Stack Shape Analysis to Detect Obfuscated calls in Binaries

Arun Lakhotia and Uday Eric
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 maybe 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 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. The abstract stack graph may be created by abstract interpretation of the binary executable and may be used to detect obfuscated calls.

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) [6, 8, 9, 12, 18]. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners [4, 13, 17].

There now exist malicious programs—virus, worms, Trojans, spyware, backdoors, etc.—that are self-obfuscating. These programs, termed as *metamorphic* by the anti-virus community, obfuscated themselves before propagating to the next host [10]. Repeated obfuscation ensures that two copies of a virus, henceforth used to refer to a malicious program, are different and successive generations are significantly harder to reverse engineer.

Christodrescu et al. [4] present malicious code detection as being an obfuscation-deobfuscation game between the virus writers and researchers working on malicious code detection. The obfuscations are such that there is considerable change in the byte sequence of the executable obtained but do not change the actual sequence of instructions being executed. The aim of the virus writer is to fool the antivirus tool into believing that it is dealing with a safe executable.

To achieve its malicious intent, a virus must access system resources, such as, the file system and network. Based on analysis of thousands of viruses anti-virus companies have developed knowledge-bases of combinations of system calls that may be considered malicious. Some anti-virus scanners look for bit patterns associated with system calls to identify malicious programs. Such scanners are easily thwarted by metamorphic viruses or viruses that simply obfuscate system calls [15, 16]. 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.

To detect obfuscated viruses some anti-virus scanners execute a virus in an emulated environment and trap the calls to the operating system. Such *dynamic analyzers* must use some heuristic to determine when to stop analyzing a program, for its possible that a program may terminate on user command. These analyzers are fooled by viruses by introducing some delay, i.e., code that wastes time.

This paper presents a method to statically detect obfuscated calls when the obfuscation is performed by 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 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, a la, static analysis, is concisely represented in an abstract stack graph.

As it is with any anti-virus solution, our method may be used to increase the level of difficulty for writing a virus. Our method can help by removing some common obfuscation from the toolkit of a virus writer. However, we do not claim to 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.

Section 2 presents related work in this area and describes how metamorphic viruses use system calls to library functions to infect, conceal and propagate. 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. The section also includes an example to enumerate the algorithm. Section 5 describes how the abstract stack graph may be used to detect various obfuscations. Section 6 concludes this paper.

2. Related work

The analysis of the bit stream of a binary executable does not offer significant information. To get any meaningful information it must at least be disassembled, i.e., translated to assembly instructions. Most often the disassembled code may be analyzed further, following steps similar to those performed for decompilation [5]. Lakhotia and Singh [11] show 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. [12] 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. [7] presents taxonomy of obfuscating transformations where a detailed theoretical description of various possible obfuscating transformations is presented. There exist obfuscation engines that may be linked to a program to create a self-obfuscating program. Two of them are Mistfall (by z0mbie), which is a library for binary obfuscation [2], and Burneye (by TESO), which is a Linux binary encapsulation tool [1].

The antivirus techniques usually applied to detect metamorphic viruses are heuristics and emulation coupled with static or dynamic analysis of the code [14]. Christodrescu et al. [4] show that current anti-virus scanners can be subverted by very simple obfuscating transformations [4].

Many recent viruses heavily depend on calls to system libraries. The Win95/Kala.7620 was one of the first viruses to use an API to transfer control to the virus code (in this case, its decryptor) [16]. Modern anti-virus tools are to have API emulation to detect these. To make these system calls, a popular technique adopted by most virus writers (as observed in recent binary viruses when disassembled and analyzed), targeting the Windows operating system is to locate the internal *KERNEL32.DLL* entry point and call *KERNEL32* functions by ordinal. This is done by locating *KERNEL32.DLL*'s *PE* header in memory and using the header info to locate *KERNEL32*'s export section.

A set of suspicious API strings will appear in non-encrypted Win32 viruses. This can make the disassembly of the virus much easier and potentially useful for heuristic scanning. An effective anti-heuristic/anti-disassembly trick appearing in various modern viruses is to hide the use of API strings to access particular APIs from the Win32 set. For example, the Win32/Dengue virus does not use API strings to access particular APIs [16]. Some recent viruses use a checksum list of the actual strings. The checksums are recalculated via the export address table of *KERNEL32.DLL* and the address of the API is found. Hence, the absence of API name strings should not be inferred as non-existence of any API calls in the program.

There is hope, however. A recent result by Barak et al. [3] proves that in general, perfect program obfuscation is impossible, i.e., for any obfuscator one can write a deobfuscator. This is likely to have an effect on the pace at which new metamorphic transformations are introduced. Lakhotia and Singh [11] observe that though 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 the same challenges that the anti-virus technologies face.

Christodrescu and Jha use abstract patterns to detect malicious patterns in executables [4]. Mohammed has developed a technique to undo certain obfuscation transformations, such as statement reordering, variable renaming, and expression reshaping, with the [13].

3. Abstract stack and abstract stack graph

We propose the notion of an *abstract stack* that captures the stack activity and simplifies analysis of control flow but is quite different from the *actual stack*. The actual stack for a program code keeps track of the values being pushed and popped in a LIFO (Last In First Out) sequence. The abstract stack on the other hand keeps track of the addresses of the instructions causing the push and pop of values in a LIFO sequence. For example, consider Figure 1. Each of the instructions 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* would have been pushed onto the abstract stack, but due to the pop instruction at *L3*, the address *L2* is popped and next *L4* is pushed onto the abstract stack.

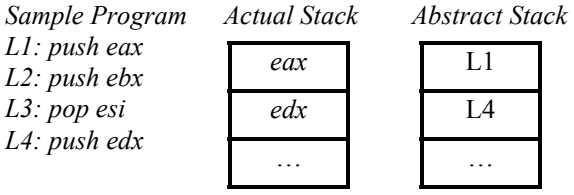

Figure 1.

An example computing the abstract stack at each program point is shown in Figure 2. Figure 2a is a sample code for which the control flow graph is as shown in figure 2b. Each block in the control flow graph represents a program point. The blocks are numbered in reverse post order and the edges of the graph are tagged as either being a tree edge, a forward edge, a cross edge or a back edge. Figure 2c shows few of the abstract stacks that are possible at some of the program points. These are not all of the abstract stacks but numerous more are possible at each program point. For instance the 3rd abstract stack at program point 2 is the result of traversing the control flow graph through the program points 1 → 2 → 3 → 4 → 3 → 4 → 5 → 2 in that order. The abstract stack at program point 4 is the result of traversing the program points 1 → 2 → 3 → 4 → 3 → 4 → 5 → 2 → 3 → 4 in that order while the abstract stack at program point 8 is the result of traversing the program points 1 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 3 → 5 → 2 → 7 → 8.

As can be seen in Figure 2c, at each program point, there can be numerous possible abstract stack states. This might result in having to keep track of exponential number of abstract stacks. A more efficient way would be to have an abstract stack graph. An abstract stack graph is a directed graph representing the union of all the abstract stacks at each program point. A backward path in the graph ending at a particular node represents an abstract stack at that program point. Figure 2d shows the abstract stack graph for the control flow graph (Figure 2b). An algorithm to construct the abstract stack graph from the control flow graph is described in the next section.

4. Algorithm to construct the abstract stack graph

In this section we explain the algorithm in plain English. The algorithm assumes that an approximate control flow graph is available.

The algorithm to construct an abstract stack graph consists of the following seven steps:

Step 1: For each block of the control flow graph, replace sequences of instructions that are equivalent *push* or *pop* instructions. These instructions modify the stack pointer. For instance, the sequence of instructions:
dec esp
dec esp
mov [esp], eax
are replaced with *push eax* while the sequence of instructions:
mov eax, [esp]
inc esp
inc esp
are replaced with *pop eax*.

Step 2: For each block of the control flow graph, check to see if there is more than one stack operation (*push, pop, call, ret*). If such a block is found, break it into appropriate number of fall-through blocks such that each block now formed has exactly one stack operation. Each block in the new control flow graph thus formed represents either a *push block*, a *pop block*, a *call block*, a *ret block*, or a *block with no stack operation*.

Step 3: Tag each edge of the control flow graph as *tree-edge, forward-edge, side-edge* or *back-edge*.

Step 4: Process each block by traversing the control flow graph in reverse post order.

Step 5: For each block type *B* do the following:

5(a) If the block *B* is of type *push block* or *call block*, then create a node in the abstract stack graph, with the address of the instruction causing the *push* (or *call*) and tag its number to it. If this block *B* has an incoming *tree-edge* in the CFG from another block *B1*,

then create an edge from *B* to *B1* in the abstract stack graph (where *B1* is also a node in the abstract stack graph). Do the same if there are more incoming tree edges to *B*.

5(b) If the block *B* is of type *pop block* or *ret block*, then for each block *B1* that has a *tree-edge* pointing to *B* in the CFG is processed. The block to which such *B1* are pointing to in the as of yet formed stack graph are marked with the tag number of *B*.

5(c) If the block *B* has *no stack operations*, then for each block *B1* that has a *tree-edge* pointing to *B* in the CFG is processed. The blocks where the tag numbers of such blocks *B1* are present in the as of yet formed stack graph are marked with the tag number of *B*.

Step 6: Each of the blocks with incoming *forward-edges*, *side-edges* and *back-edges* in the CFG are processed in this order. These blocks are processed according to their block types applying the rules as in step 5.

Step 7: If an update takes place in the abstract stack graph, steps 1 through 6 are repeated until there is no more change in the abstract stack graph.

Let us see the algorithm in working on the example control flow graph in Figure 2(b). Let us assume the control flow graph to be in required form with *push*, *pop* and *call* blocks and ignore, for the time being, the instructions that might form a *no-stack-operation* block. First, block E is formed in the abstract stack graph and tagged with 1. The next block is retrieved, which is a *push block*. Hence the block B0 is formed in the abstract stack graph and since it has an incoming *tree edge* from E in the CFG, an edge is created from B0 toward E. Similarly the *push blocks* B1 and B3 are included in the abstract stack graph. The next block 5 is a *pop block* and has an incoming *tree edge* from block 4. Blocks to which 4 has an edge toward in the as of yet formed abstract stack graph are tagged with 5 now. Hence, 5 is tagged onto the block B1 in the abstract stack graph. Similarly, B6 is formed with an edge toward B1 and 7 is tagged with E because 2 is pointing toward E. Then block B4 is formed is tagged with 8 and is pointing toward E because 7 is tagged over there. Next we consider the *forward*, *cross* and *back* edges. The *forward edge* results in an edge from B1 pointing toward B3. The *cross edge* results in 5 being tagged to E. The *back edge* results in an edge from B0 pointing toward wherever 5 is tagged and that is B1. The algorithm is repeated for a new iteration because new information has been added to the abstract stack graph. Results in an edge from B6 pointing toward E because 5 is tagged over there. 7 is tagged with B1 because B0 is now pointing toward B1. And now an edge is created from B4 to B1 because 7 is tagged to B1. The next iteration generates no new changes and so the abstract stack graph is now stable

The algorithm is robust enough to cope with the occurrences of unbalanced *push* or *pop* in loops. 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 produce a correct abstract stack graph that encompasses all the possible abstract stacks at each program point. The following section discusses various ways in which *call* obfucations are possible and the use of the abstract stack graph in detecting them.

5. Detecting obfuscated calls

A *call* to a procedure within the same segment is near and performs the following: it decrements the stack pointer (*esp*) by a word and pushes the instruction pointer (*eip*, containing the offset of the instruction following the *call*) onto the stack. Next it inserts the offset address of the called procedure into *eip*. This is shown in figure 3(a). A *ret* that returns from a near procedure basically reverses the *call*'s steps. It pops the old *eip* value from the stack into *eip* and increments *esp* by a word. This is shown in figure 5(a).

The goal of the obfuscator remains to obfuscate the *call* instruction in such a way so that the antivirus software is unable to detect that a system call is indeed being made. The obfuscation is not just limited to the call being made, but in most cases is also extended to the parameters being passed to the *call* and also to the *ret* statement.

Figure 3 shows some of the ways of obfuscating the *call* instruction. E, L0, L1, etc., at the left of each instruction represent the address of the instruction and the numbers following the "→" on the right of each instruction represent the flow of control. In figure 3(b) we see how a combination of *push/jmp* can be used to mimic the *call*. The 1st instruction pushes the entry point of the code onto the stack. The 2nd instruction pushes the offset address of the instruction following the *call*, which is the return address, onto the stack. The 3rd instruction loads the effective address of the instruction "fun" into the register *eax* and the 4th instruction does a jump through register to this address. When the function "fun" eventually does a *ret*, the control returns to the return address previously pushed onto the stack. Hence, without using a *call*, the same functionality is achieved here as is intended in 3(a). The abstract stack graph shown for the sample code in figure 3(b) can be used to detect this

Sample Program

E: //entry point
B0: push eax
C1: sub ecx, 1h
C2: beqz B2
B1: push ebx
B3: push ecx
C4: dec ecx
C5: beqz B1
C6: jmp B5
B2: pop eip
B4: push esi
B5: pop eip
C6: beq B0
B6: call abc

Figure 2a.

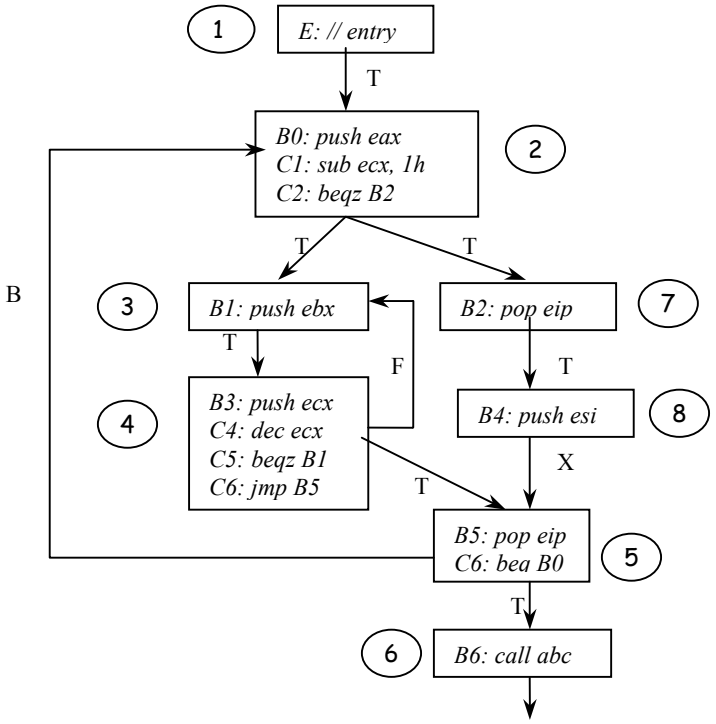

Figure 2b.

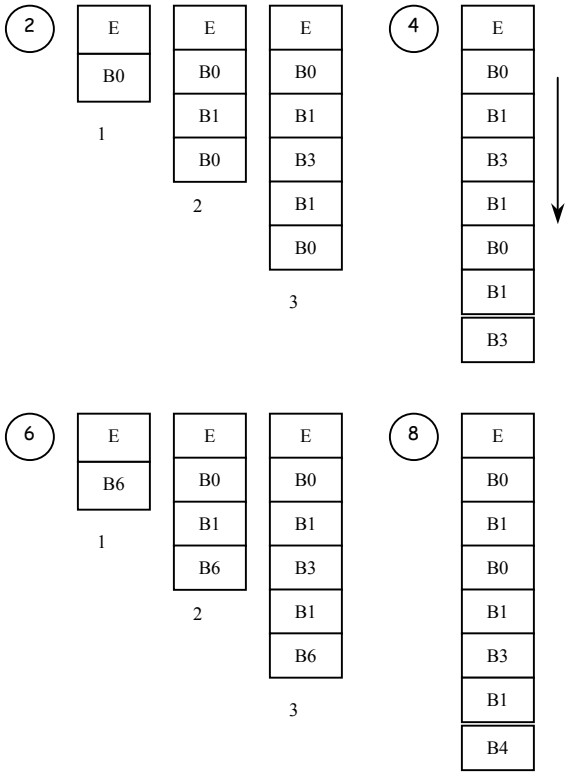

Figure 2c.

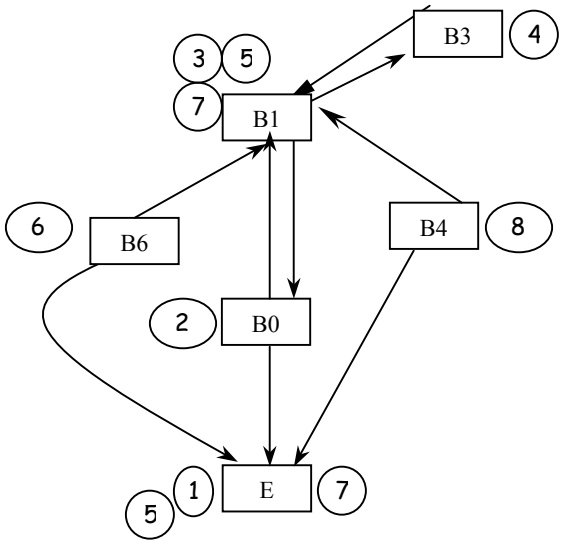

Figure 2d.

Figure 2.

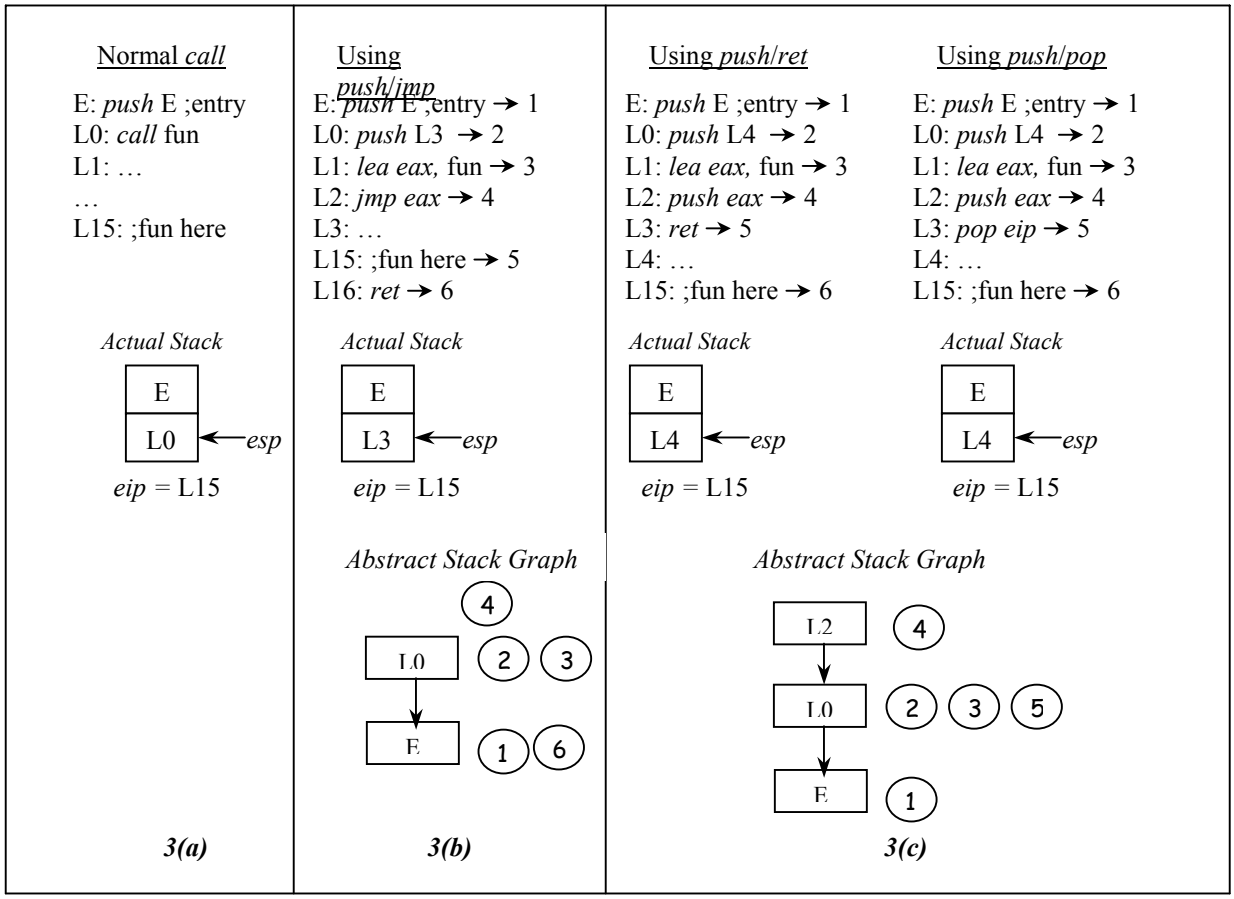

Figure 3. Possible ways of obfuscating a call instruction

obfuscation. At instruction 6, we can trace back in the abstract stack graph to the instruction that pushed the return address onto the stack, which is instruction 2 By looking at the semantics of the instructions following it we now know that a "call" is being made.

Figure 3(c) shows a combination of *push/ret* or *push/pop* that can be used to obfuscate a *call*. At instruction 5, where a *ret* or a *pop eip* is done, the address that was previously pushed onto the stack by instruction 4 is now in *eip* transferring control to "fun". The abstract stack graph shown here can be used to dtect this obfuscation. At instruction 5, tracing back in the graph to the instruction that pushed the "popped" address brings us to L2, which is the 4th instruction. By looking into *eax* we now know that "fun" was being called.

Win32.Evol uses a combination of *push/pop* to make an obfuscated *call* to a system library function (LoadLibraryA in this case). It first moves the function name into a register (*eax*) and then pushes this onto the stack and later *pop*s it to transfer control to the function. Sample code from the virus is as follows:

lea eax, [*ebp*+var_14]
mov dword ptr [*eax*], 'daoL'
mov dword ptr [*eax*+4], 'rbiL'
mov dword ptr [*eax*+8], 'Ayra'
mov byte ptr [*eax*+0Ch], 0
push eax
….
pop ebp

5.1. Detecting obfuscations to parameters being passed to a call

Parameters to a function in assembly are passed by pushing them onto the stack before a *call* to the function is made. The code in the called function later retrieves these parameters from the stack. Figure 4 shows possible ways of obfuscating parameters to calls. 4(b) shows the use of out of turn pushes to distort the actual function to which the parameters are being passed. The first *call* occurring after the pushes need not be the function to which the parameters are

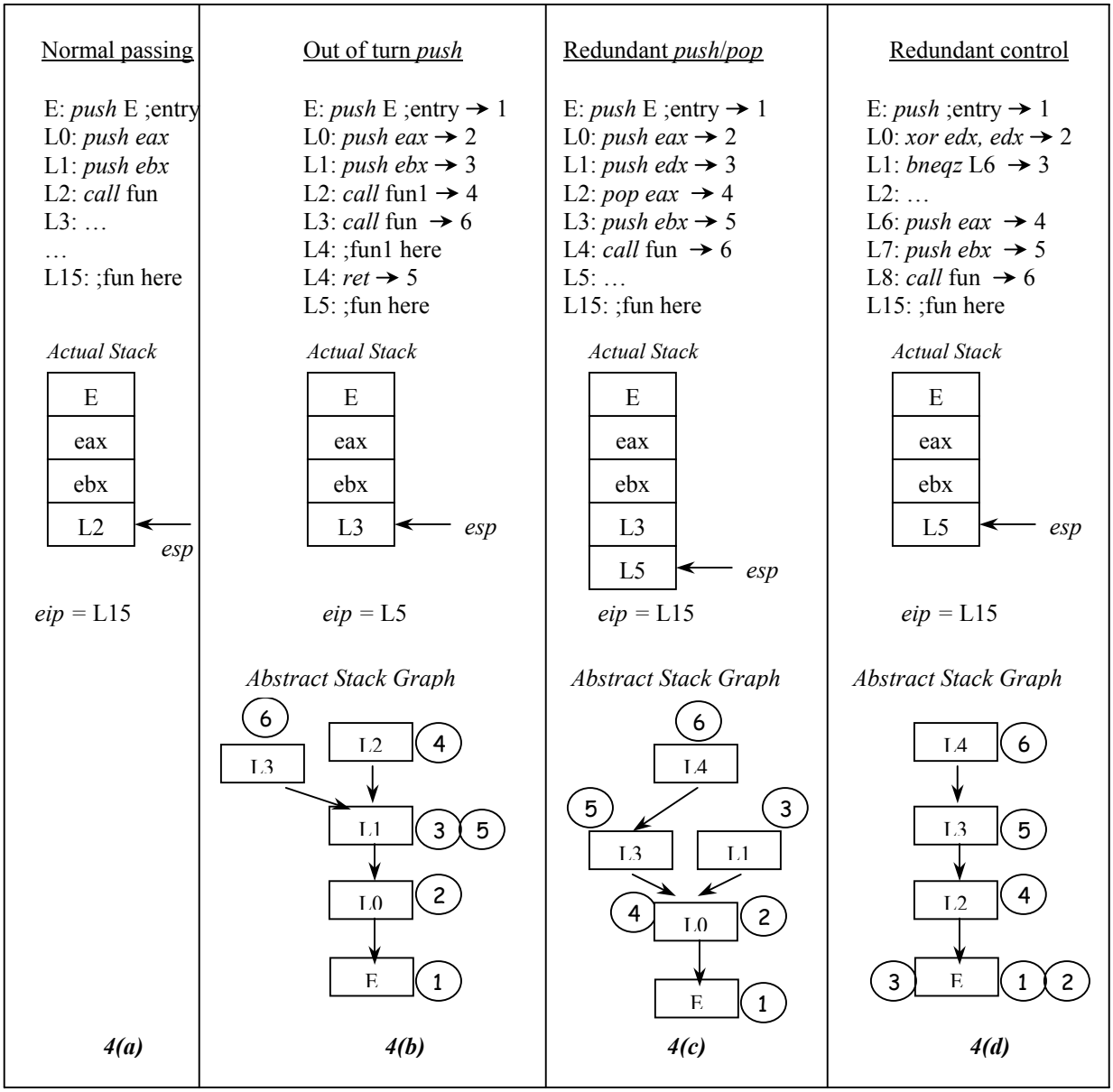

Figure 4. Possible ways of obfuscating parameters to calls

actually being passed. Instructions 2 and 3 push the parameters in registers *eax* and *ebx* onto the stack. The 4th instruction calls "fun1" that simply returns (instruction 5) without actually retrieving any of the parameters. The 6th instruction calls "fun" that actually uses these parameters. The abstract stack graph shown here can be used to detect this obfuscation. At program point 6 in the abstract stack graph, which is a *call* to "fun", the state of the abstract stack is E, L0, L1, L3. Hence the parameters being pushed at L0 and L1 pertain to the *call* to "fun". Though the same abstract stack state holds at program point 4, which is a *call* to "fun1", it is ruled out that the pushes at L0 and L1 pertain to "fun1". This is because the *ret* at program point 5 is associated with the call to "fun1". This too is deduced form the abstract stack graph itself by checking the address of the instruction that popped the top of the abstract stack. Since this is a *call* to "fun1", we have matched the *call/ret*. This shows that the abstract stack graph can also be used to match blocks of *call/ret*.

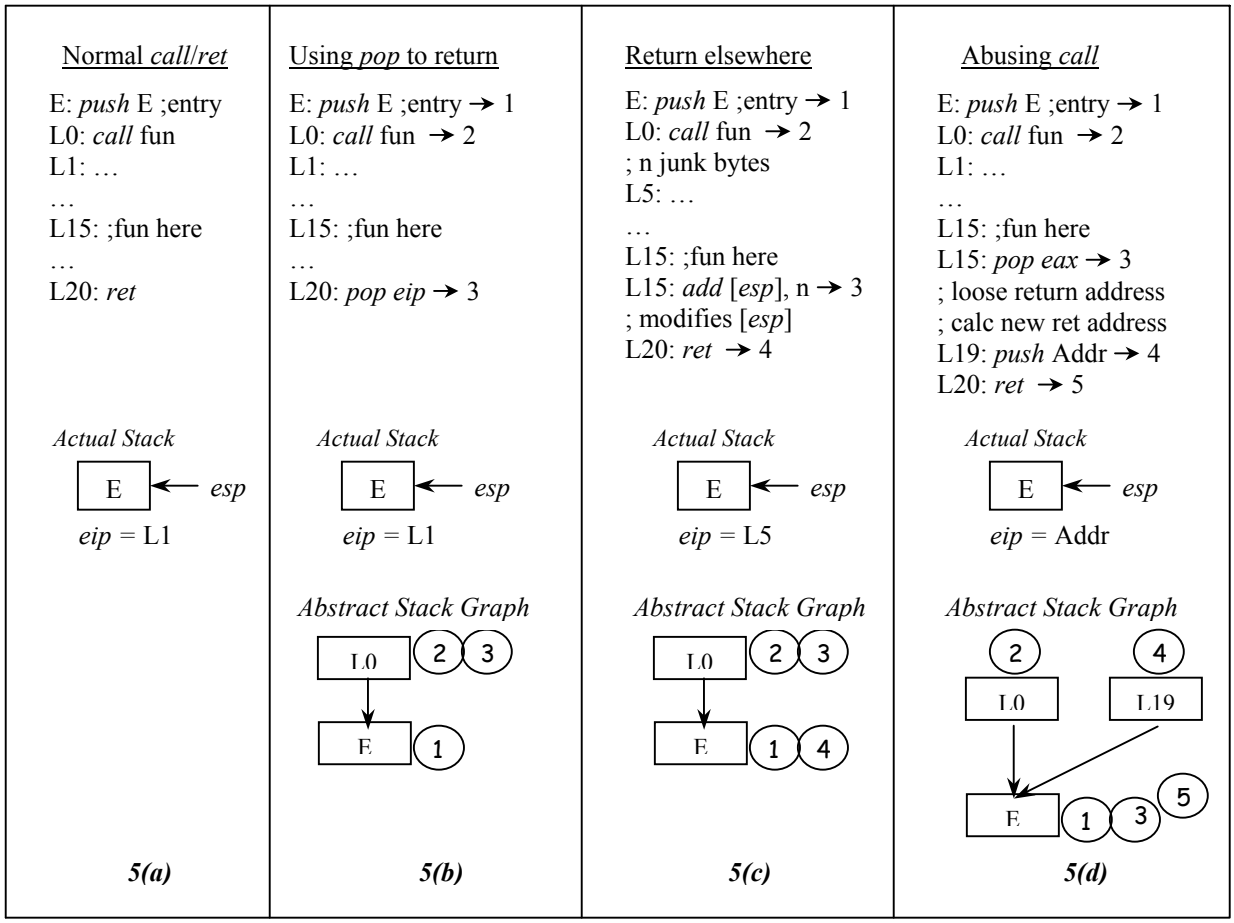

Figure 5. Possible ways of obfuscating the ret

When we get to a *call*, we know the instructions that have pushed elements on the stack. If a *call* takes two instructions, the top two pushes on the stack will be the parameters. The i^{th} parameter corresponds to the i^{th} element on the stack (starting from the top). This is assuming the first parameter is pushed last. If the last parameter is pushed first, we change it around.

Figure 4(c) shows the use of redundant *push/*pop to obfuscate parameters to a *call*. Instructions 3 and 4 are redundant. At program point 6 in the abstract stack graph (in figure 4(c)), the abstract stack state shows address L0 and L3 as being on the stack, which are the instructions that push the parameters for "fun", hence eliminating any redundant *push/pop*s.

Figure 4(d) shows the use of redundant control to obfuscate parameters to a *call*. This is done by exploiting the assumption that a conditional branch has two possible targets. The conditional branch may be instrumented, at runtime, to always go in one direction – i.e., either it is always taken and never falls through, or it is never taken and always falls through [18]. This technique relies on using predicates that always evaluate to either the constant *true* or the constant *false*, regardless of the values of their inputs: such predicates are known as *opaque predicates*. The 2^{nd} instruction results in *edx* containing zero. Hence the 3^{rd} instruction always evaluates to *true* and the branch is taken to L6. The abstract stack graph shown here can be used to detect this redundant control. At program point 6, where the call to "fun" is made, the abstract stack state includes the address of the instructions L6 and L7 that push the arguments for "fun", hence detecting the redundant control.

5.2. Detecting obfuscations to *ret*

A *ret* statement typically *pop*s the top of the stack and returns control to the instruction immediately following the *call* to the function from where it is returning. The *ret* itself can be used to abuse a *call* or

to return elsewhere. Figure 5 shows few of the possible ways of obfuscating *ret*. As shown in figure 5(b) a *pop* can be used instead of a *ret* to obfuscate the *ret*. Using the abstract stack graph, if after a *call* has been detected with no previous *push*es, then a *ret* has been detected. At program point 3 in the abstract stack graph there are no previous *push*es as the stack state shows only L0. Hence an obfuscated *ret* has been detected here.

Instead of the conventional way of returning to the instruction immediately following a *call*, the return address is modified in the called function and control transferred to some other instruction. This is obfuscating *ret* to return elsewhere. In figure 5(c), the 2nd instruction makes the *call* to "fun". Immediately after the *call* instruction, n junk bytes are inserted to locate a specific return address (L5 in this case). Within "fun", the contents of the stack pointer are modified by adding n bytes to the return address (the 3rd instruction) and thus generating a new return address. To use the abstract stack graph to detect this obfuscation requires augmenting the abstract stack to maintain an additional tag called *modified*. When a value is pushed on the stack, *modified* is set to *false*. Now we also augment the algorithm to deal with other instructions. If an instruction may change the content 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 was pushed by a *call* and that value is at the top of the stack when you get to *ret*, it implies that *ret* is returning elsewhere.

The *call* instruction can also be abused to actually jump to a particular instruction. In figure 5(d), a *call* is made to "fun" at instruction 2. Within the function, the return address from "fun" is popped out from the top of the stack. A new return address is computed and pushed onto the stack (instruction 4). The 5th instruction transfers control to the new address location now. The abstract stack graph shown here can be used to detect this abuse of the *call* instruction. At program point 5, the top of the abstract stack has only E. Tracing back on the abstract stack graph, at program point 4, we find the address of the instruction that pushed the new "Addr". If this Addr were a constant, then we can determine it, but if were to be computed at runtime, then there is no way of knowing it.

6. Conclusions

We have presented a method to detect obfuscations of *call* and *ret* instruction. The method performs abstract interpretation of stack related instructions. The interpretation yields an abstract stack graph that represents all possible shapes of the stack at any program point. We have presented various obfuscations of stack related instructions and how they may be deobfuscated using the abstract stack graph. The construction of an abstract stack graph offers a step towards analyzing obfuscated binaries for malicious behavior.

7. References

[1] "TESO, burneye elf encryption program," http://teso.scene.at, Last accessed April 10, 2004.

[2] "z0mbie," http://z0mbie.host.sk, Last accessed April 10, 2004.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs," presented at Advances in Cryptology (CRYPTO'01), 2001.

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

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

[6] C. Collberg and C. Thomborson, "Watermarking, Tamper-proofing, and Obfuscation - Tools for Software Protection," The Department of Computer Science, University of Arizona Technical Report TR00-03, February 2000 2000.

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

[8] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstraction and Unstructuring Data Structures," presented at Proceedings of 1998 IEEE International Conference on Computer Languages, 1998.

[9] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," presented at ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98), San Diego, CA, 1998.

[10] M. Jordon, "Dealing with Metamorphism," in *Virus Bulletin*, 2002, pp. 4 - 6.
[11] A. Lakhotia and P. K. Singh, "Challenges in Getting Formal with Viruses," in *Virus Bulletin*, 2003, pp. 14 -18.
[12] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," presented at Proceedings of the 10th ACM Conference on Computer and Communication Security 2003,, Washington D.C., 2003.
[13] M. Mohammed, "Zeroing in on Metamorphic Viruses," in *The Center for Advanced Computer Studies*. Lafayette, LA: University of Louisiana at Lafayette, 2003.
[14] Symantec, "Understanding heuristics: Symantec's Bloodhound technology," *Symantec White Paper Series*, vol. XXXIV, 1997.
[15] P. Szor, "Coping with Cabanas," in *Virus Bulletin*, 1997, pp. 10 - 12.
[16] P. Szor, "HPS (Hantavirus Pulmonary Syndrome)," in *Virus Bulletin*, 1998, pp. 13 - 15.
[17] P. Szor and P. Ferrie, "Hunting for Metamorphic," presented at Virus Bulletin Conference, 2001.
[18] G. Wroblewski, "General Method of Program Code Obfuscation," Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.