A Method for Detecting Obfuscated Calls in Malicious Binaries

Arun Lakhotia, Eric Uday Kumar, and Michael Venable

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 *ret* instruction, the first *push* pushes the address of instruction after the *ret* 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 work presents a method to statically detect obfuscated calls in binary code. The idea is to use abstract interpretation to detect where the normal *call-ret* convention is violated. These violations can be detected by what is called an abstract stack graph. An abstract stack graph is a concise representation of all potential abstract stacks at every point in a program. An abstract stack is used to associate each element in the stack to the instruction that pushes the element. An algorithm for constructing the abstract stack graph is also presented. Methods for using the abstract stack graph are shown to detect eight different obfuscations. The technique is demonstrated by implementing a prototype tool called DOC (Detector for Obfuscated Calls).

Index Terms—Invasive software (viruses, worms), program analysis, validation, obfuscation, abstract stack.

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) [11], [19], [27]. Programs may also be obfuscated to hide malicious behavior and to evade detection by antivirus scanners [8], [17], [24].

The primary goal of obfuscation is to increase the effort involved in manually or automatically analyzing a program. In the context of antivirus 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 antivirus company's laboratory [23]. In contrast, manual analysis is performed by engineers in Emergency Response Teams of antivirus companies and research laboratories. The goal of obfuscation in malicious programs—viruses, worms, trojans, spywares, backdoors—is to escape detection by automated analysis and significantly delay detection by manual analysis.

One of the first steps in determining whether a program is malicious is to identify the system calls it makes. If the program performs certain collections of file operations, registry operations, or network operations, it may be considered potentially malicious. The set (sometimes, the sequence) of system calls a program makes is referred to as its behavior. The behavior of a program may be determined by either static analysis or by dynamic analysis. In static analysis, a program is analyzed (by humans and/or tools) without running or simulating it. In dynamic analysis, a program's behavior is observed, often by trapping the system calls or sniffing network activity.

Malware writers have developed obfuscation techniques that make it difficult to statically identify the calls made by their programs. These programs effectively make a *call* without actually using the call instruction [24]. For instance, the *call addr* instruction may be replaced by two *push* instructions and a *ret* instruction, the first *push* pushing the address of 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 1) identify the kernel functions used by a program and 2) to delineate code into procedures. 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. While dynamic analysis is helpful and often necessary, they are often cumbersome, time-consuming, and fallible. Malware authors already know many methods for circumventing detection by a dynamic analyzer, including detecting the dynamic analysis method, introducing delay loops to bypass stopping heuristics, and executing the malicious behavior in only particular circumstances. Therefore, static analysis is a necessary component of antivirus (AV) analysis.

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

- *A. Lakhotia and M. Venable are with the Center for Advanced Computer Studies, University of Louisiana at Lafayette, PO Box 44330, Lafayette, LA 70504-4330. E-mail: {arun, mpv7292}@cacs.louisiana.edu.*
- *E.U. Kumar is with Authentium, Inc., 1020 Cypress Way E, Palm Springs, FL 33406. E-mail: ekumar@authentium.com.*

Manuscript received 17 May 2005; revised 27 Aug. 2005; accepted 21 Sept. 2005; published online 1 Dec. 2005.
Recommended for acceptance by A. Ruben.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0155-0505.

using other stack (-related) instructions, such as *push*, *pop*, *ret*, or instructions that can statically be mapped to such stack operations. The method uses abstract interpretation [14] 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 infinite set of abstract stacks resulting from all possible executions of a program, via, static analysis, is concisely represented by an abstract stack graph. Obfuscated calls are detected by analyzing the 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 detect 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 [10].

The method presented 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. Ongoing work in this area by Venable et al. [25] aims to overcome these deficiencies 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 describes implementation of a prototype tool and results on applying it on a virus called W32.Evol. Section 7 outlines future work to develop a complete solution for detecting obfuscations and as well concludes this paper.

2 Background and Related Work

Most of the significant problems related to the detection of viruses are undecidable [10], [5]. As a result, AV technologies utilize heuristics and tricks designed to catch specific viruses [20]. The process of providing a solution to identify a virus consists of two parts. In the first part, a virus sample is analysed in the laboratory of an AV company. This analysis, which is often performed manually, leads to identifying special characteristics of the virus which may be used to identify the virus. In the second part, these special characteristics are then encoded in a "signature" and transmitted to the AV engine running on a user's desktop or an enterprise server. A signature is a unique sequence of bytes that identifies the virus. The AV engine uses the signature to identify the specific virus.

To extract meaningful information from a binary it is first disassembled, i.e., translated to assembly instructions [6], [15], [22]. The disassembled code is usually analyzed further, often following steps similar to those performed for decompilation [9]. Vinciguerra et al. have compiled a survey of disassembly and decompilation techniques [26]. Lakhotia and Singh [18] discuss how a virus writer could attack the various stages in the decompilation of binaries by taking advantage of the limitation of static analysis. Indeed, Linn et al. [19] 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. [12] 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 AV scanners [8]. Such viruses can sometimes be detected if the operating system calls made by the program can be determined. For example, Symantec's Bloodhound technology executes a program in a sandbox (or an emulator), traps the calls made by the program, and then determines whether it is malicious by using classification algorithms to compare the set against a database of calls made by known viruses and clean programs [23]. The challenge, however, is in detecting the operating system calls made by a program. The PE and ELF format for binaries include mechanism 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*. W32.Evol virus uses precisely this method for getting addresses of kernel functions and 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 [18] observe that, though metamorphic viruses pose a serious challenge to antivirus technologies, the virus writers too are confronted with the same theoretical limits and have to address some of the same challenges that the antivirus technologies face.

Indeed research results in detecting obfuscated viruses are beginning to emerge. Christodorescu and Jha use abstract patterns to detect malicious patterns in executables [8]. Lakhotia and Mohammed have developed a technique to undo certain obfuscation transformations, such as

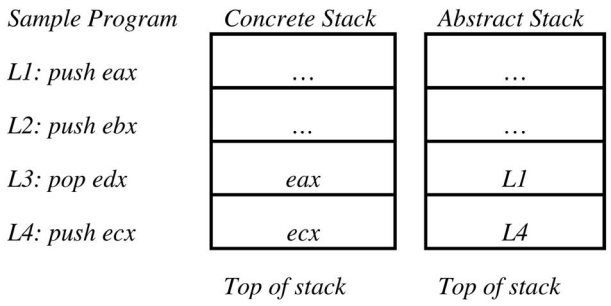


Fig. 1. Concrete and abstract stacks.

statement reordering, variable renaming, and expression reshaping [17]. In a more recent work, Christodorescu et al. have developed a semantic directed method to find variants of a virus [7]. Their method, which requires creating templates of code fragments to be searched, works well for variants created by hand-modifying or recompiling a virus. Lakhotia et al. also find similar virus variants by cluster analysis of the code using maximal-pi patterns [16]. This approach is completely automated and, unlike Christodorescu's approach, does not require templates.

3 The Abstract Stack

An *abstract stack* is an abstraction of the *concrete stack*. The concrete 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 the addresses of the instructions that push and pop values in a LIFO sequence. For example, consider Fig. 1. Each instruction in the sample program is marked with its address from *L1* through *L4*. The concrete stack and the abstract stack, after execution of the instruction at address *L4*, are as shown in Fig. 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.

The following example highlights some issues in creating abstract stacks for each point in the program. Fig. 2 shows a

```
E:  //entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

Fig. 2. Sample program.

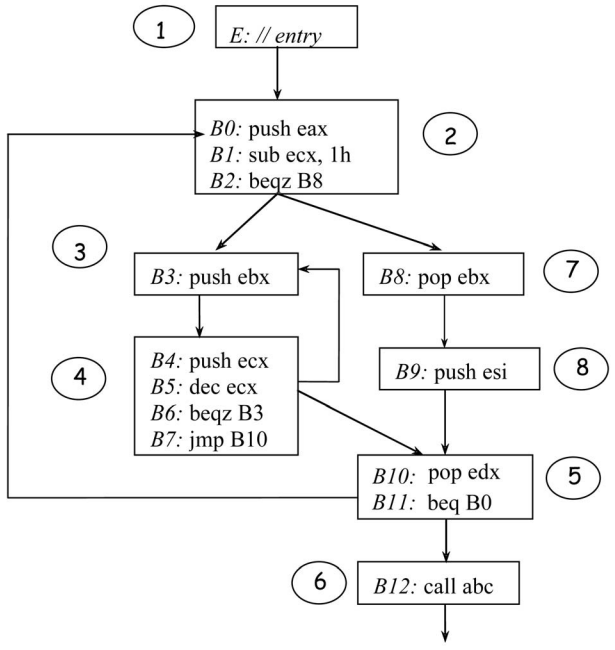


Fig. 3. Control flow graph for sample program.

sample program; its control flow graph appears in Fig. 3. Each block in the control flow graph may contain only a single *push*, *pop*, or *call* instruction or may additionally contain a control transfer instruction. The program points are numbered. Fig. 4 shows a few abstract stacks that are possible at four program points. For instance, the third abstract stack at program point 2 is the result of the following execution trace:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2.$$

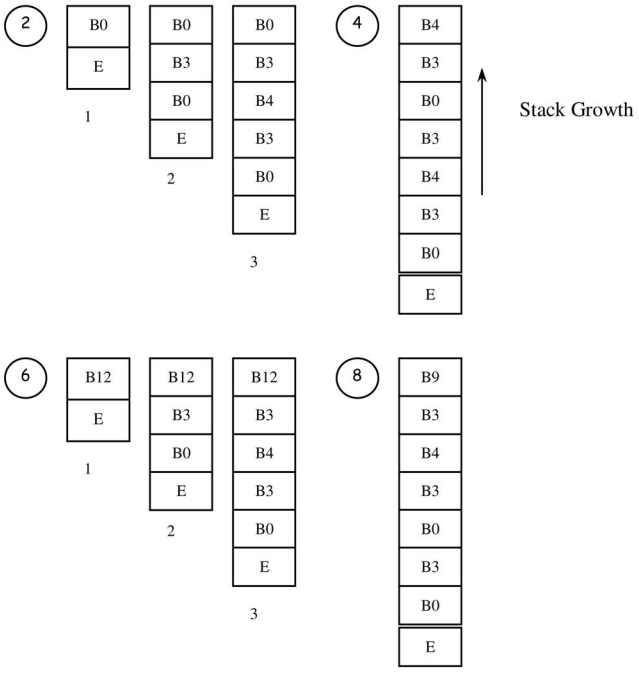


Fig. 4. Possible abstract stacks at some program points.

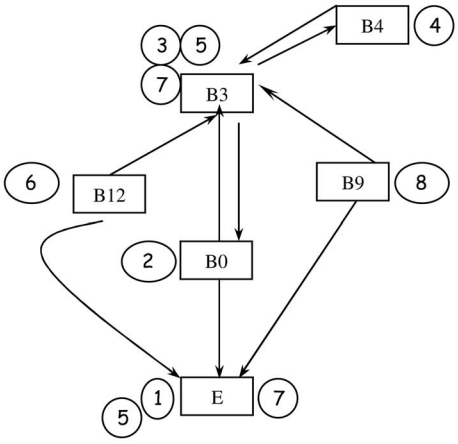


Fig. 5. Abstract stack graph for sample program.

The abstract stack shown at program point 4 results from the trace

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4.$$

The execution trace

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 7 \rightarrow 8$$

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 Fig. 4. In fact, program points 3 and 4 may have infinite number of abstract stacks because of loops. A more efficient way to handle all possible abstract stacks at each program point is required.

3.1 The Abstract Stack Graph

An abstract stack graph is a concise representation of all, potentially an infinite number of, abstract stacks at all points in the program. Fig. 5 shows the abstract stack graph for the example program in Fig. 2. A path (sequence of nodes beginning from the abstract stack top toward the bottom) in the graph represents a specific abstract stack.

3.2 Defining the Domain

Let $ADDR$ denote a set of addresses. An abstract stack graph is a directed graph represented by the 3-tuple $< N, AE, ASPR >$ defined as follows:

- $N \subseteq ADDR$ is a set of nodes. An address $n \in N$ implies the instruction at address n performs a push operation. Our convention is to show nodes as rectangular boxes in diagrams.
- $AE \subseteq ADDR \times ADDR$ is a set of edges. An edge $< n, m > \in AE$ denotes that there is possible execution trace in which the instruction at address n may push 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. We show this diagrammatically 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.

The domain $INST$ is the abstract syntax domain, representing the set of instructions. Each instruction is annotated with its address in the program. Thus, $[\mathrm{m} : \mathit{call\ addr}]$ is the abstract interpretation of the concrete instruction "*call addr*" at address m.

The domain ASG is the domain of abstract stack 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 stack 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 $ASPR$.

A path in ASG beginning at some stack top, say t, and ending at the entry point E is associated with every abstract stack that can occur at the program points associated with t. A path p in ASG is represented as $n_1|n_2|n_3|..|n_j$ such that $< n_i \rightarrow n_{i+1} > \in AE$. p is mapped by a function Ψ to an abstract stack with the last-in element n_1 and the first-in element n_j.

To be concise in Fig. 3, the number of each block in the CFG and not the address of instructions in the block are used to annotate the CFG nodes. Here, 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, in Fig. 3, all points in a block receive the same top of stack. In Fig. 5, $B3$ 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 $B3$, i.e., the abstract stack pointer $asp = B3$. 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

Constructing an abstract stack graph involves defining an evaluation function that provides the interpretation of each assembly instruction in abstract terms. A set of abstract operations over the ASG domain needs to be defined first. The following sections explain the evaluation function built from these abstract operations.

4.1 Evaluation Function

Fig. 6 presents an evaluation function $\mathcal{E}_0$ for constructing an abstract stack graph. It is defined piecewise as a set of rewrite rules or equations. The evaluation function and the abstract operations depend on the following primitive operators:

- *next*: $ADDR \rightarrow ADDR$, returns the address of the next instruction to be interpreted.
- *inst*: $ADDR \rightarrow INST$, returns the instruction at a given address.

$\mathcal{E}_0$: *INST* → *ASG* → *ASG*

$\mathcal{E}_0$ [m: *push*] *asg* =

$\mathcal{E}_0$ *next*(m) (*abspush* m *asg*)

$\mathcal{E}_0$ [m: *call addr*] *asg* =

$\mathcal{E}_0$ *inst*(*addr*) (*abspush* m *asg*)

$\mathcal{E}_0$ [m: *ret*] *asg* =

$\cup$ $\mathcal{E}_0$ *n* (*abspop* m *asg*)

n ∈ *absret* m *asg*

$\mathcal{E}_0$ [m: *pop*] *asg* =

$\mathcal{E}_0$ *next*(m) (*abspop* m *asg*)

$\mathcal{E}_0$ [m: <*jmp_conditional*> *addr*] *asg* =

($\mathcal{E}_0$ *inst*(*addr*) *asg*)∪($\mathcal{E}_0$ *next*(m) *asg*)

$\mathcal{E}_0$ [m: *jmp addr*] *asg* =

$\mathcal{E}_0$ *inst*(*addr*) (*i asg*)

$\mathcal{E}_0$ [m: <*mov* | *add* | *sub*> *esp x*] *asg* =

$\mathcal{E}_0$ *next*(m) (*reset* m *asg*)

$\mathcal{E}_0$ [m: <*jmp_conditional* | *jmp* | *call*> <*reg* | *mem*>] *asg* =

$\mathcal{E}_0$ *next*(m) (*i asg*)

Fig. 6. Evaluation function.

abspush: *ADDR* → *ASG* → *ASG*

abspush m (*N*, *AE*, *ASP*)

= (*N* ∪ m,

AE ∪ { m → asp | asp ∈ *ASP* },

{ m })

abspop: *ADDR* → *ASG* → *ASG*

abspop m (*N*, *AE*, *ASP*)

= (*N*, *AE*, { x | a ∈ *ASP*, (a ➔ x) ∈ *AE* })

absret: *ADDR* → *ASG* → ℘*ADDR*

absret m (*N*, *AE*, *ASP*)

= {*next*(a) | a ∈ *ASP*, *is_call*(a)}

∪ {*next*(m) | a ∈ *ASP*, *is_push*(a)}

reset: *ADDR* → *ASG* → *ASG*

reset m (*N*, *AE*, *ASP*)

= (*N* ∪ { m },

AE,

{ m })

i: *ASG* → *ASG*

i (*N*, *AE*, *ASP*) = (*N*, *AE*, *ASP*)

Fig. 7. Abstract operations.

The evaluation function $\mathcal{E}_0$ takes in two parameters of type $INST$ and ASG and outputs an element of ASG. This is denoted by $\mathcal{E}_0$: $INST \rightarrow ASG \rightarrow ASG$. For example, $\mathcal{E}_0[\ \mathrm{m} : inst\ ]\ asg = (N, AE, ASP)$, denotes the evaluation of the instruction $inst \in INST$ with address $m \in ADDR$ being the execution address and $asg \in ASG$ being the execution context.

Now, we can, loosely speaking, say that ASP and $ASPR$ are related as follows: Let $\mathcal{E}_0[\ \mathrm{m} : inst\ ]\ asg = (N, AE, ASP)$, then $(m, a) \in ASPR$, where $a \in ASP$. The evaluation function determines the next instruction to be interpreted using the primitive operators and also determines the appropriate abstract operation used to interpret the current instruction. The next section defines these abstract operations.

4.2 Abstract Operations

Fig. 7 defines the effects of the abstract operations. Note that the operations and evaluation function are recursively defined in terms of each other. The operations are *abspush*, *abspop*, *absret*, *reset*, and i that operate on the domain ASG.

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 modelled like the *call* instruction. Instructions that increase the content of stack by directly manipulating the stack pointer, such as *sub esp, 8h*, are modeled using the push instruction.

Operation *abspop* pops an element from the abstract stack resulting in a new set of top of stack. The operator is used in the evaluation of *ret* and *pop* instructions.

Operation *absret* supports the evaluation of the *ret* instruction. It uses the function $is\_call()$ to check whether the address at the top of 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 actual return address, the address to return to when a *call* is made by obfuscation is not known. This operation identifies such obfuscations using the $is\_push()$ function. Details of detecting obfuscations are explained in subsequent sections.

Operation *reset* is for all those instructions that explicitly modify the stack pointer with value not known to the analysis. For example instructions such as *move esp, eax*. 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.

4.3 Algorithm

The naïve algorithm constructs an abstract stack graph of a section of code by applying the evaluation function to the entry address of the program on an initial abstract stack graph $< \emptyset, \emptyset, \emptyset >$ and then continuing until a termination condition is reached. The termination condition may be due to reaching some specific memory address, or reaching an invalid instruction, or when an empty stack is popped. Details of the termination condition of the evaluation function are not shown in Fig. 6. A sketch of the algorithm follows:Assume that the disassembled code and an entry point in the code are available. The current abstract stack graph is initialized to $< \emptyset, \emptyset, \emptyset >$. The assembly instructions are then interpreted one by one using $\mathcal{E}_0$. A work list W is maintained such that each element in W is a tuple $< ip, asp, succ >$. Here, ip (instruction pointer) is the address of the next instruction to be executed, asp (abstract stack pointer) is the address of an instruction denoting top of the abstract stack graph, $succ$ is the number of successor abstract nodes of asp. Initially, W is the singleton set $\{< Entry, 0, 0 >\}$.

A visited list V is also maintained which keeps track of the instructions previously interpreted for a given state of the abstract stack graph. This is necessary to avoid getting trapped in a loop because of a backward control transfer or jump. The visited list V maintains a list of already interpreted work list elements for a given state of the abstract stack graph. Each $w \in W$ carries the abstract stack graphs' state information in $succ$. This is important because, whenever a conditional branch instruction is encountered from within a loop, information about the updated state of the abstract stack graph has to pass along the two possible branch paths. This is accomplished by including $succ$ in the tuple for w. The pseudocode for this is as shown below:

```
W = {< E → nextInst, E, 0 >} // Initial work list
V = {∅} // Initial visited list is empty
while (W ≠ NULL) {
    for w ∈ W, retrieve w from W
    if (w ∉ V) {
    add w to V;
    List = abstract_interpret(w);
    W = W ∪ List; }
}
```

Here, w is the tuple <ip, asp, succ>. The function $abstract\_interpret(w)$ interprets the instruction specified by ip according to the evaluation function $\mathcal{E}_0$, modifies the ASG accordingly and either returns null (if the ASG is not modified), or a list of work list elements.

The algorithm generates a correct abstract stack graph even for programs with loops with unbalanced *push* or *pop* instructions. This means that, if there are individual loops within which *push* or *pop* occur and within these loops the *push* or *pop* are not balanced (i.e., there are more *push* than *pop*, or more *pop* than *push*), the algorithm can still generate the correct abstract stack graph that encompasses all the possible abstract stacks at each program point, including the stack representing the balancing of *push* and *pop* after the two loops.

Each node in the abstract stack graph is created only when a *push* or a *call* instruction is encountered. Since a program is finite, the abstract stack graph will have a finite set of nodes. This implies that the abstract stack graph is finite since each node in the graph may contain only a finite number of edges, no more than the number of nodes. This property ensures the state will reach a stable condition where further interpretation will no longer result in state modifications because only a limited number of edges may be added. Thus, we are assured termination. This also means that, in the worst case, each node may contain n edges, resulting in $\mathrm{O}(n^2)$ performance, where n is the number of instructions to be interpreted.

Since the ASG domain is a finite, powerset lattice, no specific widening operator [13] is needed for termination. If the $\mathrm{O}(n^2)$ computational cost is an issue, one could use a widening operator that rapidly converges to the top element. However, the resulting approximation will defy the purpose of the analysis.

4.4 Proof of Correctness

The previous section shows that the algorithm is guaranteed to terminate. This section proves the correctness of its computation. To prove that our abstract semantics is a sound approximation of the concrete semantics, it is sufficient to define the homomorphisms needed to form Galois connections between the abstract and concrete domains [21]. This involves first defining the concrete and abstract domains and then defining the homomorphisms.

The concrete domain consists of memory and a set of registers. Each location in memory contains a value, as does each register. Formally, this is defined as

$$\mathrm{CSTORE} = (\mathrm{ADDR} \rightarrow \mathrm{VALUE}) \times (\mathrm{REGISTER} \rightarrow \mathrm{VALUE}),$$

where REGISTER is the set of registers and VALUE is the set of possible values that can be placed in a register and memory. We define an extended semantics that subsumes the aforementioned concrete semantics. In the extended semantics, the memory locations and registers contain value-address pairs, as opposed to containing only values. This is represented by the extended store

$$\begin{aligned}\mathrm{ESTORE} = {} & (ADDR \rightarrow (VALUE \times ADDR)) \\ & \times (\mathrm{REGISTER} \rightarrow (\mathrm{VALUE} \times \mathrm{ADDR})).\end{aligned}$$

The address that is paired to the value is the address of the instruction responsible for creating that value. More specifically, if an instruction pushes a value onto the stack, the top of the stack will contain the value being pushed and the address of the push instruction.

The extended semantics preserves the concrete semantics because the values computed by the latter are also computed by the former. The extended semantics however also makes explicit the relation between a value and the instruction creating it, which then sets the stage for defining the homomorphism.

For the abstract semantics, we define the abstract store

$$\begin{aligned}\mathrm{ASTORE} = {} & (ADDR \rightarrow \wp(ADDR)) \\ & \times (\mathrm{REGISTER} \rightarrow \wp(\mathrm{ADDR})).\end{aligned}$$

The domain ASG defined in Section 3 maps to the abstract store ASTORE as follows: the map $\mathrm{ADDR} \rightarrow \wp(\mathrm{ADDR})$ of ASTORE is equivalent to the binary relation $\mathrm{ADDR} \times \mathrm{ADDR}$, the set of edges of ASG. The map $REGISTER \rightarrow \wp(ADDR)_\top$ is equivalent to $ASPR$, the third element of ASG, if it maps all registers, except the stack pointer (*esp*), to $\top$. Then, $ASPR$ is the value associated to register *esp*.

With the concrete and abstract domains defined, the final step is to define the homomorphisms between the abstract values and the concrete values. To reiterate, the concrete and abstract domains are, respectively,

$$\begin{aligned}\mathrm{ESTORE} &= (\mathrm{ADDR} \rightarrow (\mathrm{VALUE} \times \mathrm{ADDR})) \\ &\quad \times (\mathrm{REGISTER} \rightarrow (\mathrm{VALUE} \times \mathrm{ADDR})) \text{ and} \\ \mathrm{ASTORE} &= (\mathrm{ADDR} \rightarrow \wp(\mathrm{ADDR})) \\ &\quad \times (\mathrm{REGISTER} \rightarrow \wp(\mathrm{ADDR})).\end{aligned}$$

The only difference between the two is that the concrete domain contains $\mathrm{VALUE} \times \mathrm{ADDR}$, where the abstract domain contains $\wp(\mathrm{ADDR})$. Given a value-address pair from the concrete domain, one can easily find the corresponding set of addresses in the abstract domain. This operation can be expressed as

$$\begin{aligned}&\alpha : \wp(\mathrm{VALUE} \times \mathrm{ADDR}) \rightarrow \wp(\mathrm{ADDR}) \\ &\gamma : \wp(\mathrm{ADDR}) \rightarrow \wp(\mathrm{VALUE} \times \mathrm{ADDR}) \\ &\alpha(\mathrm{C}) = \{\mathrm{a} | (\mathrm{v}, \mathrm{a}) \in \mathrm{C}\} \\ &\gamma(\mathrm{A}) = \cup\{\mathrm{c} | \alpha(\mathrm{c}) \subseteq \mathrm{A}\},\end{aligned}$$

where α translates elements from the concrete domain to the abstract domain and γ translates from the abstract to the concrete.

5 Detecting Obfuscations

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

- Call obfuscation.
- Parameter passing obfuscation.
- Return obfuscation.

Example programs are used to illustrate the use of the abstract stack graph to detect these obfuscations. In these examples, each instruction is annotated with an address label, such as *E*, *L0*, *L1*, etc. The instructions are also annotated with an arrow followed by a number, such as "→ 4." The number is the symbolic program point associated with the instruction. The number is an alias for the instruction's label: The different symbols are used to simplify the discussion. In the examples, each program point of interest is associated with a single abstract stack. Hence, the discussion focuses on the specific stack. This should not be construed to imply that the methods are restricted to a single stack. Rather, the method discussed may be applied to every abstract stack associated with a program point. Throughout the following, obfuscation is detected when the contents of the abstract stack graph at control points is not what would be expected if the call was not obfuscated.

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*).

Fig. 8 contains several examples of obfuscated calls. Each example achieves the semantics of a *call* using a different sequence of instructions. From Fig. 8a, in case of normal *call-ret* the top of stack points to the return address *L1* on reaching address *L8*. It is evident from the abstract stack graph of Fig. 8a that when program point 3 is reached, the *ret* instruction is returning from a node *L1* which is the address of a *call* instruction. Hence, *L1* and *L3* constitute a valid *call-ret* site.

As shown in Fig. 8b, the *call* can be obfuscated by substituting it with three other instructions. The first instruction *push L3* pushes the return address, the second instruction *push L8* pushes the target address and the *ret* transfers control to address *L8* as well as pops it from the stack. Hence, on reaching *L8* the top of stack points to the return address *L3* which is equivalent in semantics to normal *call-ret*. The abstract stack graph in Fig. 8b can be used to detect this obfuscation. When program point 4 is reached, the *ret* instruction is returning from node *L1* which is the address of a *push* instruction and not a *call* instruction, hence detecting the obfuscation.

Fig. 8c shows a program that performs the semantics of a *call* using a combination of *push* and *jmp* instructions. The target address *L8* is loaded in register *eax* and a *jmp* indirect transfers control to *L8*. From the abstract stack graph in Fig. 8c, when program point 5 is reached, the *ret* instruction is returning from the node *L0* which is the address of a *push* instruction hence disclosing the obfuscation due to *push/jmp*.

Fig. 8d shows a program that performs the semantics of a *call* using a combination of *push* and *pop* instructions. The instruction *pop ebx* at program point 4 retrieves the target address *L8* from the stack and a *jmp* indirect transfers control to *L8*. The abstract stack graph in Fig. 8d can be used to detect this obfuscation. When program point 6 is reached, the *ret* instruction is returning from node *L0* which is the address of a *push* instruction and not a *call* instruction, hence detecting the obfuscation.

The fragment of evaluation function $\mathcal{E}_1$, created by modifying corresponding fragment of $\mathcal{E}_0$, captures the logic for detecting obfuscated *calls*.

$$\begin{aligned}&\mathcal{E}_1 : INST \rightarrow ASG \rightarrow ASG \times \wp(ADDR \times ADDR) \\ &\mathcal{E}_1[\mathrm{m}: ret](N, AE, ASP) = \\ &(\cup \{\mathcal{E}_1\ n\ (abspop\ \mathrm{m}\ (N, AE, ASP))\}) \\ &\cup (\emptyset, \{(\mathrm{a},\mathrm{m}) | \mathrm{a} \in ASP, is\_push(\mathrm{a})\}) \\ &\text{where } n \in absret\mathrm{m}\,(N, AE, ASP).\end{aligned}$$

The evaluation returns a 2-tuple consisting of the modified ASG and a set of pair of addresses. Each pair of addresses represent a procedure boundary, addresses of *call* and *ret* instructions. The union operator $\cup$ is overloaded to do pointwise union of tuples. Hence, a union with $\emptyset$ returns

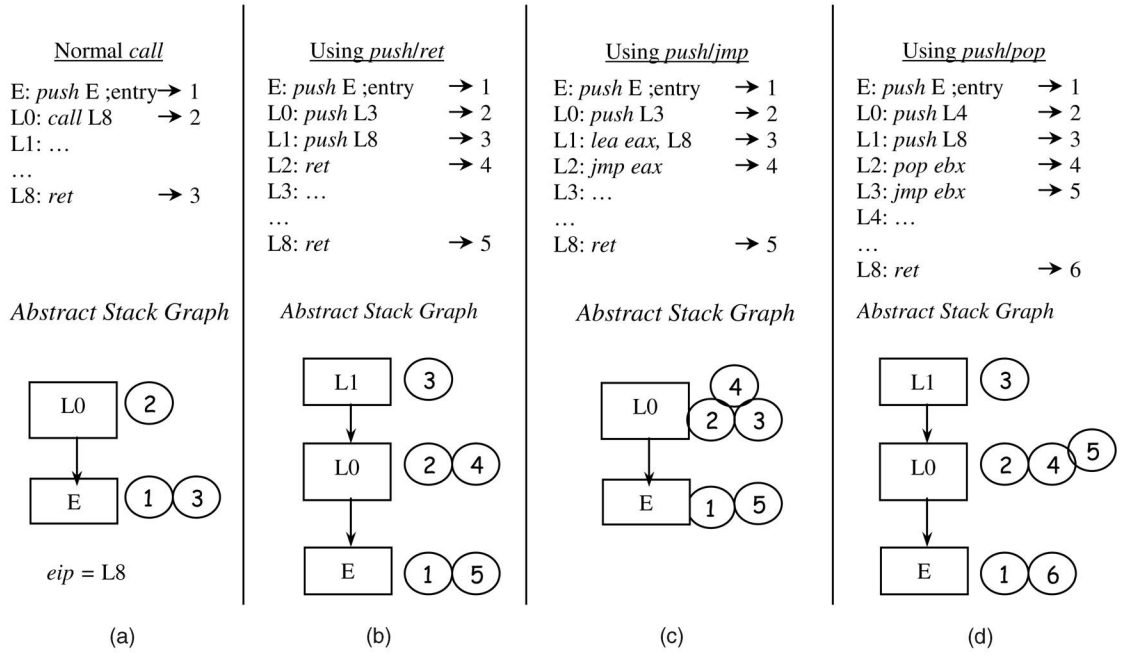


Fig. 8. Possible ways of obfuscating the *call* instruction.

the modified ASG. This fragment provides a modified evaluation of the *ret* instruction. The evaluation of other instructions remains the same, except for providing an empty set for the second element of the tuple.

Obfuscation of *call* is detected when the top of stack is identified as a *push* instruction. The function $is\_push()$ checks if the address at the top of the stack is the address of a *push* instruction. The address pair (a, m) is identified as an obfuscated *call*. Assuming this to be an obfuscation of a *system call*, the next instruction to be interpreted is from the set of addresses returned by the abstract operator *absret*.

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 parameters. For instance, calling a file-open with parameters set to read may be considered benign, but the same call with parameters set for writing may indicate malicious intent.

Parameters to a function are ordinarily passed via the stack or through registers. An 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 ith parameter corresponds to the ith element on the stack (starting from the top). This is assuming the first parameter is pushed last. If the last parameter is pushed first, the order is changed to match. At the entry point, the parameter addresses are connected by compensating for the pushed return address.

Fig. 9 presents example programs that obfuscate where parameters are pushed. Fig. 9a contains a sample normal code. In this program, the arguments to the function are pushed immediately before the *call* instruction.

Fig. 9b contains an example of out-of turn push. In this program, the 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 L5*, 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 5, immediately after a *call L6*, the state of the abstract stack is L3 | L1 | L0 | E. The top of stack, *L3*, represents the return address. The two elements on the abstract stack, *L1* and *L0*, represent the instructions that push the parameters for the function.

The abstract stack graph can also be used to detect noncontiguous procedures, that is, when all the instructions of a procedure are not in a contiguous sequence of memory. Such type of control transfers are usually absent in compiler generated code that adhere to conventional procedure entry and exit, but occur in malicious code or hand coded assembly. Noncontiguous procedures can be identified by computing call-return instruction pairs, and analyzing these pairs to determine whether 1) return address is before (less than) the address of a call instruction and 2) there is another entry point between the entry point of a procedure and any of its return instructions.

The evaluation function $\mathcal{E}_0$ may be modified to collect *call* and *ret* instruction pairs. The following fragment of function $\mathcal{E}_2$ summarizes the modification needed to the *ret* instruction:

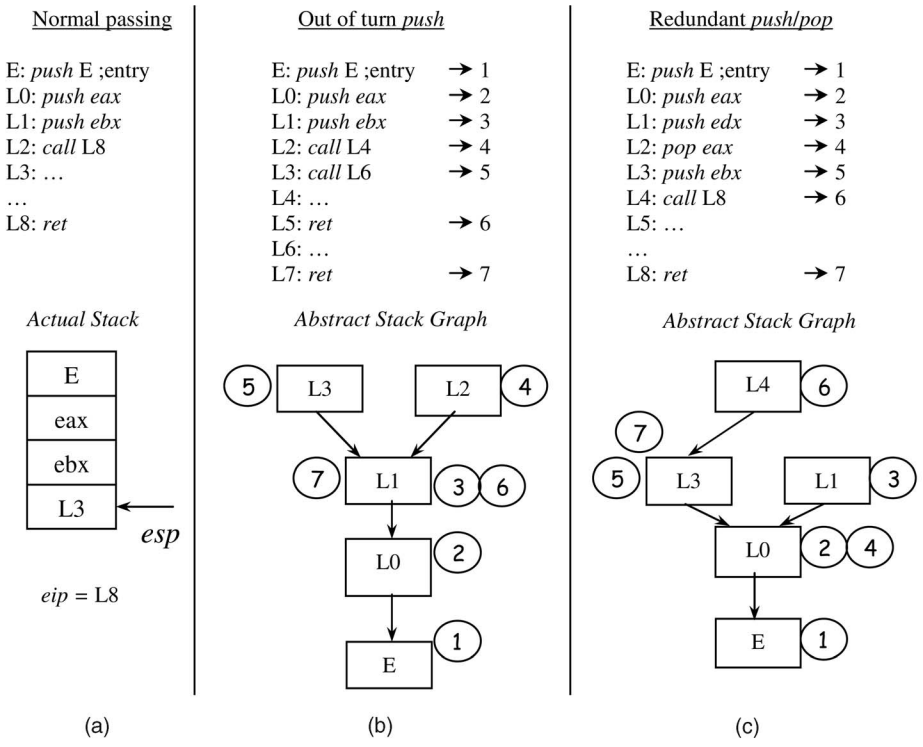


Fig. 9. Possible ways of obfuscating parameters to a *call*.

$$\mathcal{E}_2 : INST \rightarrow ASG \rightarrow ASG \times \wp(ADDR \times ADDR)$$
$$\mathcal{E}_2[\mathrm{m}\colon ret](N,AE,ASP) = (\cup\{\mathcal{E}_2\, n\, (abspop\, \mathrm{m}(N,AE,ASP))\})$$
$$\cup\, (\emptyset, \{(\mathrm{a},\mathrm{m}) | \mathrm{a} \in ASP, is\_call(\mathrm{a})\})$$
$$\text{where } \mathrm{n} \in absret\ \mathrm{m}(N, AE, ASP).$$

The addresses of *call* and *ret* instructions that form procedure boundaries are returned as the pair (a, m) when the top of stack is identified as a *call* instruction.

The example also shows how the abstract stack graph may be used to match *call* and *ret* instructions. At program point 4, where the *call* to *L4* is made, the abstract interpreter actually simulates a control transfer to the target of the call site to interpret the next instruction at *L4*. The abstract stack state passed is L2 | L1 | L0 | E with *L2* as the abstract stack top. At program point 6, which is a *ret* instruction; the top of the abstract stack contains *L2*. Thus, the *ret* instruction will be seen to return from a *call* made at address label *L2*. At program point 7, the abstract stack state is L3 | L1 | L0 | E and does not include *L2* since at a call site updated information is only passed down the taken branch. Hence, at program point 7, the *ret* instruction will be seen to return from the *call* made at address label *L3*.

Introducing redundant push and pop instructions can obfuscate the parameters. Consider the program in Fig. 9c. 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 Return

A *ret* statement typically pops the top of the stack and returns control to address it pops, essentially reverses a *call*. It pops the old *eip* value from the stack into *eip* and increments *esp* by a word. The conventional way of using *call* and *ret* is as shown in Fig. 10a. After *ret* is executed, control transfers to the instruction immediately after the *call*. The return may be obfuscated by simulating it using non-return instructions or by having it transfer control to a location other than the instruction after the *call* instruction.

Fig. 10 presents some examples of obfuscating the *ret* instruction. In the example of Fig. 10b, 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 *ret* instruction can also be obfuscated by returning elsewhere. Instead of the conventional way of returning to the instruction immediately following the *call* instruction, the return address is modified in the called function and control transferred to some other instruction. In Fig. 10c, the instruction at *L0* makes the *call* to *L3*. Immediately after the *call* instruction, two junk bytes are inserted to locate a specific return address (*L3* in this case). The instruction at *L4*, the contents of the stack pointer, are modified by adding two bytes to the return address to generate a new return address so that the *ret* instruction transfers control to two 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

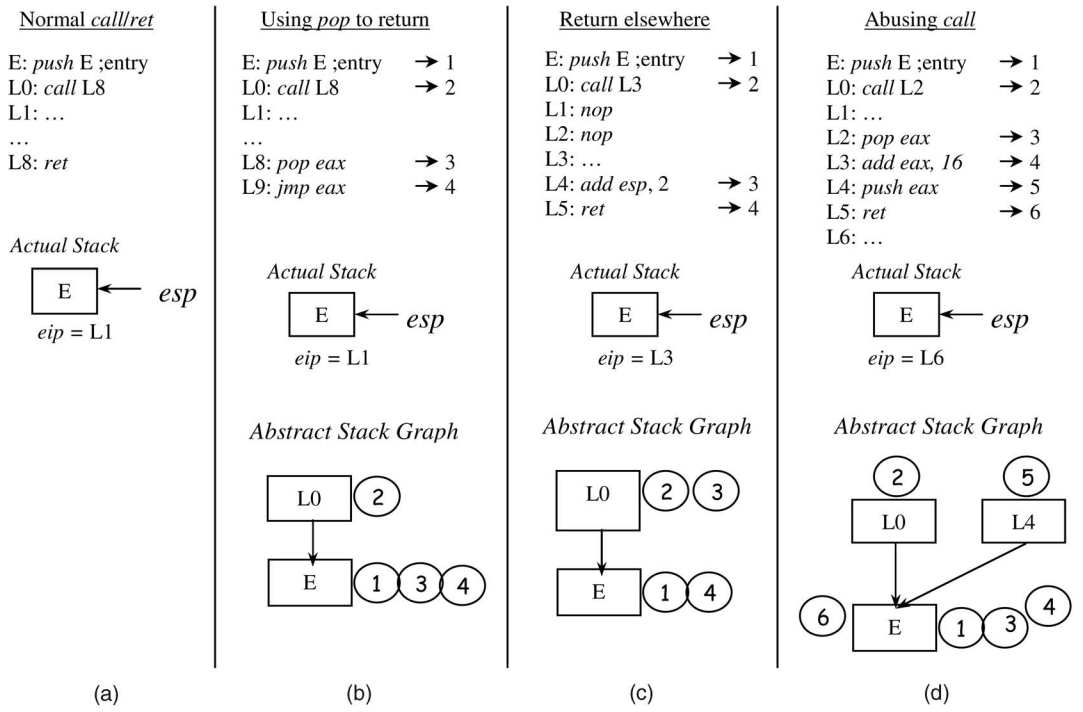


Fig. 10. Possible ways of obfuscating the *ret* instruction.

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 jump to a particular instruction. In Fig. 10d, at instruction *L0* a *call* is made to *L2*. At *L2*, the return address is popped off the stack. A new return address is computed and pushed onto the stack (instruction at *L4*). The *ret* instruction at *L5* transfers control to the new address location. The abstract stack graph shown here can be used to detect such abuse. At program point 5, immediately before the *ret* instruction the abstract stack state is L4 | 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*.

The evaluation function $\mathcal{E}_3$ that identifies obfuscated *ret* is applied on the *pop* instruction and is defined as:

$$\mathcal{E}_3 : INST \rightarrow ASG \rightarrow ASG \times \wp(ADDR \times ADDR)$$
$$\mathcal{E}_3\ [\mathrm{m} : pop](N, AE, ASP) =$$
$$\mathcal{E}_3\ next(\mathrm{m})(abspop\ \mathrm{m}\ (N, AE, ASP))$$
$$\cup\ (\emptyset, \{(\mathrm{a}, \mathrm{m}) | \mathrm{a} \in ASP, is\_call(\mathrm{a})\}).$$

Obfuscated *ret* is detected when the top of stack is identified as a *call* instruction. The function *is_call( )* checks if the address at the top of the stack is the address of a *call* instruction. The address pair (a, m) is identified as an obfuscated *ret*. Since the return address was popped using the *pop* instruction, interpretation continues at the instruction following the *pop* instruction.

6 Implementation and Results

The obfuscations described in the previous section can take away important cues that are used during both automated and manual analysis of suspicious binaries. While a determined, experienced programmer can discover the obfuscations, the time spent in making the discovery can be precious when the malware is actively spreading. Hence, it becomes important to have a tool that can automate their analysis and detection.

We have implemented a tool called DOC (Detector of Obfuscated Calls) which is a prototype to demonstrate our method for detecting obfuscated calls and returns in binaries. DOC statically identifies several types of obfuscations related to the *call* and *ret* instructions, promising to speed up the process of determining whether a program is malicious. In the following sections we present the results of using DOC to analyze W32.Evol virus.

6.1 About DOC

DOC is implemented in Java as a plug-in to the Eclipse Platform (www.eclipse.org). Fig. 11 shows a screenshot of

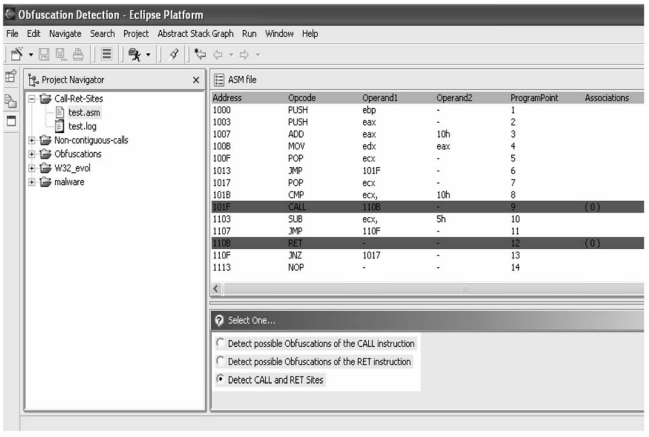


Fig. 11. DOC user interface.

```
:0040153F                mov     dword ptr [eax], 'TteG'
:00401545                mov     dword ptr [eax+4], 'Ckci'
:0040154C                mov     dword ptr [eax+8], 'tnuo'
:00401553                mov     byte ptr [eax+0Ch], 0
:00401557                push    eax
:00401558                call    sub_401280
:0040155D                push    eax
:0040155E                call    sub_4012A7
:00401563                mov     [ebp+0], eax
:00401566                add     esp, 10h
:00401569                pop     ebp
:0040156A                retn
:0040156A sub_401530     endp ; sp = -0Ch
```

Fig. 12. W32.Evol code with Multiple Obfuscations.

DOC when opening an assembly file (.asm extension). DOC provides the ability to open any number of projects at the same time. The navigator view is used to browse and open files in a project. The files are displayed in the file view.

DOC takes as input an assembly file or a disassembled binary obtained from a disassembler such as IDAPro. The abstract stack graph for a given assembly program is constructed by interpreting each instruction of the program. The operations performed by the instruction on a real stack are instead performed on the abstract stack graph. Once the abstract interpretation terminates, the abstract stack graph contains an abstraction of all possible stacks at each statement. DOC analyzes the abstract stack graph to:

- Match *call-ret* instructions.
- Detect obfuscated calls.
- Detect obfuscated returns.

DOC returns its results by highlighting and annotating the assembly. The annotations contain links to related code when there are multiple occurrences of the same type of obfuscation.

6.2 Call Obfuscation in W32.Evol

W32.Evol is a virus that hides constant data as code and modifies it from generation to generation. It builds the constant data on the stack from variable data before it passes them to the actual function or API that needs them. An antivirus tool that looks at the address of the target of the *call* instruction to determine if a system library function is being called will fail in this case. Instead of using the *call* instruction, the virus first pushes the address of the function to be called on the stack, and then later uses the *ret* instruction to make the *call*. Analyzers looking for the explicit *call* will miss it.

The common sequence of instructions to make a system call, say *GetTickcount*, on a Windows environment is as follows:

Push add1 ; "kernel32.dll"
Push add2 ; "GetTickCount"
call GetProcAddress
call [eax] ; "call GetTickCount"

Here, addr1 and addr2, respectively, are pointers to strings "kernel32.dll" and "GetTickCount" located in the data segment. The addresses of these strings are pushed on the stack. The kernel32.dll function GetProcAddress is called, which returns the address of the function "GetTickCount" in the *eax* register. The program then does an indirect call to the address in *eax*, effectively making a call to *GetTickCount*. Disassemblers, such as IDA Pro, can detect such patterns of call and aid in detecting system calls.

Fig. 12 shows a code fragment from W32.Evol for calling the function *GetTickCount*. This code has multiple obfuscations, none of which are detected by IDA Pro. The reasons for this are instructive. IDAPro assumes that the *retn* instruction at address 0040156A actually returns from the procedure. Thus, it deems this statement as ending the procedure that has entry at address 00401530. IDA Pro indicates the end of a procedure by introducing the dummy directive *endp*. Thus, it deduces that the *retn* statement matches "*call* 00401530" instructions.

The *retn* instruction, it turns out, is performing a *call*. The value returned from *GetProcAddress* is moved to the stack, and the stack pointer modified such that when the *retn* instruction is executed, it transfers control to *GetTickCount*. This can be verified by manually analyzing the virus in a debugger such as OllyDbg.

Fig. 13 presents the code of Fig. 12 with annotations created by such a manual analysis. The address of the API functions is looked up form the entry points or addresses within kernel32.dll using another Win32 API function

```
0040153F mov dword ptr ds:[eax], 54746547 ;'TteG'
00401545 mov dword ptr ds:[eax+4], 436B6369 ;'Ckci'
0040154C mov dword ptr ds:[eax+8], 746E756F ;'tnuo'
00401553 mov byte ptr ds:[eax+c], 0;  '\0'
00401557 push eax; ptr to "GetTickCount".
00401558 call 00401280; gets base address of kernel32.dll base.
0040155D push eax;
0040155E call 004012A7; obfuscated call to GetProcAddress()
00401563 mov dword ptr ss:[ebp], eax; addr of GetTickCount().
00401566 add esp, 10
00401569 pop ebp
0040156A retn; transfer control to GetTickCount().
```

Fig. 13. Annotated code of Fig. 12.

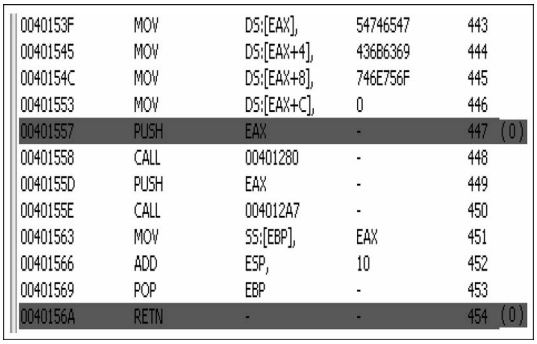


Fig. 14. Using DOC to detect obfuscated *call*.

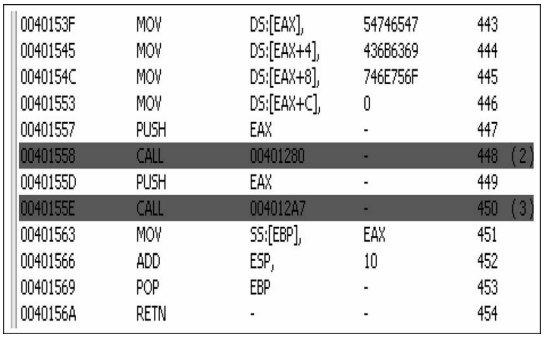


Fig. 15. Using DOCs to detect valid calls.

called *GetProcAddress( )*. This function requires as parameters the name of the Win32 API function to be called and the kernel32 module handle which is the *kernel32.dll* base address. These are passed in an obfuscated way as parameters to *GetProcAddress( )*.

The name of the string of the function being called is passed in a piece meal fashion by pushing several two byte values on the stack. The kernel32 module handle is placed above a string marker "eVOL" that it previously pushed on the stack. As shown in Fig. 13, instructions labeled 0040153F through 0040155D are instrumental in doing this.

The obfuscation lies in the *call* to *GetProcAddress( )* as well as in the *call* to each of the other kernel functions. The virus searches for the *GetProcAddress( )* API entry-point using an 8-byte string. This string is calculated as the virus generates new mutations. The actual string is placed on the stack only. Therefore, the virus cannot be detected using any search strings with wildcards once the virus mutates itself to a few generations. To detect this call, the stack data must be analyzed. The instruction at 0040155E calls a routine that searches through the stack for a special string marker, "eVOL." The address of the function *GetProcAddress( )* is placed at some constant distance from this string marker. It retrieves this address, pushes it on the stack and then executes a *ret* instruction which transfers control to *GetProcAddress( )*. *GetProcAddress( )* returns the address of the kernel function that needs to be called in the register *eax* (instruction at 00401563). This value is pushed on the stack and control is transferred to this kernel function by executing a *ret* instruction (instruction at 0040156A).

6.3 Using DOC to Detect *call* Obfuscations

Fig. 14 shows a portion of the code where DOC detects the obfuscated *call* to the kernel function *GetTickCount()*. The *push* instruction at address 00401557 and the *retn* instruction at address 0040156A are instrumental in obfuscating the *call* to *GetTickCount()*. This is indicated by highlighting these instructions in red. The annotation "(0)" at the end of these instructions indicates that the two belong to the same call obfuscation. W32.Evol uses similar code to make system calls in 25 locations. IDA Pro misses all of these calls, where as DOC highlights every such *retn* instruction as making a call.

6.4 Using DOC to Match *call-ret* Sites

Fig. 15 shows the same code as Fig. 12, but it also shows of the results of running DOC's analysis for matching *call-retn* instructions. The two call instructions at addresses 00401558 and 0040155E are highlighted and are annotated "(2)" and "(3)," respectively. These numbers are arc labels in the effective call graph.

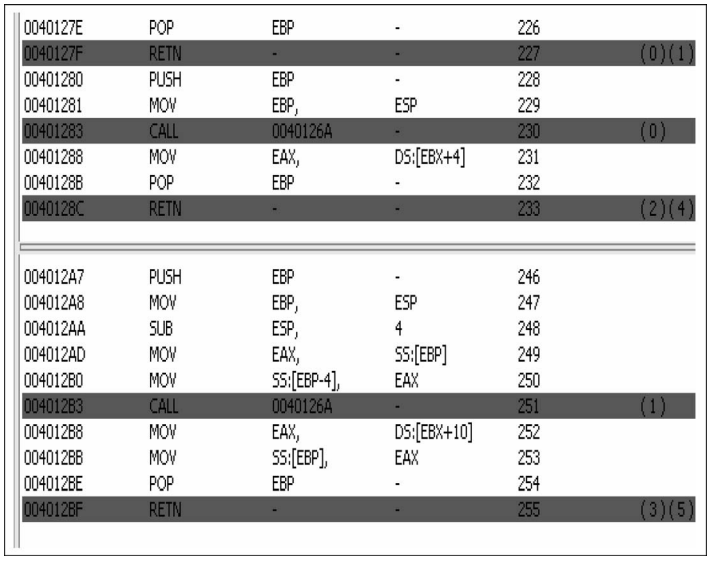
Fig. 16 shows return sites corresponding to these statements. These statements are annotated with the numbers "(2)" and "(3)," which are matched to the call sites so labeled. This figure also shows *retn* statements matching call sites annotated as "(0)" and "(1)." As is expected, one *retn* statement may match multiple call sites. DOC correctly found matching *retn* statements for all 33 *call* statements of W32.Evol. In several instances, the procedure code was not contiguous.

7 Concluding Remarks

A method for modeling stack use 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. The abstract stack graph was defined in abstract interpretation form. An algorithm for constructing it was also defined.

An abstract stack graph may be used to support disassembly of obfuscated code and to detect obfuscations related to stack operations. Eight different obfuscations were shown to be detectable, and the methods for doing so outlined. These are obfuscations in common use by virus writers. The obfuscation detection technique presented is efficient and is demonstrably effective in finding the sort of *call/retn* obfuscations found inW32.Evol. We believe its techniques can be an important part of an AV researcher's toolkit, and can significantly speed up analysis of obfuscated binaries.

The abstract stack graph and the algorithm for constructing an abstract stack graph are partial solutions for detecting obfuscations in binaries. It is partial in the sense that the obfuscation detection is confined or rather narrowed down to those done using stack related instructions such as *push*, *pop*, *call*, and *ret*. Also, only those



0040127E	POP	EBP	-	226	
0040127F	RETN	-	-	227	(0)(1)
00401280	PUSH	EBP	-	228	
00401281	MOV	EBP,	ESP	229	
00401283	CALL	0040126A	-	230	(0)
00401288	MOV	EAX,	DS:[EBX+4]	231	
0040128B	POP	EBP	-	232	
0040128C	RETN	-	-	233	(2)(4)
004012A7	PUSH	EBP	-	246	
004012A8	MOV	EBP,	ESP	247	
004012AA	SUB	ESP,	4	248	
004012AD	MOV	EAX,	SS:[EBP]	249	
004012B0	MOV	SS:[EBP-4],	EAX	250	
004012B3	CALL	0040126A	-	251	(1)
004012B8	MOV	EAX,	DS:[EBX+10]	252	
004012BB	MOV	SS:[EBP],	EAX	253	
004012BE	POP	EBP	-	254	
004012BF	RETN	-	-	255	(3)(5)

Fig. 16. Using DOC to detect valid *call-ret* sites.

instructions that perform unit *push* and *pop* operations are handled. Instructions such as *pusha* and *popa* that increment and decrement the stack by more than one unit (usually 4 bytes) are not handled.

Also, the abstract stack graph does not model the contents of memory locations and registers. Thus, modifications made to the stack pointer by moving data through other registers and memory locations cannot be identified. Ongoing work in this area by Venable et al. [25] attempts to overcome this limitation by combining the abstract stack graph model with a "value-set analysis" model of memory locations and registers developed by Balakrishnan and Reps [3].

REFERENCES

[1] "TESO, Burneye Elf Encryption Program," https://teso.scene.at, Nov. 2004.

[2] "z0mbie," http://z0mbie.host.sk, Nov. 2004.

[3] G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in X86 Executables," *Proc. Int'l Conf. Compiler Construction (CC),* 2004.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)Possibility of Obfuscating Programs," *Proc. Conf. Advances in Cryptology (CRYPTO '01),* 2001.

[5] D. Chess and S. White, "An Undetectable Computer Virus," *Proc. Virus Bulletin Conf.,* 2000.

[6] S. Cho, "Win32 Disassembler," http://www.geocities.com/~sangcho/disasm.html, Nov. 2004.

[7] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant, "Semantics-Aware Malware Detection," *Proc. IEEE Symp. Security and Privacy,* 2005.

[8] M. Christodrescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *Proc. 12th USENIX Security Symp. (Security '03),* 2003.

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

[10] F. Cohen, "Computational Aspects of Computer Viruses," *Computers and Security,* vol. 8, pp. 325-344, 1989.

[11] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Trans. Software Eng.,* vol. 28, pp. 735-746, 2002.

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

[13] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs," *Proc. Second Int'l Symp. Programming,* 1976.

[14] N.D. Jones and F. Nielson, "Abstract Interpretation: A Semantics-Based Tool for Program Analysis," *Handbook of Logic in Computer Science: Semantic Modelling,* vol. 4, pp. 527-636, 1995.

[15] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," *Proc. USENIX Security Conf.,* 2004.

[16] A. Lakhotia, M.E. Karim, A. Walenstein, and L. Parida, "Phylogeny Using Maximal pi-Patterns," *Proc. 14th EICAR Conf.,* 2005.

[17] A. Lakhotia and M. Mohammed, "Imposing Order on Program Statements and Its Implication to AV Scanners," *Proc. 11th IEEE Working Conf. Reverse Eng.,* 2004.

[18] A. Lakhotia and P.K. Singh, "Challenges in Getting Formal with Viruses," *Virus Bull.,* pp. 14-18, 2003.

[19] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *Proc. 10th ACM Conf. Computer and Comm. Security,* 2003.

[20] C. Nachenberg, "Computer Virus-Antivirus Coevolution," *Comm. ACM,* vol. 40, pp. 46-51, 1997.

[21] D. Schmidt, "Abstract Interpretation and Static Analysis," http://www.cis.ksu.edu/santos/schmidt/Escuela03/, Feb. 2005.

[22] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," *Proc. Ninth Working Conf. Reverse Eng. (WCRE '02),* 2002.

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

[24] P. Szor and P. Ferrie, "Hunting for Metamorphic," *Proc. Virus Bull. Conf.,* 2001.

[25] M. Venable, M. Chouchane, M.E. Karim, and A. Lakhotia, "Analyzing Memory Accesses in Obfuscated x86 Executables," *Proc. Conf. Detection of Intrusions and Malware and Vulnerability Assessment,* 2005.

[26] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra, "An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java," *Proc. 10th Working Conf. Reverse Eng.,* 2003.

[27] G. Wroblewski, "General Method of Program Code Obfuscation," technical report, Inst. of Eng. Cybernetics, Wroclaw Univ. of Technology, 2002.

Arun Lakhotia is an associate professor of computer science at the Center for Advanced Computer Studies, University of Louisiana at Lafayette. For almost a decade, he has worked in the area of program understanding, reverse engineering, and reengineering. He is now directing that experience to analysis of malicious programs under a project for developing Next Generation Antivirus Technologies, a project funded by the Louisiana Governor's Information Technology Initiative. Dr. Lakhotia has also ventured into working on autonomous vehicles and is the Team Leader of Team CajunBot, a finalist in the 2004 DARPA Grand Challenge, and a participant in the 2005 Challenge. He is the recipient of 2004 Louisiana Governor's University Technology Leader of the Year award.

Eric Uday Kumar received the MS degree from the University of Louisiana, Lafayette, where his research included virus detection using static analysis and resulted in the creation of DOC, a tool for detecting call obfuscations in executables. He is currently a software engineer developing antivirus software for Authentium, Inc.

Michael Venable received the MS degree from the University of Louisiana at Lafayette, where he currently works as a software engineer. His research focuses on reverse-engineering, static analysis, and antivirus technologies with an emphasis on virus detection via abstract interpretation. He is currently working on a version of DOC that will utilize value range analysis for more precise results.