# Abstracting Stack to Detect Obfuscated Calls in Binaries

Arun Lakhotia and Eric Uday Kumar
*Center for Advanced Computer Studies, University of Louisiana at Lafayette, LA*
*{arun, euk4141}@cacs.louisiana.edu*

## Abstract

*Information about calls to the operating system (or kernel libraries) made by a binary executable may be used to determine whether the binary is malicious. Being aware of this approach, malicious programmers hide this information by making such calls without using the call instruction. For instance, the 'call addr' instruction may be replaced by two push instructions and a return instruction, the first push pushes the address of the instruction after the return instruction, and the second push pushes the address addr. The code may be further obfuscated by spreading the three instructions and by splitting each instruction into multiple instructions. This paper presents a method to statically detect obfuscated calls in binary code. The notion of abstract stack is introduced to associate each element in the stack to the instruction that pushes the element. An abstract stack graph is a concise representation of all abstract stacks at every point in the program. An abstract stack graph, created by abstract interpretation of the binary executables, may be used to detect obfuscated calls and other stack related obfuscations.*

## 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 [8, 13, 19]. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners [6, 14, 17].

The primary goal of obfuscation is to increase the effort involved in manually or automatically analyzing a program. In the context of anti-virus scanning, the context of our study, automated analysis may be performed at the desktop, at quarantine servers in an enterprise, or on back-end machines of an anti-virus company's laboratory [16]. In contrast, manual analysis is performed by engineers in Emergency Response Teams of anti-virus companies and research laboratories. The goal of obfuscation in malicious programs—viruses, worms, Trojans, spy wares, backdoors—is to escape detection by automated analysis and significantly delay detection by manual analysis.

A common obfuscation technique that is found in viruses, henceforth used generically to mean malicious programs, is that they obfuscate *call* instructions [17]. For instance, a *call addr* instruction may be replaced by two *push* instructions and a *ret* instruction, the first *push* pushing the address of the instruction after the *ret* instruction, the second *push* pushing the address *addr*. The code may be further obfuscated by spreading the three instructions and by further splitting each instruction into multiple instructions.

Obfuscation of *call* instructions breaks most static analysis based methods for detecting a virus 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 it makes. Such analysis can easily be thwarted by what is termed as "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 may not terminate without user input. Virus writers can bypass stopping heuristics by introducing a delay loop that simply wastes cycles. It is therefore important to detect obfuscated calls both for static and dynamic analysis of viruses.

This paper presents a method to statically detect obfuscated calls when the obfuscation is performed by

This work was supported in part by funds from Louisiana Governor's Information Technology Initiative.

using other stack (-related) instructions, such as *push* and *pop*, *ret*, or instructions that can statically be mapped to such stack operations. The method uses abstract interpretation [10] wherein the stack instructions are interpreted to operate on an abstract stack. Instead of keeping actual data elements, an abstract stack keeps the address of the instruction that pushes an element on the stack. The potentially infinite set of abstract stacks resulting from all possible executions of a program is concisely represented in an abstract stack graph.

Our method may be used to improve manual and automated analysis tools, thereby raising the level of difficulty for a virus writer. Our method can help by removing some common obfuscation techniques from the toolkit of a virus writer. However, we do not claim that the method can deobfuscate *all* stack related obfuscations. Indeed, writing a program that detects *all* obfuscations is not achievable for the general problem maps to detecting program equivalence, which is undecidable.

The method we present is a partial solution. It addresses only the evaluation of operations that can be mapped to stack *push* and *pop* instructions, where each is performed as a unit operation. It does not model situations where the *push* and *pop* instructions themselves may be decomposed into multiple instructions, such as one to move the stack pointer and one to move data in/out of the stack. Further, our solution does not model other memory areas, the content of the stack, and the content of registers. This deficiency may be overcome by combining our stack model with the Balakrishnan and Reps' method for analyzing the content of memory locations [3].

Section 2 presents related work in this area. Section 3 presents the notion of an abstract stack and the abstract stack graph. Section 4 presents our algorithm to construct the abstract stack graph. Section 5 describes how the abstract stack graph may be used to detect various obfuscations. Section 6 outlines future work to develop a complete solution for detecting obfuscations. Section 7 concludes this paper.

## 2. Background and Related Work

To extract meaningful information from a binary it is first disassembled, i.e., translated to assembly instructions [5, 11, 13, 15]. The disassembled code is usually analyzed further, often following steps similar to those performed for decompilation [7]. Vinciguerra et al. have compiled a survey of disassembly and decompilation techniques [18]. Lakhotia and Singh [12] discuss how a virus writer could attack the various stages in the decompilation of binaries by taking advantage of the limitations of static analyses. Indeed, Linn et al. [13]present code obfuscation techniques for disrupting the disassembly phase, making it difficult for static analysis to even get started.

The art of obfuscation is very advanced. Collberg et al. [9] present "a taxonomy of obfuscating transformations" and a detailed theoretical description of such transformations. There exist obfuscation engines that may be linked to a program to create a *metamorphic* virus, a virus that creates a transformed copy of itself before propagation. The transformations are such that they change the byte sequence of the executable but do not disrupt the functionality of the program. Two such engines are Mistfall (by z0mbie), which is a library for binary obfuscation [2], and Burneye (by TESO), which is a Linux binary encapsulation tool [1].

Metamorphic viruses are particularly insidious because two copies of the virus do not have the same signature. Hence, they escape signature based anti-virus scanners [6]. Such viruses can sometimes be detected if the operating system calls made by the program can be determined. For example Symantec's Bloodhound technology uses classification algorithms to compare the set against a database of calls made by known viruses and clean programs [16].

The challenge, however, is in detecting the operating system calls made by a program. The PE and ELF format for binaries include mechanisms to inform the linker about the libraries used by a program. But there is no requirement that this information be included in the file headers. In Windows, the entry point address of various system functions may be computed by a program at runtime using a Kernel32 function called *GetProcAddress*. Win32.Evol worm uses precisely this method for getting addresses of kernel functions and then further obfuscates the method it uses to call these functions.

There is hope, however. A recent result by Barak et al. [4] proves that in general program obfuscation is impossible, i.e., there are certain program properties that cannot be obfuscated. This is likely to have an effect on the pace at which new metamorphic transformations are introduced. Lakhotia and Singh [12] observe that, although metamorphic viruses pose a serious challenge to anti-virus technologies, the virus writers too are confronted with the same theoretical limits and have to address some of the same challenges that the anti-virus technologies face.

Indeed research results in detecting obfuscated viruses are beginning to emerge. Christodrescu and Jha use abstract patterns to detect malicious patterns in executables [6]. Mohammed has developed a technique

to undo certain obfuscation transformations, such as statement reordering, variable renaming, and expression reshaping [14].

## 3. Abstract Stack and Abstract Stack Graph

An *abstract stack* is an abstraction of the *actual stack*. The actual stack of a program keeps actual data values that are pushed and popped in a LIFO (Last In First Out) sequence. The abstract stack instead stores keeps the addresses of the instructions that push and pop values in a LIFO sequence. For example, consider Figure 1. Each instruction in the sample program is marked with its address from *L1* through *L4*. The actual stack and the abstract stack, after execution of the instruction at address *L4*, are as shown in Figure 1. Initially the addresses *L1* and *L2* are pushed onto the abstract stack, but due to the *pop* instruction at *L3*, the address *L2* is popped and next *L4* is pushed.

| *Sample Program* | *Actual Stack* | *Abstract Stack* |
|---|---|---|
| *L1: push eax* | … | … |
| *L2: push ebx* | ... | … |
| *L3: pop edx* | *eax* | *L1* |
| *L4: push ecx* | *ecx* | *L4* |
| | *Top of stack* | *Top of stack* |

**Figure 1. Actual and Abstract Stacks**

Figure 2 gives an example that highlights some issues in creating abstract stacks for each point in the program. Figure 2a shows a sample program, its control flow graph appears in Figure 2b. Each block in the control flow graph represents a program point. The program points are numbered. Figure 2c shows a few abstract stacks that are possible at four program points. For instance, the 3rd abstract stack at program point 2 is the result of the following execution trace: 1 → 2 → 3 → 4 → 3 → 4 → 5 → 2. The abstract stack shown at program point 4 results from the trace 1 → 2 → 3 → 4 → 3 → 4 → 5 → 2 → 3 → 4. The execution trace 1 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 3 → 5 → 2 → 7 → 8 yields the abstract stack at program point 8.

Our interest is in finding all possible abstract stacks at each program point for all execution traces. Since there may be multiple execution traces from the entry node to any program point, there may be multiple abstract stacks at each program point. This is enumerated in the example by the multiple traces for program points 2 and 6 in Figure 2c. In fact, program points 3 and 4 may have infinite number of abstract stacks. This is because there is a loop between program points 3 and 4 and the loop contains unbalanced *push*, i.e., a *push* that is not matched with a *pop*.

An abstract stack graph is a concise representation of all, potentially infinite number of, abstract stacks at all points in the program. Figure 2d shows the abstract stack graph for the example program in Figure 2a.

More formally an abstract stack graph is defined as a directed graph represented by the 3-tuple (*N*, *AE*, *ASPR*). Let *ADDR* denote the domain of addresses.

$N \subseteq ADDR$ is the set of nodes. An address $n$ is in the set $N$ implies the instruction at address $n$ performs a push operation. In Figure 2, the nodes are shown in rectangular boxes.

$AE \subseteq ADDR \times ADDR$ is the set of edges. An edge $(n \rightarrow m) \in AE$, shown in the diagram by an arrow from node $n$ to node $m$, implies that there is an execution trace in which the instruction at address $n$ pushes a value on top of a value pushed by the instruction at address $m$.

$ASPR \subseteq ADDR \times ADDR$ captures the set of abstract stack pointers (stack tops) for each statement. A pair $(x, n) \in ASPR$ means that program point $x$ receives the abstract stack resulting from the value pushed by instruction $n$ at the top. In the diagram, this relation is shown by annotating each node $n$ with the address $x$ in circle, such that $(x, n) \in AE$. This relation may be read as: $n$ is the top of stack at program point $x$. It is also stated as: the top of stack $n$ is associated with the program point $x$.

A path in the abstract stack graph beginning at some stack top, say *asp*, and ending at the entry point *E* represents an abstract stack at all the program points associated with *asp*. A path in an ASG is represented as $n_1 | n_2 | n_3 | .. | n_j$ such that $(n_i \rightarrow n_{i+1}) \in AE$. This path represents an abstract stack with the last-in element, $n_1$, to the left and the first-in element to the right. This abstract stack reaches every program point associated with $n_1$.

In Figure 2 the number of each block in the CFG—and not the address of instructions in the block—are

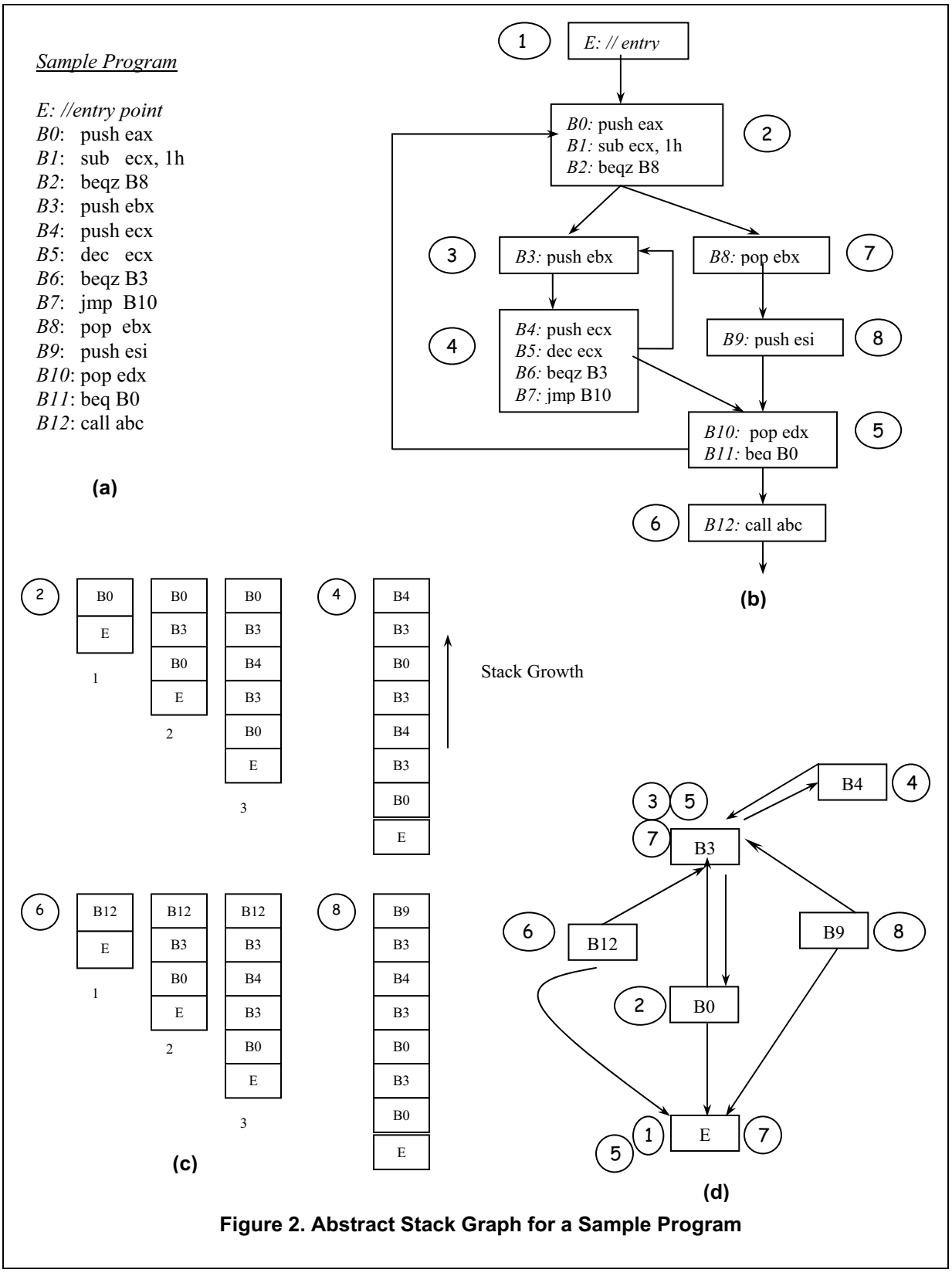


**Figure 2. Abstract Stack Graph for a Sample Program**

$\mathcal{E}$: $INST \rightarrow ASG \rightarrow ASG$

$\mathcal{E}$ **[** m: *push* **]** *asg* =
$\mathcal{E}$ *next*(m) ( *abspush* m *asg* )

$\mathcal{E}$ **[** m: *call addr* **]** *asg* =
$\mathcal{E}$ *inst*(*addr*) ( *abspush* m *asg* )

$\mathcal{E}$ **[** m: *ret* **]** *asg* =

$\bigcup_{n \in absret\ asg}$ $\mathcal{E}$ *n* ( *abspop* m *asg* )

$\mathcal{E}$ **[** m: *pop* **]** *asg* =
$\mathcal{E}$ *next*(m) ( *abspop* m *asg* )

$\mathcal{E}$ **[** m: *jnz addr* **]** *asg* =
($\mathcal{E}$ *inst*(*addr*) *asg* ) $\cup$ ($\mathcal{E}$ *next*(m) *asg*)

$\mathcal{E}$ **[** m: *jmp addr* **]** *asg* =
$\mathcal{E}$ *inst*(*addr*) ( *i asg* )

$\mathcal{E}$ **[** m: *mov esp x* **]** *asg* =
$\mathcal{E}$ *next*(m) ( *reset* m *asg* )

**Figure 3a. Evaluation Function**

used to annotate the CFG nodes. In the example chosen an instruction performing the push operation is always the first instruction in the block, and a block contains either an instruction that performs a push operation or an instruction that performs a pop operation, but not both. Thus, for the example of Figure 2, all points in a block receive the same top of stack. In Figure 2d, *B3* is an abstract node which is the address of the instruction *push ebx* and is associated with the set of program points $P = \{3, 5, 7\}$. Program points in $P$ receive abstract stacks with top *B3*. Two possible abstract stacks, when traversed from *asp = B3* are, B3|B0|E and B3|B4|B3|B0|E.

## 4. Constructing an abstract stack graph

Figure 3a presents the evaluation function $\mathcal{E}$ for constructing an abstract stack graph. The domain *INST* is the abstract syntax domain, representing the set of instructions. Each instruction is annotated with its address in the program. Thus, **[** m: *call addr* **]** is the instruction '*call addr'* at address *m*. The abstract domain *ASG* represents the domain of abstract syntax graphs. An element of *ASG* is a three-tuple (*N*, *AE*, *ASP*), where *N* and *AE* have the same meaning as in the definition of abstract syntax graph. However, the set *ASP* is not the same as *ASPR*.

$ASP \subseteq ADDR$ is the set of stack tops. *ASP* is a projection of *ASPR*. Loosely speaking, *ASP* and *ASPR*

*abspush*: $ADDR \rightarrow ASG \rightarrow ASG$
*abspush* m ( *N*, *AE*, *ASP* )
= ( $N \cup \{ m \}$,
$AE \cup \{ m \rightarrow asp \mid asp \in ASP \}$,
$\{ m \}$
)
*abspop*: $ASG \rightarrow ASG$
*abspop* m ( *N*, *AE*, *ASP* )
= ( *N*,
*AE*,
$\{ x \mid a \in ASP, (a \rightarrow x) \in AE \}$
)
*absret:* $ASG \rightarrow \wp ADDR$
*absret* ( *N*, *AE*, *ASP* )
= {*next*(*x*) | $a \in ASP$, $(a \rightarrow x) \in AE$, *validcall*(*x*)}
*reset*: $ADDR \rightarrow ASG \rightarrow ASG$
*reset* m (*N*, *AE*, *ASP* )
= ( $N \cup \{ m \}$,
*AE*,
$\{ m \}$
)
*i*: $ASG \rightarrow ASG$
*i* ( *N*, *AE*, *ASP* ) = ( *N*, *AE*, *ASP* )

**Figure 3b. Abstract Operations**

are related as follows: Let $\mathcal{E}$ **[** m: *inst* **]** *asg* = (*N*, *AE*, *ASP*), then $(m, a) \in ASPR$ where $a \in ASP$.

Figure 3b gives the abstract operations used to define the evaluation function. The operations are *abspush*, *abspop*, *absret*, *reset*, and *i* that operate on the domain *ASG*. The evaluation function and the abstract operations depend on the following primitive operators:

*next*: $ADDR \rightarrow ADDR$, gives the address of the next instruction.

*inst*: $ADDR \rightarrow INST$, gives the instruction at an address.

*validcall*: $ADDR \rightarrow$ Boolean, checks whether the instruction at a given address is a *call* instruction.

Operation *abspush* pushes a new address on the abstract stack. It is used in the evaluation of the *call* and *push* instructions. These two instructions are representative of instructions that perform the push operation. Other instructions may be modeled similar to these instructions. For example, the *INT* (software interrupt) instruction may be modeled like the *call* instruction. Instructions that increase the content of stack by directly manipulating the stack pointer, such as *sub esp*, 8*h*, are modeled using the push instruction.

Operation *abspop* pops an element from the abstract stack resulting in a new set of top of stacks.

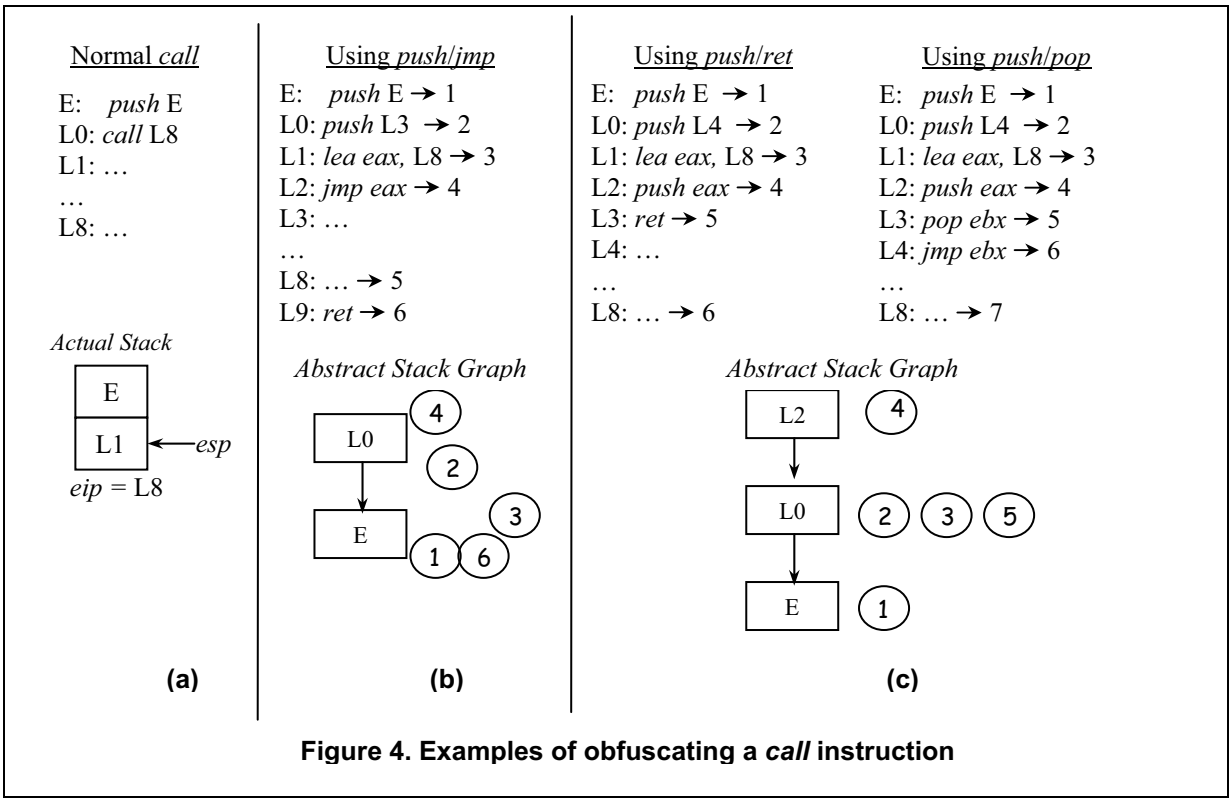


**Figure 4. Examples of obfuscating a *call* instruction**

The operator is used in the evaluation of *ret* and *pop* instructions.

Operation *absret* supports the evaluation of the *ret* instruction. It checks whether the address at the top of the stack represents the address of a *call* instruction. If so, it returns the address of instruction after the *call*. Since the abstract stack does not maintain the actual return address, the address to return to when a call is made by obfuscation is not known. This function identifies such obfuscations.

Operation *reset* is for all those instructions, such as *mov esp, eax*, that explicitly modify the stack pointer with a value not known to the analysis. Instructions such as *add esp, 8h* and *sub esp, 8h* whose effect on the stack pointer is known may be modeled as *pop* and *push* respectively.

Operation *i* is the identity operator. It is used for evaluation of any operation that does not modify the stack.

The abstract stack graph of a program, or a section of code, may be computed by applying the evaluation function to the entry address of the program on an initial abstract stack graph <Ø, Ø, Ø>. The evaluation continues until a fixed point is reached.

As demonstrated using the example in Figure 2 the algorithm generates the correct abstract stack graph even if a program contains loops with unbalanced *push* or *pop* instructions.

## 5. Detecting Obfuscations

We now discuss how an abstract stack graph may be used to detect stack related obfuscations. The obfuscations we study are:

- Obfuscation of call
- Obfuscation of parameters
- Obfuscation of return

These obfuscations include other abuses of stack, such as for obfuscating *jump* instruction by using a *call* that does not return or using *push* and *return* for achieving a *jump*.

Figures 4, 5, and 6 present example programs enumerating the three types of obfuscation. The figures show the actual stack at a program point of interest, and the abstract stack graph at that point. Each instruction in the example is annotated with a label, such as E, L0, L1, etc. A label represents the address of the instruction to which it is attached. The instructions are also annotated with an "→" followed by a number, such as "→ 4". The number is the symbolic program point associated to the instruction. The number is an alias for a label. We are using two different symbols to simplify the discussion.

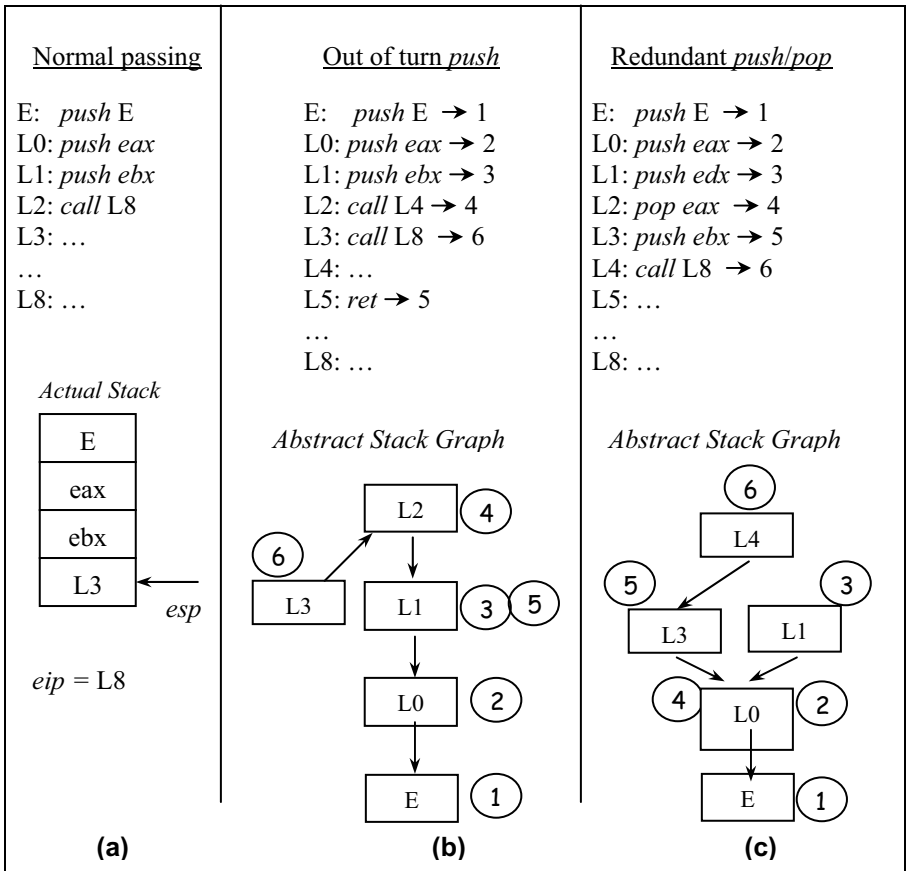


**Figure 5. Examples of obfuscating parameters to calls**

In the examples discussed below each program point of interest receives a single abstract stack. Hence, the discussion focuses on the specific stack. This should not be construed to imply that the methods require that the program points of interest receive a single abstract stack. The method discussed may simply be applied to every abstract stack received by a program point.

### 5.1. Detecting obfuscated calls

The semantics of a *call addr* instruction may be defined operationally as follows:

1. Push the address of the next instruction on the stack
2. Assign the address *addr* to the instruction pointer (*eip*).

Figure 4 contains several examples of obfuscated calls. Each example achieves the semantics of a *call* using a different sequence of instructions.

Figure 4(b) shows a program that performs the semantics of a *call* using a combination of *push* and *jmp* instructions. That the *jmp* instruction actually performs a *call* becomes known from the abstract stack graph at the entry point of the call. When an address is not known to be the entry point of a procedure, the abstract stack graph at the *ret* instruction, program point 6, discloses the obfuscation. During normal execution the top of the stack at this program point contains the return address *L3* pushed by the *push* instruction at label *L0*. In the abstract stack graph, the top of stack at program point *6* contains is *E*. The top of stack when evaluating the *ret* instruction is *E*, which is not the address of a *call* instruction. This indicates that the *ret* instruction is returning to an obfuscated call.

Figure 4(c) shows two obfuscations of *call*. The two differ in how control is transferred to the target address. In the first, the target address is pushed on the stack and a *ret* instruction pops this address from the stack and transfers control. In the second, the target address is pushed on the stack, it is then popped into a

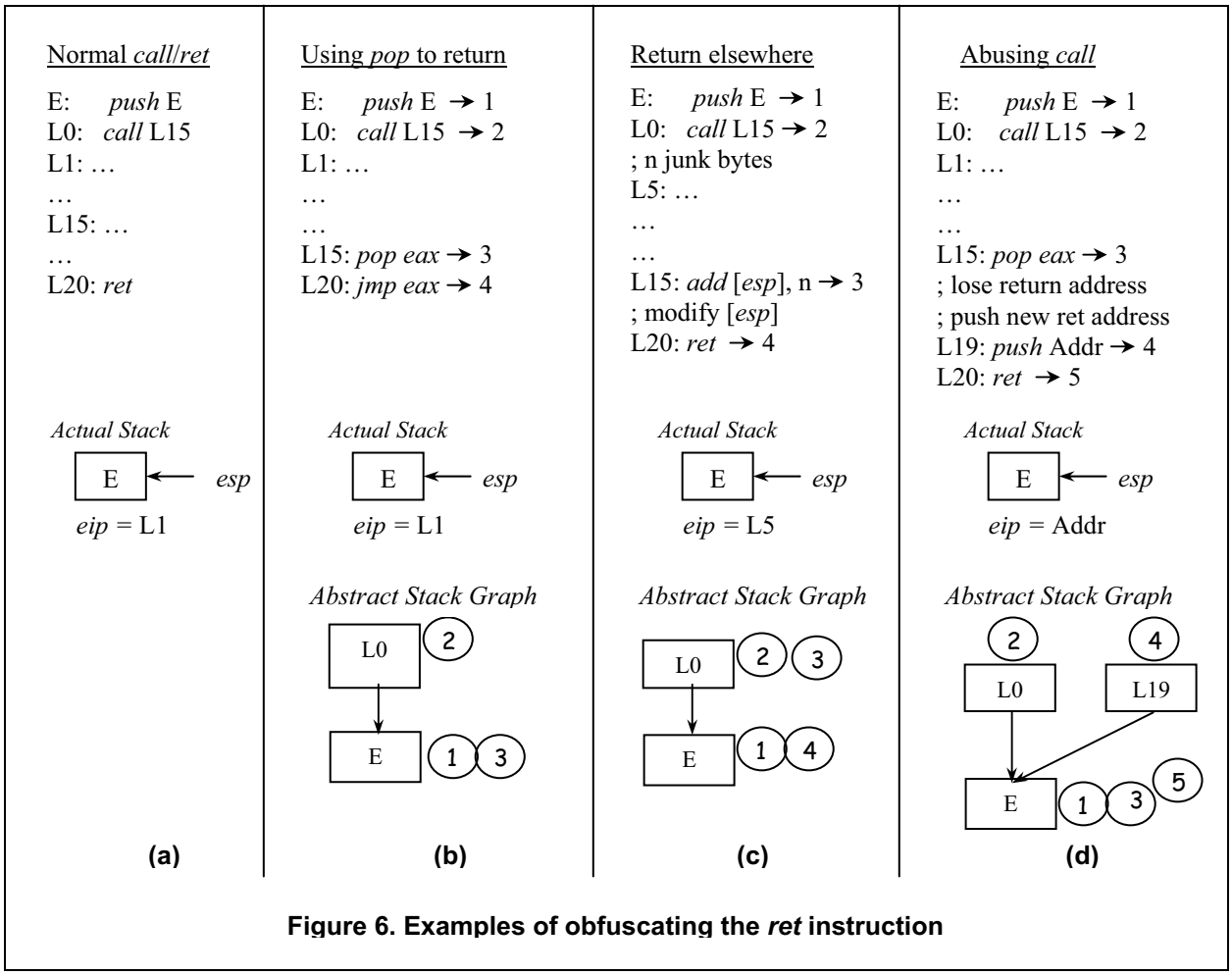


**Figure 6. Examples of obfuscating the *ret* instruction**

register, and an indirect jump is performed to the address in the register. The labels and program points in the two examples have been chosen such that both examples have the same stack and abstract stack graph. In both examples the actual transfer of control is done at instruction labeled *L3*. In the first example this is a *ret* instruction and in the second example it is *pop ebx*. These instructions are designated as program point 5. The top of the abstract stack at program point 5 contains *L0*, the address of the instruction that pushed the target address on the stack. Thus, once again when a *ret* statement is encountered it can be determined that it was reached due to an obfuscated call.

### 5.2. Detecting obfuscated parameters

When analyzing a program for malicious behavior it is often useful to know the parameters being passed to a function. A program may be deemed malicious depending on the parameter. For instance, opening a file for read may be considered acceptable, but opening for write may indicate malicious intent.

Parameters to a function are most often passed on the stack or in registers. Abstract stack graph can aid in determining the parameters that are passed on the stack. If a *call* takes *n* instructions, the top *n* elements on the abstract stacks at a program point before the *call* instruction represent the locations where those parameters were pushed. The $i^{th}$ parameter corresponds to the $i^{th}$ element on the stack (assuming the first parameter is pushed last). If the last parameter is pushed first, we change it around. At the entry point, the parameters are determined after compensating for the return address.

Figure 5 presents example programs that obfuscate where parameters are pushed. Figure 5(a) contains a sample normal code. In this program, the arguments to the function are being pushed immediately before the *call* instruction. Figure 5(b) contains an example of out-of-turn push. In this program instructions at *L0* and *L1* push the parameters in registers *eax* and *ebx* onto

the stack. These are parameters intended to be parameters to *call L8*, but they are pushed before the instruction *call L4*. This gives the incorrect appearance that the parameters are being passed to the function at *L4*. Thus, a *push* instruction need not pass parameters to the first *call* instruction. The abstract stack graph for the program can be used to detect where the parameters to a function are assembled. At program point 6, immediately after a *call L8*, the state of the abstract stack is L3|L2|L1|L0|E. The top of stack, *L3*, represents the return address. The two elements on the abstract stack, *L1* and *L0*, represent the location where the parameters for the function are pushed.

This example is also instructive on how abstract stack graph may be used to match *call* and *ret* instructions. At program point 5, the *ret* instruction, the top of the abstract stack contains *L2*. Thus, the *ret* instruction will return from call made by the *call* instruction at address label *L2*.

Introducing redundant push and pop instructions may also obfuscate parameters. Consider the program in Figure 5(c). The value pushed at instruction *L1* is popped at *L2*. They are thus redundant. The abstract stack at program point 5, before the *call* instruction is L3|L0|E, indicating that the parameters to the call are pushed at L3 and L0. The effect of the redundant push and pop instructions is visible at prior statements, but not at program point 5.

### *5.3.* Detecting obfuscated *ret*

A *ret* statement typically pops the top of the stack and returns control to the address it pops. The return instruction may be obfuscated by using different instructions for the same task or by having it transfer control to a location other than the instruction after the *call* instruction.

Figure 6 presents some examples of obfuscating the *ret* instruction. In the example of Figure 6(b) the effect of a *ret* instruction is achieved by popping the address into a register and jumping to that address. The abstract stack immediately before the address is popped is L0|E. Thus, it can be determined that the *pop* instruction is popping the return address from the call at *L0*, thereby indicating that the *ret* address is obfuscated.

The instruction *add* [*esp*], *n* in Figure 6(c) modifies the return address on the stack so that the *ret* instruction transfers control to *n* bytes after the original return address. This is obfuscating *ret* to return elsewhere. The abstract stack graph may be augmented to detect this obfuscation. Along with each location in the stack an additional tag, *modified*, may be maintained. When a value is pushed on the stack, *modified* is set to *false*. If an instruction may change the contents of the stack, and we can determine the stack offset that is being changed, then we can change the tag of that location to *modified*. If the value at the top of the stack at a *ret* instruction is modified, it implies that *ret* is returning elsewhere.

The *call* instruction can also be abused to actually jump to a particular instruction. In Figure 6(d), a *call* is made to *L15* at instruction *L0*. Inside this function, the return address is popped off the stack. A new return address is computed and pushed onto the stack (instruction at *L19*). The instruction at *L20* transfers control to the new address location. The abstract stack graph shown here can be used to detect such abuse. At program point 4, immediately before the *ret* instruction the stack is L19|E. This indicates that the *ret* instruction is obfuscated, since it will transfer control to the address pushed by a *push* instruction, and not after a *call*.

## 6. Future Work

The concept of abstract stack graph, introduced above, and the method for constructing abstract stack graphs are partial solutions for detecting obfuscations in binaries. A complete solution will also include evaluation of other instructions, a method of modeling actual memory locations, and model for content of memory locations and content of registers.

Balakrishnan and Reps [3] have proposed a method for abstract interpretation of non-stack related instructions. Their effort is aimed at discovering "something similar to C variables" from analyzing the memory accesses of a binary executable. Since their analysis assumes that a program conforms to a 'standard compilation model,' their model of stack is static. An activation record is associated with each procedure. The stack is a set of activation records that are linked together during interprocedural analysis.

Adapting Balakrishan and Reps' algorithm to use an abstract stack graph may help create a complete system for detecting obfuscations. The adaptation may also help create a disassembler for obfuscated programs that cannot be fooled easily.

## 7. Conclusions

A method for modeling the stack for static analysis of assembly programs has been presented. The set of all possible stacks due to all possible executions of a program is represented as an abstract stack graph. The graph is a 3-tuple, with nodes, edges, and annotation

on nodes. Each instruction that pushes a value on the stack is represented as a node in the graph. An edge represents a push operation, from an instruction pushing a value to an instruction that pushed the value on the top of the stack. A path in the graph represents a specific abstract stack. A node is annotated with the statements that receive an abstract stack with that node at the top.

An abstract stack graph may be used to support disassembly of obfuscated code and to detect obfuscations related to stack operations. Examples of how to use the abstract stack graph to detect few obfuscated calls, returns and parameters were presented. The construction of an abstract stack graph offers a step towards analyzing obfuscated binaries for malicious behavior.

## 8. References


[1] "Teso, Burneye Elf Encryption Program," https://teso.scene.at, Last accessed July 1, 2004.

[2] "Z0mbie," http://z0mbie.host.sk, Last accessed July 1, 2004.

[3] G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in X86 Executables," in *International Conference on Compiler Construction (CC) 2004*, Barcelona, Spain, 2004.

[4] B. Barak, et al., "On the (Im)Possibility of Obfuscating Programs," in *Advances in Cryptology (CRYPTO'01)*, Santa Barbara, California, USA, 2001.

[5] S. Cho, "Win32 Disassembler," http://www.geocities.com/~sangcho/disasm.html, Last accessed July 1, 2004.

[6] M. Christodrescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *The 12th USENIX Security Symposium (Security '03)*, Washington DC, USA, 2003.

[7] C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," *Software Practice and Experience*, vol. 25, pp. 811 - 829, 1995.

[8] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," *IEEE Transactions on Software Engineering*, vol. 28, pp. 735-746, 2002.

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

[10] N. D. Jones and F. Nielson, "Abstract Interpretation: A Semantics-Based Tool for Program Analysis," in *Handbook of Logic in Computer Science: Semantic Modelling*, vol. 4, S. Abramsky, et al., Eds. Oxford, UK: Oxford University Press, 1995, pp. 527-636.

[11] C. Kruegel, et al., "Static Disassembly of Obfuscated Binaries," in *USENIX Security 2004*, San Diego, 2004.

[12] A. Lakhotia and P. K. Singh, "Challenges in Getting Formal with Viruses," *Virus Bulletin*, 2003, http://www.virusbtn.com/magazine/archives/200309/formal.xml.

[13] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communication Security 2003*, Washington D.C., USA, 2003.

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

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

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

[17] P. Szor and P. Ferrie, "Hunting for Metamorphic," in *Virus Bulletin Conference*, Prague, 2001.

[18] 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.

[19] G. Wroblewski, *General Method of Program Code Obfuscation*, Institute of Engineering Cybernetics, Wroclaw University of Technology, PhD. Thesis, 2002.