STACK_LOCATION × STATE → VALUE
load(location, state) = state↓2(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: STACK_LOCATION × VALUE × STATE → STATE
store(loc, value, state) = (state↓1, [loc ↦ value]state↓2, state↓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: STATE → P(VALUE)
top(state) = ⊔ state↓2(m), where m ∈ state↓1(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 × STATE → STATE
push(value, state) = ([esp ↦ m]state↓1, [m ↦ value]state↓2, [∪ (m, n)] ∪ state↓3)
where n ∈ (state↓1(esp)) ↓2

pop: REGISTER × STATE → STATE
pop(reg, state)=
([reg ↦ top(state), esp ↦ ⊔ succ(1, n, state↓3)]state↓1, state↓2, state↓3)
where n ∈ state↓2(esp)

4.3 Miscellaneous Operations

The following functions 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 ⊥ (the bottom of the stack).

reset: STATE → STATE

reset(state) = ([esp ↦ (⊤, {⊥})]state↓1, state↓2, state↓3)

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

make-value: ℕ → VALUE

make-value(c) = (0×[0,0]+c, ⊤)

5 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$\downarrow$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$\downarrow$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$\downarrow$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 = $\cup$ $\mathcal{E}$ (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))

6 Examples

The following sections contain examples demonstrating the analysis of various obfuscation techniques. Section 6.1 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.

6.1 Using Push/Jmp

Fig. 4 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 ouput 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. Code such as this can cause problems during CFG generation and thus may cause analysis methods that rely on them to behave unpredictably.

Upon entry, all registers are initialized to $\top$, 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 (Fig. 5a).

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 (Fig. 5b). 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 (Fig. 5c).

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 L6 in this case. L6 is a mov instruction that moves the value located at *esp*+4 into register *eax* (Fig. 5d).

Instruction L7 performs in a manner similar to L6. Instruction L8 has no effect on the state.

Instruction L9 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.

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

Fig. 4. Obfuscated call using push/jmp

Instruction L10 copies the value from *ebx* to *eax* (Fig. 5e).

The ret 8 instruction at L11 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 L11 can be reached from L9 and L10, the results of evaluating the two paths must be joined before processing L11. 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 L9, *eax* is 2, whereas at instruction L10, *eax* is 4. The union of the two is the set {2, 4}, or the RIC 2[1,2]+0 (Fig. 5f).

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

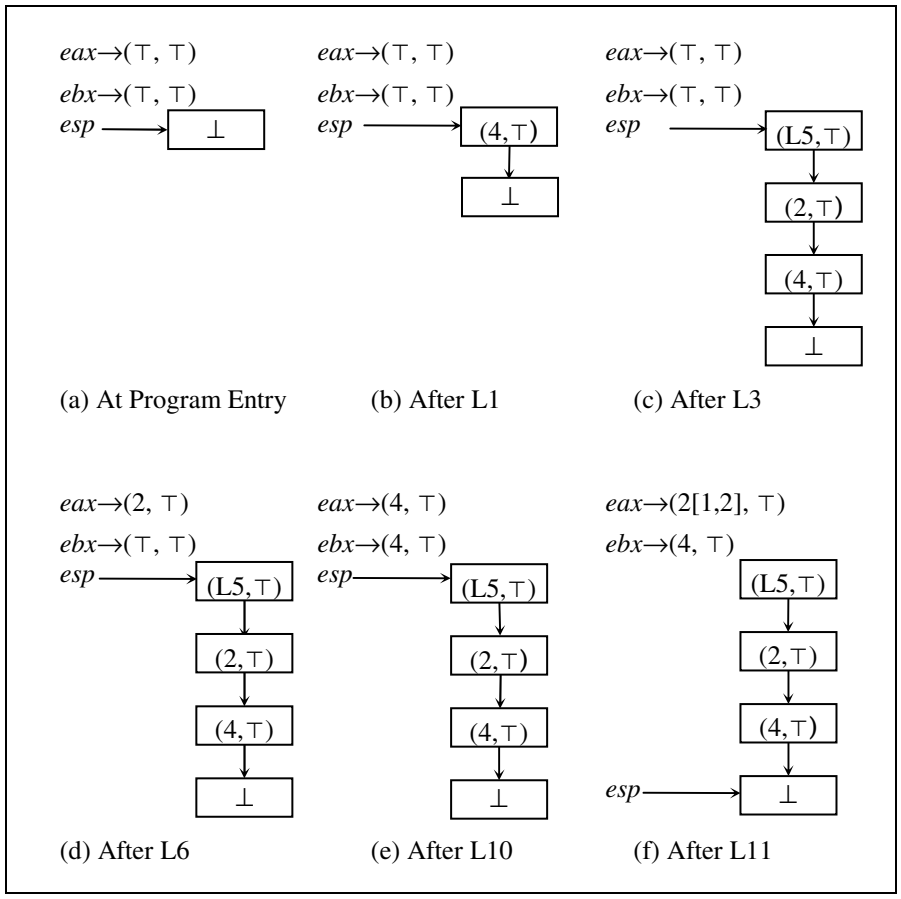

Fig. 5. Contents of the state at various points in the example program (see Fig. 4)

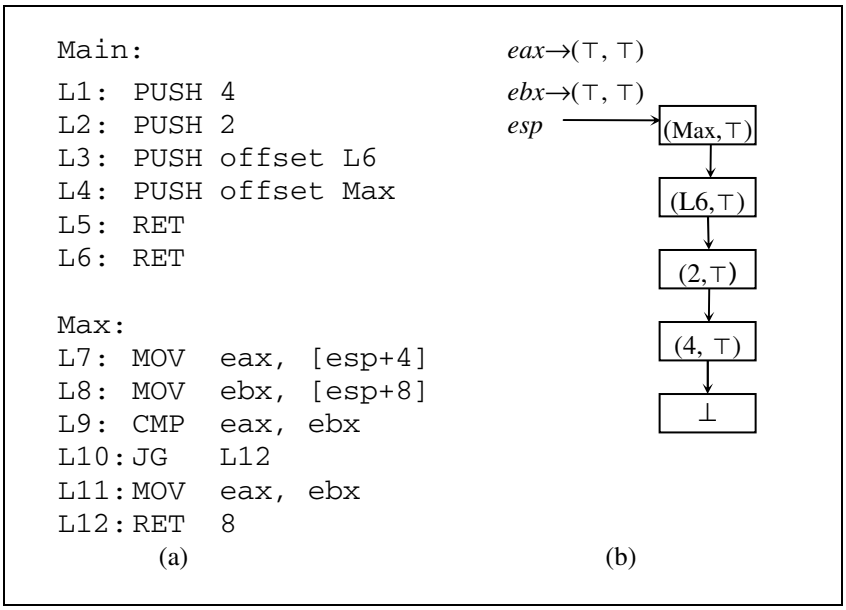

Fig. 6. (a) Call obfuscation using push/ret (b) State at instruction L5

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.

6.2 Using Push/Ret

Fig. 6a 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. L6 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.

During the analysis, at instruction L5, there are four nodes in the abstract stack, as shown in Fig. 6b. 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, ⊤). Thus, L6 becomes the return address of the Max procedure.

6.3 Using Pop to Return

In Fig. 7a, 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 L10-L12).

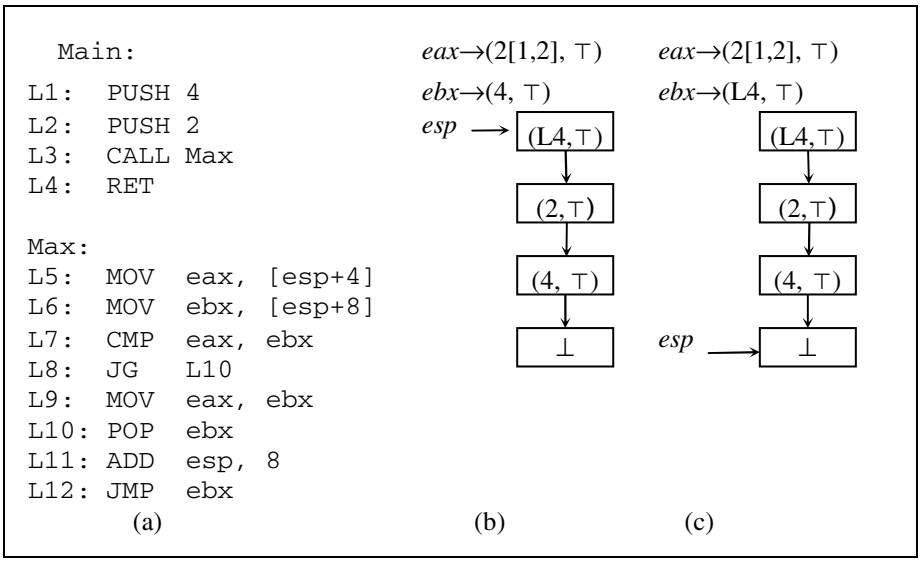

Fig. 7. (a) Obfuscation using pop to return. (b) State at L10. (c)State at L12

At instruction L10, the stack contains four nodes, as shown in Fig. 7b. L10 removes the value from the top of the stack and places it in *ebx*. L11 adds eight to *esp*, which causes *esp* to point to the bottom of the stack. L12 is an indirect jump to the address in *ebx*. Looking at the stack at instruction L12 (Fig. 7c), *ebx* contains (L4, ⊤), thus analysis continues at instruction L4, the original return address.

6.4 Modifying Return Address

In Fig. 8a, 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.

At instruction L10, the top of the stack originally contains (Max, ⊤). L10 removes this value from the stack and L11 pushes the value (L30, ⊤) onto the stack. Fig. 8b shows the resulting state. The ret statement at L12 causes analysis to continue at instruction L30.

7 Future Work

Currently, this work approximates only the values stored in registers or on the stack. No effort is taken to determine the values that may be stored at any arbitrary location on the heap. Future work will involve extending the architecture to handle this additional task and the ability to handle other kinds of obfuscations. We will also construct a prototype for testing how well the proposed solution performs at detecting metamorphic viruses with call obfuscations.

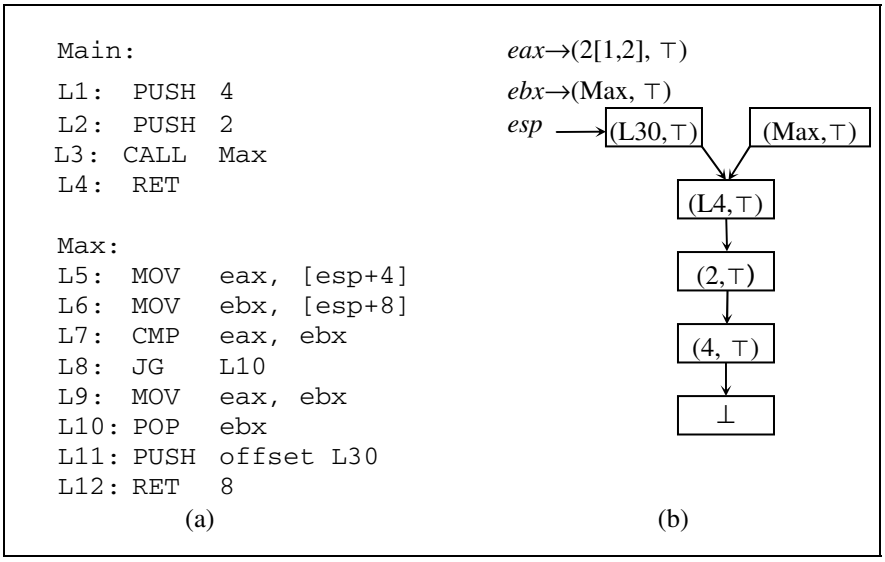

Fig. 8. (a) Obfuscation by modifying return address (b) State at instruction L12

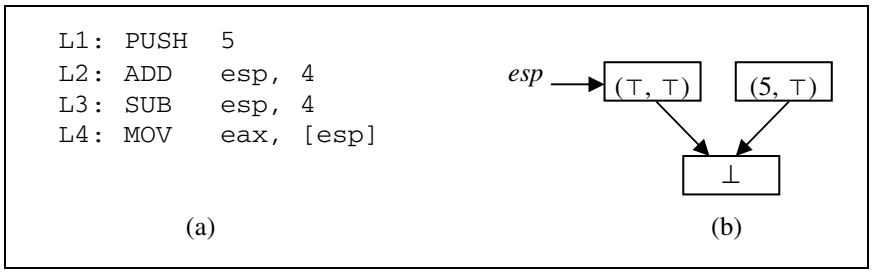

Fig. 9. Manipulation of the abstract stack graph

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 Fig. 9a 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 Fig. 9b. 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.

8 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 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 for 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 in the presence of 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 paper can be applied to help uncover obfuscated calls and provide for a more reliable analysis.

Acknowledgements

We are grateful to Eric Uday Kumar for his very helpful discussions and contribution to this paper.

References

1. C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in 10th ACM Conference on Computer and Communications Security (CCS), 2003.
2. C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Department of Computer Science, University of Auckland, 1997.
3. 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.
4. P. Ször and P. Ferrie, "Hunting for Metamorphic," in Virus Bulletin Conference, Prague, Czech Republic, 2001.

5. A. Lakhotia and P. K. Singh, "Challenges in Getting 'Formal' with Viruses," Virus Bulletin, 2003
6. P. Ször, "The New 32-Bit Medusa," Virus Bulletin, pp. 8-10, 2000.
7. 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.
8. Symantec, "Understanding Heuristics: Symantec's Bloodhound Technology," http://www.symantec.com/avcenter/reference/heuristc.pdf, Last accessed July 1, 2004.
9. G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in X86 Executables," in 13th International Conference on Compiler Construction, 2004.
10. P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs," in 2nd Int. Symp. on Programming, Dumod, Paris, France, 1976.