Abstract Stack Graph as a Representation to Detect Obfuscated Calls in Binaries

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Eric Uday Kumar

Fall 2004

Abstract Stack Graph as a Representation to Detect Obfuscated Calls in Binaries

Eric Uday Kumar

APPROVED:

_____	_____
Arun Lakhotia, Chair	William R. Edwards
Associate Professor of Computer Science	Associate Professor of Computer Science
_____	_____
Anthony Maida	C. E. Palmer
Associate Professor of Computer Science	Dean of the Graduate School

To Mom, Dad and dear brothers
Pradeep, Uttam

Acknowledgements

I thank my advisor, Dr. Arun Lakhotia for his valuable guidance, extraordinary support, inspiration, and encouragement. This thesis would never have been conceptualized without the ideas and motivation that he provided me. I greatly appreciate his patience and the trust he showed in me throughout my thesis. He was always there to support me both morally and technically whenever I was in a fix. My gratitude for him cannot be expressed in a paragraph.

I thank my parents and my brothers for their never-ending encouragement, trust, and support during my lows and my highs in the research period. I am grateful to Dr. Andrew Walenstein for reviewing my thesis document in a short time and giving me helpful insights. Thanks to Aditya Kapoor, Prashant Pathak, Enaam, Mohamed and Michael Venable who gave me helpful feedback during all times in my thesis and shared intriguing discussions about the challenges I faced. Thanks to Firas Bouz without whose timely help and guidance the implementation would not have materialized.

Lastly, I would also like to thank my friends Thomas Voitier, Jason Bourgeois, Charles Gravely, Jon Ziegler, Zeke Davy, Steven Sabatier and many others for speaking into my life the truth about God, encouraging me during these past two years to walk the path of truth, in spite of all odds. Their words of wisdom to trust God and seek Him to the fullest have shaped me into the person that I am now.

TABLE OF CONTENTS

List Of Figures

1 Introduction

1.1 Motivation

The highly interconnected world of computers ever poses the threat of malicious code. Such code can break into hosts using a variety of methods such as attacking known software flaws and vulnerabilities in regular programs. Hence detecting the presence of such malicious code on a given host is a problem of high concern. Whenever such hostile programs succeed in spreading over the internet, there is a significant loss to businesses. For example, mi2g website [1] quotes that within one quarter the NetSky worm and all it's A - Q variants put together, had already caused between $35.8 billion and $43.8 billion of estimated economic damages worldwide. The website also quotes that, in March, combined loss due to the three worms Beagle, MyDoom, and NetSky crossed the $100 billion mark within a week.

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) [14], [20], [33]. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners [11], [22], [31]. Most malicious code writers add or rearrange code in malicious programs to make their detection difficult, if not impossible. Recent virus writing trends that employ obfuscating transformations to conceal their behavior are the most difficult to detect. These viruses are called metamorphic viruses.

The primary goal of obfuscation is to increase the effort involved in manually or automatically analyzing a program. In the context of anti-virus scanning, the context of our study, automated analysis may be performed at the desktop, at quarantine servers in an enterprise, or on back-end machines of an anti-virus company's laboratory [27]. In contrast, manual analysis is performed by engineers in Emergency Response Teams of anti-virus companies and research laboratories. The goal of obfuscation in malicious programs—virus, worms, Trojans, spy wares, backdoors—is to escape detection by automated analysis and significantly delay detection by manual analysis.

A common obfuscation technique that is found in viruses, henceforth used generically to mean malicious programs, is that they obfuscate *call* instructions [31]. 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 (a) identify the kernel functions used by the program and (b) to identify procedures in the code. The obfuscation also takes away important cues that are used during manual analysis. We are then left only with dynamic analysis, i.e., running a suspect program in an emulator and observing the kernel calls it makes. Such analysis can easily be thwarted by what is termed as "picky" virus—one that does not always execute its malicious payload. In addition dynamic analyzers must use some heuristic to determine when to stop analyzing a program, for it may not terminate without user input. Virus writers can

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

1.2 Research Objectives

This aim of this research is to propose and implement 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.

1.3 Research Contributions

The main contribution of this thesis is a novel approach towards detecting obfuscated calls when the obfuscation is performed by using stack related instructions. The method uses abstract interpretation [17] wherein the stack instructions are interpreted to operate on an abstract 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. A method for constructing the abstract stack graph has also been presented. Application exploration via analysis of malicious programs has been done to bring forth the merits and limitations of this work.

1.4 Impact of the Research

The proposed detection technique may be used to improve manual and automated analysis tools, thereby raising the level of difficulty for a virus writer. The method can help by undoing some common obfuscation techniques. However, it is not claimed 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 [23]. The method presented here 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 situation 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, the solution does not model other memory areas, the content of the stack, and the content of registers. This deficiency may be overcome by combining this stack model with the Balakrishnan and Reps' method for analyzing the content of memory locations [8].

1.5 Organization of Thesis

Chapter 2 presents background work in this area. Chapter 3 presents the notion of an abstract stack and the abstract stack graph. Chapter 4 presents the algorithm to construct an abstract stack graph. Chapter 5 describes how the abstract stack graph may be used to detect various obfuscations. Chapter 6 discusses implementation and results. Chapter 7 presents conclusions and future work to develop a complete solution for detecting obfuscations. Appendix A describes the analysis of a virus called w32.evol. Appendix B presents the pseudo code for constructing an abstract stack graph along with an example followed by bibliography.

2 Background

This chapter outlines the process of static analysis of binary executables. It describes the application of obfuscating transformations with intent to thwart disassembly of binary executables and hide malicious code.

2.1 Static Analysis of Binary Code

Static Analysis is the automatic derivation of static properties that hold on every execution leading to a program point. It can be thought of as interpreting the program over an "abstract domain" and executing it over a larger set of execution paths. This helps to automatically obtain information about all executions of the program without really having to execute it for all possible inputs. *Static flow analysis* propagates estimates of actual values and these estimates are always conservative to uphold correctness. Since static flow analysis considers all (syntactic) program paths (both directions at every branch) it can be conservative or precise, but not both. It can be conservative in the sense that it might include some non-executable paths too. By doing this we can only obtain approximate results. Such approximation methods are particular cases of *abstract interpretations* of program semantics [17].

To extract meaningful information from a binary it is first disassembled, i.e., translated to assembly instructions [10], [18], [20], [25]. The assembler code is usually analyzed further, often following steps similar to those performed for decompilation [13] (see Vinciguerra et al. [32] for a survey of disassembly and decompilation techniques). Commercial antivirus groups are known to use disassemblers frequently to analyze the behavior of suspect programs [34]. Lakhotia and Singh [19] proposed a staged

architecture for binary malware analysis. This architecture is shown in Fig. 2-1. Disassembly is the first step, and it generally must occur before subsequent analysis can take place. Current disassemblers are used for different purposes such as rewriting binaries for efficiency [24], portability [12], [26], program maintenance when source code is not available and for detecting malicious programs. Lakhotia and Singh [19] 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. [20] present code obfuscation techniques for disrupting the disassembly phase, making it difficult for static analysis to even get started.

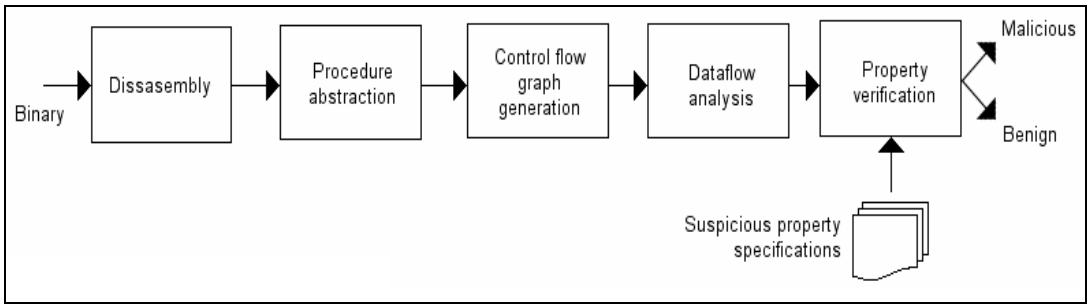


Fig. 2-1. Stages in static analysis of binary[19].

2.2 Code Obfuscation to Thwart Disassembly

A number of approaches have been proposed to make the reverse-engineering process harder [14], [20]. These techniques are based on transformations that preserve the program's semantics and functionality and, at the same time, make it more difficult for a reverse-engineer to extract and comprehend the program's higher-level structures. The process of applying one or more of these transformations to an existing program is called *obfuscation*.

Techniques used by code-obfuscators [21] for the purpose of protecting intellectual property can be used by malicious code writers. These writers also rely on many of the existing virus creation tools [24] and obfuscation techniques, such as junk byte insertion, statement reordering, call conversions, and opaque predicates, to hinder the disassembly process of the current algorithms [18], [20].

A major challenge in correctly disassembling malicious code is due to implications of the von Neumann architecture, where code and data are indistinguishable [16]. The problem is created by self-modifying code, where code is treated as data, and what was once data becomes executable. A disassembly algorithm could fail, either by incorrectly interpreting some instruction as data (false negative) or by incorrectly interpreting some data as an instruction (false positive).

Fig. 2-2 shows code of a variant of mass mailing worm Beagle.h. Column 1 shows the output of the open source debugger Ollydbg, while column 2 shows the desired disassembly. This is IA32 code which does not have fixed length instructions. At location 0040A001, opcode E8 is a 5 byte instruction code. The linear sweep algorithm for disassembly disassembles instructions byte-by-byte without regard to the control flow of a program. Due to this the algorithm assumes that the next instruction starts at 0040A006 and disassembles from there, interpreting the data as a 5 byte call instruction (*CALL E845648E,)* and so the next disassembled instruction starts at 0040A006 which is, in the correct disassembly, a junk byte. The junk byte is a code for a 5-byte instruction, which throws off disassembly because it assumes that the next instruction starts at 0040A00A when in reality it is supposed to start at 0040A007. Since E8 is an opcode for *call* instruction, it looks like a legitimate instruction starting at 0040A006. The virus writer

likely chose E8 since he has to insert an opcode that starts a valid instruction else linear sweep can raise alarms.

Location	Column 1 (Disassembly Ollydbg)		Column 2 (Actual Disassembly)	
	Hex	Disassembly	Hex	Disassembly
0040A000	60	PUSHAD	60	PUSHAD
0040A001	E8 01000000	CALL 0040A007	E8 01000000	CALL 0040A007
…				
0040A006	E8 83C404E8	CALL E845648E		
0040A007			83C4 04	ADD ESP,4
…				
0040A00A			E8 01000000	CALL 0040A010
0040A00B	0100	ADD DWORD PTR DS:[EAX],EAX		

Fig. 2-2. Obfuscation by junk byte insertion (Beagle.H).

Malicious code writers intentionally use a toolkit of similar tricks to try to defeat static disassembly. They use tricks like jumping into the middle of what appears to be a valid instruction or use computed jumps make it difficult to determine jump targets statically. Apart from junk byte insertion, some other obfuscation techniques as proposed by Linn et al [20] are:

- *Call conversion:* This obfuscation technique changes the return address of a call instruction. The program does not return to the instruction just after the call, rather it is manipulated to return at a predefined offset from the calling location. The bytes between the offset and location just after call can be filled by junk bytes to confuse a disassembler.
- *Opaque predicates:* In this technique the obfuscators can change all the unconditional *jumps* and *calls* to conditional *jumps* and *calls*. The branch that is always taken is known. Malicious code writers insert junk bytes at the location of branch that is never taken.

2.3 Obfuscation Game

Malware analysis can be described as an obfuscation-deobfuscation game between the malicious code writers and researches working on malicious code detection. The obfuscations are such that there is considerable change in the byte sequence of the executable obtained but does not change the program behavior i.e. the actual sequence of instructions being executed is retained. The aim of the malicious code writer is to fool the antivirus tool in believing that it is dealing with a safe executable. As malicious code writers try to induce newer obfuscating techniques to fool antivirus tools, the malicious code detectors hectically race to deobfuscate them.

Christodrescu et al. [11] presents a better understanding of these obfuscating transformations being employed by malicious code writers. Christodrescu tested the resilience of three popular commercial virus scanners against code obfuscation attacks. His results showed that these virus scanners could be subverted by very simple obfuscating transformations.

Code obfuscation techniques are increasingly being applied in enhancing software security [14], [20], [33] as well as in malicious code writing to evade detection [11], [22], [31]. Though the intentions of these two activities differ, there is no denying the fact that applications of these techniques toward prevention of malicious reverse engineering can be reused by virus writers to thwart detection of their malicious code, and vice versa.

2.3.1 Simple Code – Level Techniques

Collberg et al. [15] presents taxonomy of obfuscating transformations where a detailed theoretical description of various possible obfuscating transformations is presented.

Dead Code Insertion: Inserting code fragments that do not modify program behavior such as semantic *nop* insertion, adding zero to a register or an equivalent operation (such as *xor eax, eax*), jump/branch to the next instruction, instructions that modify dead registers, sequence of instructions that modify the program state, only to restore it back immediately such as: *add eax, 1* followed by *sub eax, 1*, are all examples of dead code insertion.

Register Renaming: This transformation replaces usage of one register with another in a specific live range. This technique exchanges register names and has no other effect on program behavior.

Instruction Reordering: The instructions are shuffled so that the order in the binary image is different from the execution order, or from the order of instructions assumed in the signature used by the antivirus software. To achieve the first variation, instructions are randomly reordered and unconditional branches or *jump* instructions are inserted to restore the original control flow. The second variation swaps the instructions if they are not interdependent to randomize the instruction stream.

Instruction Substitution: This obfuscation technique uses a dictionary of equivalent instruction sequences to replace one instruction sequence with another. This poses a tough challenge for automatic detection of malicious code. The Intel instruction set is rich and often provides several ways of performing an operation. For example, a

memory-based stack can be accessed both as a stack using dedicated instructions and as a memory area using standard memory operations. Hence, the Intel assembly language provides ample opportunities for instruction substitution. To handle obfuscation based on instruction substitution, an analysis tool must maintain a dictionary of equivalent instruction sequences, similar to the dictionary used to generate them. This is not a comprehensive solution, but it can cope with the common cases. For example: an instruction such as *test esi, esi* can be replaced by *or esi, esi*; an instruction *xor eax, eax* sets *eax* to zero and can be replaced by *sub eax, eax*.

Metamorphic viruses apply the above described obfuscation techniques to evade detection by anti-virus software. The common metamorphic transformations applied are dead code insertion, register renaming, code transposition (statement reordering or break & join transformations) and reshaping of expressions [22]. These transformations give birth to a new *variant* of the metamorphic virus. 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 [7], and Burneye (by TESO), which is a Linux binary encapsulation tool [2]. Other obfuscations exits, in particular *call* obfuscation.

2.3.2 Call Obfuscation

Recent virus writing trends heavily depend on making calls to kernel functions to infect, conceal and propagate [3], [4], [5], [6] [28], [29]. Calls being made to kernel functions may be used to determine whether the binary is malicious. For example

Symantec's Bloodhound technology uses classification algorithms to compare the set of calls being made by any program against a database of calls made by known viruses and clean programs [27]. Being aware of this approach, rogue programmers make such calls without using the *call* instruction [31]. 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 *ret* instruction transfers control to *addr*. Effectively a *call* is being made though the *call* instruction itself is not being used. This is instruction substitution obfuscation as described above. The code may be further obfuscated by splitting each instruction into multiple instructions. Obfuscation of *call* instructions breaks most static analysis based methods for detecting a virus since these methods depend on recognizing *call* instructions to (a) identify the kernel functions used by the program and (b) to identify procedures in the code.

2.4 Deobfuscation Game

Metamorphic viruses are particularly insidious in obfuscating their code to evade detection. Examples of 32-bit Windows metamorphic viruses are Win32/Regswap (created by Vecna in December 1998), Win32/Apparation, Win95/Zmorph (discovered in January 2000), Win95/Zperm (appeared in June 2000), and Win32/Evol (appeared in July 2000). Unlike polymorphic viruses, that create new decryptions using different encryption methods to encrypt the virus body, metamorphic viruses do not have a decryptor, nor a constant virus body. However, they are able to create new generations each time by applying obfuscating transformations to their code. They do not use a constant data area filled with string constants but have one single code body that carries

data as code. Hidden within the code are the various system call function names and parameters. The data would other wise have appeared in the data area. Most polymorphic viruses decrypt themselves to a single virus body in memory whereas metamorphic viruses do not.

The classic virus-detection techniques look for the presence of a fixed virus-specific sequence of instructions (called a virus *signature*) inside a program. If the *signature* is found, it is considered highly probable that the program is infected. This detection approach is effective when the virus code does not change significantly over time and the *signatures* chosen do not lead to false positives or too many false negatives. The *signature* is ideally chosen common to virus variants without increasing the false positive rates. A problem arises when virus writers obfuscate the virus code so that the fixed *signatures* used by the antivirus software cannot detect these obfuscated viruses anymore. To detect these obfuscated viruses, the virus scanners must first undo the obfuscation transformation used by the virus writers. A typical example is the metamorphic virus that modifies its own code [31]. Metamorphic viruses are particularly insidious because two copies of the virus do not have the same signature. Hence, they escape signature based anti-virus scanners [11]. Such viruses can sometimes be detected if the operating system calls made by the program can be determined. For example Symantec's Bloodhound technology uses classification algorithms to compare the set against a database of calls made by known viruses and clean programs [27].

2.4.1 Detecting Obfuscations

Existing static analysis can effectively detect simple obfuscations, like *nop*-insertion, by using regular expressions instead of fixed signatures [18]. The signature

must allow for any number of *nop*s at instruction boundaries. Most modern antivirus software use regular expressions as virus signatures. Some others use heuristic analysis and emulation techniques. As virus writers employ more complex obfuscation techniques, these malicious code detection techniques are bound to fail. There is a level of metamorphosis beyond which no reasonable number of strings can be used to detect the code that it contains. What is needed is a deeper inspection of malicious code based upon more sophisticated static analysis techniques. This appears to require the use of structures that are closer to the semantics of the code rather than mere syntactic techniques such as regular expression matching.

The antivirus technique usually applied to detect such kind of viruses is by emulating them using certain heuristics. Virus writers constantly come up with ways to foil the emulation techniques and make the analysis procedure difficult, if not impossible. Lakhotia and Singh [19] observe that though metamorphic viruses pose a serious challenge to anti-virus technologies, these virus writers are confronted with the same theoretical limitations and have to address some of the same challenges that the anti-virus technologies face. A recent result by Barak et al. [9] proves that in general program obfuscation is impossible. This in turn says that a computationally bounded adversary will not be able to obfuscate a virus to completely hide its malicious behavior. This is likely to have an effect on the pace at which new metamorphic transformations are introduced.

Indeed, research results in detecting obfuscated viruses are beginning to emerge. Christodrescu and Jha [11] use abstract patterns to detect malicious patterns in executables. Mohammed [22] has developed a technique to undo certain obfuscation

transformations, such as statement reordering, variable renaming, and expression reshaping.

The challenge, however, is in detecting the operating system calls made by a program. The Win32 standard 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*( ). Win32.Evol virus uses precisely this method for getting addresses of kernel functions and further obfuscates the method it uses to call these functions.

2.4.2 Using System Call Information for Detection

Many recent viruses heavily depend on calls to system libraries or kernel functions to conceal, infect and propagate. The Win95/Kala.7620 was one of the first viruses to use a system call to transfer control to the virus code (in this case, its decryptor) [30]. Modern anti-virus tools are to have system call 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. This export section is then used to locate the export info for *GetModuleHandle*( ) and *GetProcAddress*( ), to extract the correct entry point for these two functions and then use these functions to call any and all needed exported functions from any valid Windows module.

The achieve this, virus writers exploit internal facts of the underlying operating system such as: loading of *kernel32.dll* at the same starting address regardless of the version, revision etc.; *DLL* files are of the *PE* format and part of the *PE* format is an export table that the operating system uses to resolve what functions are in a module, where they are and how to call them; there are functions to call functions such as the *GetModuleHandle*( ) and *GetProcAddress*( ) class of functions that allow the exported entry point for any function in any module to be located. The strategy used by recent virus writers would be to locate *kernel32.dll*'s *PE* header in memory and use the header info to locate *kernel32*'s export section [30].

Normally, an API import happens by using the name of the API such as *FindFirstFileA*( ), *FindNextFileA*( ), *GetSystemDirectoryA*( ) *OpenFile*( ), *ReadFile*( ), *WriteFile*( ), *GetFileAttributesA*( ), *SetFileAttributesA*( ), etc. used by many first generation viruses. A set of suspicious system call name strings will appear in non-encrypted Win32 viruses. This can make the disassembly of the virus much easier and potentially useful for heuristic scanning. For example a portion of code from the virus Win95.z0mbie reads:

1. *call _GetCommandLineA*
2. *SW_NORMAL equ 1*
3. *push SW_NORMAL*
4. *push eax*
5. *call _WinExec*

As can be seen from this listing, Win95.z0mbie uses the address at line 1 to determine its command line path and then loads it once again through the *WinExec* function (this is essentially spawning a copy of self).

A trick used to thwart disassembly and heuristic match appears in various modern viruses is to hide the use of system call name strings to access particular kernel functions from the Win32 set. For example, the Win32/Dengue virus does not use system call name strings to access particular kernel functions [30]. Modern 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 kernel function is found. Hence, the absence of system call name strings should not be inferred as non-existence of any system calls in the program.

2.4.3 Obfuscating System Calls to Evade Detection

Some antivirus tools can detect malicious code by identifying the calls being made to kernel functions. To evade this detection process, some metamorphic viruses obfuscate these calls. 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. The techniques used by malicious code writers to implement this are centered on instruction substitution.

2.4.4 Call Obfuscation in Win32.Evol

Win32.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.

Instead of a *push* the virus may use a *mov* that modifies the stack pointer to point to the address of the Windows API function to be called and *ret* transfers control to the function. Some of the Win32 API functions that the virus makes use of are: *CreateFileA*( ), *CreateThread*( ), *FindFirstFileA*( ), *FindNextFileA*( ), *GetCommandLineA*( ), *GetDriveTypeA*( ), *GetWindowsDirectoryA*( ), *LoadLibraryA*( ). The address of the API functions is looked up form the entry points or addresses within kernel32.dll using another Win32 API function 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*( ) by constructing the name of the string of the function being called 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.

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 virus 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*. This value is pushed on the stack and control is transferred to this kernel function by executing a *ret* instruction. A detailed analysis of the virus code can be found in Appendix A.

3 Abstract Stack Graph

Detection of call obfuscation in malicious code requires the ability to statically monitor the stack. In this chapter we discuss the notion of an abstract stack which is an abstraction of the program's stack as it would appear while executing. The chapter also introduces a concise way of representing all possible abstract stacks at each program point; this compact representation is called an abstract stack graph. Examples are used to clarify the discussion.

3.1 The Abstract Stack

An abstract stack is an abstraction of the actual/concrete stack that might be observed on a running program. The actual stack of a program keeps actual data values that are pushed and popped in a LIFO (Last In First Out) sequence. The abstract stack instead stores the addresses of the instructions that push values in a LIFO sequence. For example, consider Fig. 3-1. Each instruction in the sample program is marked with its address from *L1* through *L4*. The actual stack and the abstract stack, after execution of the instruction at address *L4*, are as shown in Fig. 3-1.

Sample Program	*Concrete Stack*	*Abstract Stack*
L1: push eax	…	…
L2: push ebx	...	…
L3: pop edx	*Eax*	*L1*
L4: push ecx	*Ecx*	*L4*
	Top of stack	*Top of stack*

Fig. 3-1. Concrete and abstract stacks.

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. 3-2 shows a sample program; its control flow graph appears in Fig. 3-3.

```
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. 3-2. Sample program.

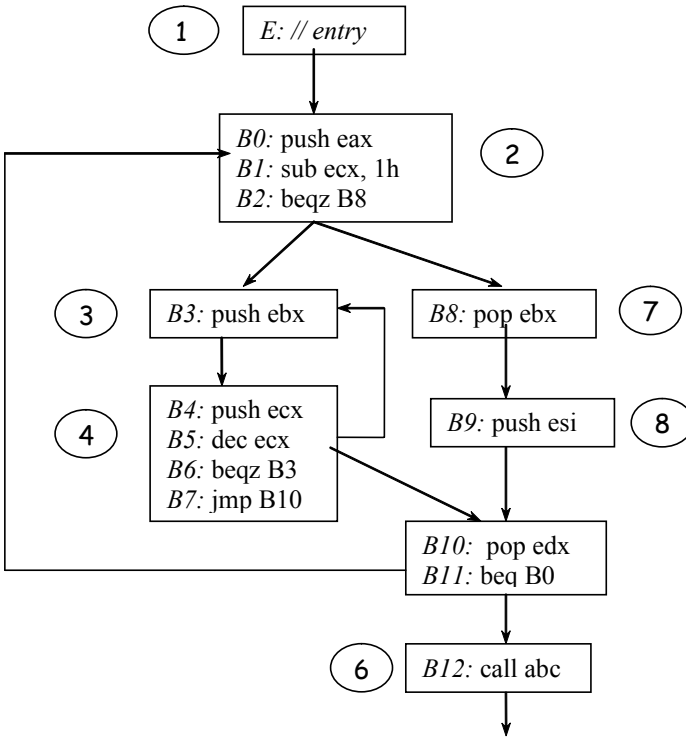


Fig. 3-3. Control flow graph for sample program.

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. 3-4 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 → 2 → 3 → 4 → 3 → 4 → 5 → 2. The abstract stack shown at program point 4 results from the trace 1 → 2 → 3 → 4 → 3 → 4 → 5 → 2 → 3 → 4. The execution trace 1 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 3 → 5 → 2 → 7 → 8 yields the abstract stack at program point 8.

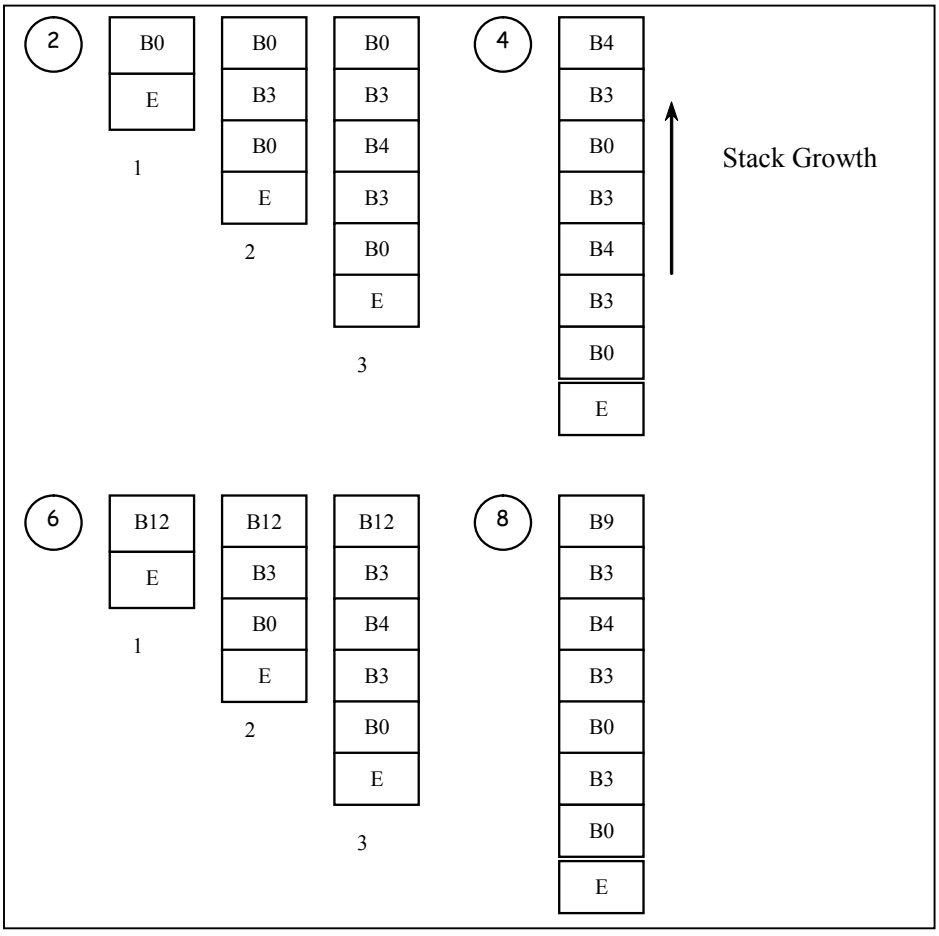


Fig. 3-4. Possible abstract stacks at some program points.

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. 3-4. In fact, program points 3 and 4 may have infinite number of abstract stacks. This is because there is a loop between program points 3 and 4 and the loop contains unbalanced *push*, i.e., a *push* that is not matched with a *pop*. A more efficient way to handle all possible abstract stacks at each program point is required.

3.2 The Abstract Stack Graph

An abstract stack graph is a concise representation of all, potentially infinite number of, abstract stacks at all points in the program. Fig. 3-5 shows the abstract stack graph for the example program in Fig. 3-2. A path (sequence of nodes beginning form the abstract stack top towards the bottom) in the graph represents a specific abstract stack.

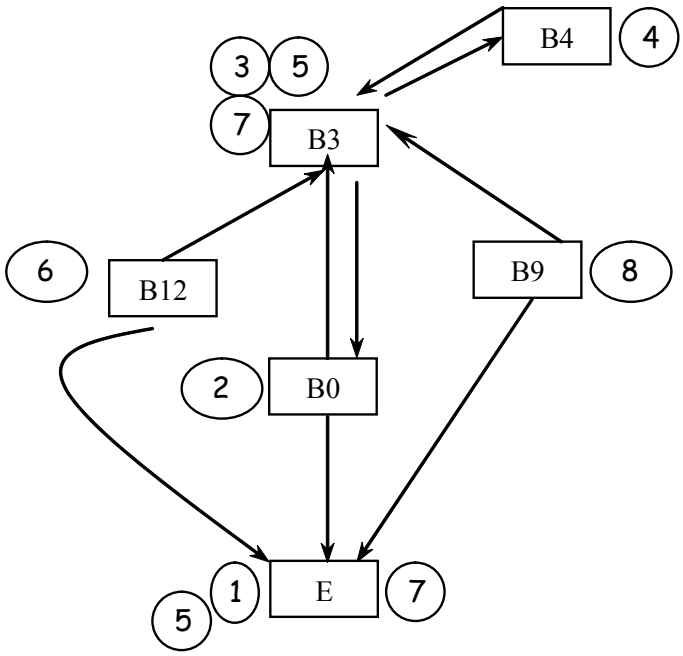


Fig. 3-5. Abstract stack graph for sample program.

4 The ASG Domain

As discussed in the previous chapter, an abstract stack graph is a data structure to efficiently represent all possible abstract stacks at each program point. In abstract interpretation one needs to define a domain and evaluation function. These are defined in this chapter and an *O(n)* algorithm for constructing an abstract stack graph.

4.1 Defining the ASG 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, **[** m: *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. 3-3 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. 3-3, all points in a block receive the same top of stack. In Fig. 3-5, *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.

4.2 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.

4.2.1 Evaluation Function

Fig. 4-1 presents an evaluation function $\mathcal{E}$ 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; *PRIM*

next: $ADDR \rightarrow ADDR$, returns the address of the instruction executed after the instruction at the parameter.

inst: $ADDR \rightarrow INST$, returns the instruction at the address.

isvalidcall: $ADDR \rightarrow$ Boolean, returns true iff the instruction at the address is a *call* instruction.

The evaluation function $\mathcal{E}$ takes in two parameters of type *INST* and *ASG* and outputs an element of *ASG*. This is denoted by $\mathcal{E}$: $INST \rightarrow ASG \rightarrow ASG$. For example, $\mathcal{E}$**[** 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.

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

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

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

$\mathcal{E}$ **[** m: *ret* **]** *asg* =
$\quad$ $\cup$ $\quad$ $\mathcal{E}$ *n* (*abspop* m *asg*)
$\quad$ *n* $\in$ *absret asg*

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

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

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

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

Fig. 4-1. Evaluation function.

Now we can, loosely speaking, say that *ASP* and *ASPR* are related as follows: Let $\mathcal{E}$ **[** m: *inst* **]** *asg* = (*N*, *AE*, *ASP*), then (*m*, *a*) $\in$ *ASPR* where *a* $\in$ *ASP*. The evaluation function determines what operations in *PRIM* are to be applied, and the next instruction to be interpreted. The next section defines these abstract operations.

4.2.2 Abstract Operations

Fig. 4-2 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*.

$$
\begin{aligned}
&\mathit{abspush}: \mathit{ADDR} \rightarrow \mathit{ASG} \rightarrow \mathit{ASG} \\
&\mathit{abspush}\ \mathrm{m}\ (\ N, AE, ASP\ ) \\
&\qquad = (\ N \cup \{\ \mathrm{m}\ \},\\
&\qquad\quad AE \cup \{\ \mathrm{m} \rightarrow\ \mathrm{asp} \mid \mathrm{asp} \in \mathit{ASP}\ \},\\
&\qquad\quad \{\ \mathrm{m}\ \}\\
&\qquad\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abspop}: \mathit{ASG} \rightarrow\ \mathit{ASG} \\
&\mathit{abspop}\ \mathrm{m}\ (\ N, AE, ASP\ ) \\
&\qquad = (\ N,\\
&\qquad\quad AE,\\
&\qquad\quad \{\ \mathrm{x} \mid \mathrm{a} \in \mathit{ASP}, (\mathrm{a} \rightarrow \mathrm{x})\ \in AE\ \}\\
&\qquad\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{absret}: \mathit{ASG} \rightarrow \wp\, \mathit{ADDR} \\
&\mathit{absret}\ (\ N, AE, ASP\ ) \\
&\qquad = \{\mathit{next}(x) \mid a \in \mathit{ASP}, (a \rightarrow x)\ \in AE,\\
&\qquad\qquad \mathit{validcall}(x)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reset}: \mathit{ADDR} \rightarrow\ \mathit{ASG} \rightarrow\ \mathit{ASG} \\
&\mathit{reset}\ \mathrm{m}\ (N, AE, ASP\ ) \\
&\qquad = (\ N \cup \{\ \mathrm{m}\ \},\\
&\qquad\quad AE,\\
&\qquad\quad \{\ \mathrm{m}\ \}\\
&\qquad\ )
\end{aligned}
$$

$$
\begin{aligned}
&i: \mathit{ASG} \rightarrow\ \mathit{ASG} \\
&i\ (\ N, AE, ASP\ ) \qquad = (\ N, AE, ASP\ )
\end{aligned}
$$

Fig. 4-2. Abstract operations.

Operation *abspush* pushes a new address on the abstract stack. It is used in the evaluation of the *call* and *push* instructions. These two instructions are representative of

instructions that perform the push operation. Other instructions may be modeled similar to these instructions. For example, the *INT* (software interrupt) instruction may be modeled like the *call* instruction. Instructions that increase the content of stack by directly manipulating the stack pointer, such as *sub esp*, 8*h*, are modeled using the push instruction.

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

Operation *absret* supports the evaluation of the *ret* instruction. It checks whether the address at the top of 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 function identifies such obfuscations.

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.

4.2.3 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 <*Ø*, *Ø*, *Ø*> 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. 4-1. A sketch of the algorithm follows; a complete pseudo is in Appendix B.

Assume that the disassembly and an entry point to the code are available. The current abstract stack graph is initialized to <*Ø*, *Ø*, *Ø*>. The assembly instructions are then interpreted one by one. 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_Inst*, 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 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. Hence, nodes in the graph are finite since instructions in the program space are finite. This implies that the abstract stack graph is finite. Also, since each instruction is interpreted only once, the algorithm to construct the abstract stack graph is linear in time and space.

5 Detecting Obfuscations

This chapter shows how an abstract stack graph may be used to detect stack related obfuscations. The obfuscations detected are:

- Call obfuscation
- Parameter passing obfuscation
- Return obfuscation

For each detection, example programs are used to illustrate the mechanism. They show the effective real/concrete stack at a program point of interest as well as the abstract stack graph at that point. 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 flow. 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

A *call* to a procedure within the same segment is termed a "near" *call* and performs the following: it decrements the stack pointer (*esp*) by a word and pushes the instruction pointer onto the stack; the *eip*, contains the offset of the instruction following

the *call*. Next it inserts the offset address of the called procedure into *eip*. The semantics of a *call addr* instruction may be defined operationally as follows:

1. Push the memory address of the byte after the current instruction onto the stack.
2. Assign the address *addr* to the instruction pointer (*eip*).

The concrete stack at beginning of program point *L0* is as shown in Fig. 5-1.

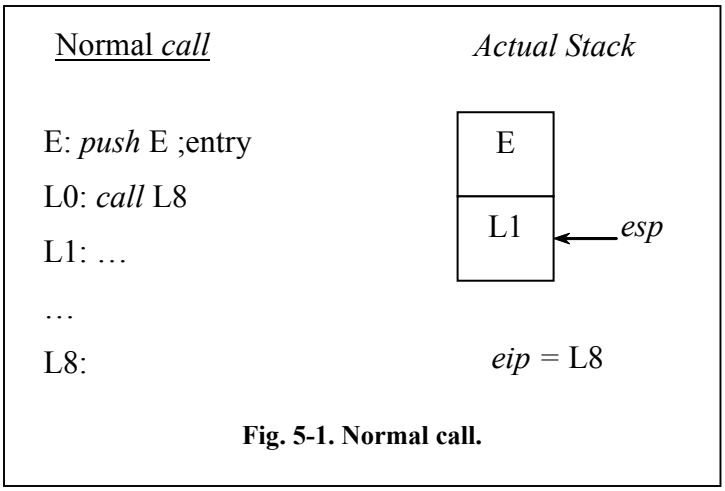


Fig. 5-1. Normal call.

5.1.1 Obfuscation using *push/jmp*

Fig. 5-2 shows a program that simulates a *call* using a combination of *push* and *jmp* instructions. The *jump* through a register transfers control to *L8*. Before the *jump* is executed, the offset of the instruction following the *call* is pushed onto the stack. The instruction at *E* pushes the entry point of the code onto the stack. The instruction at *L0* pushes the offset address of the instruction following the *call*, which is the return address, onto the stack. The instruction at *L1* loads the effective address of the instruction at *L8* into *eax* and then the instruction at *L2* jumps to this address. When *ret* is encountered at *L9*, the control returns to the return address previously pushed onto the stack. Hence,

without using the *call* instruction itself, the same functionality is achieved here as is intended in Fig. 5-1.

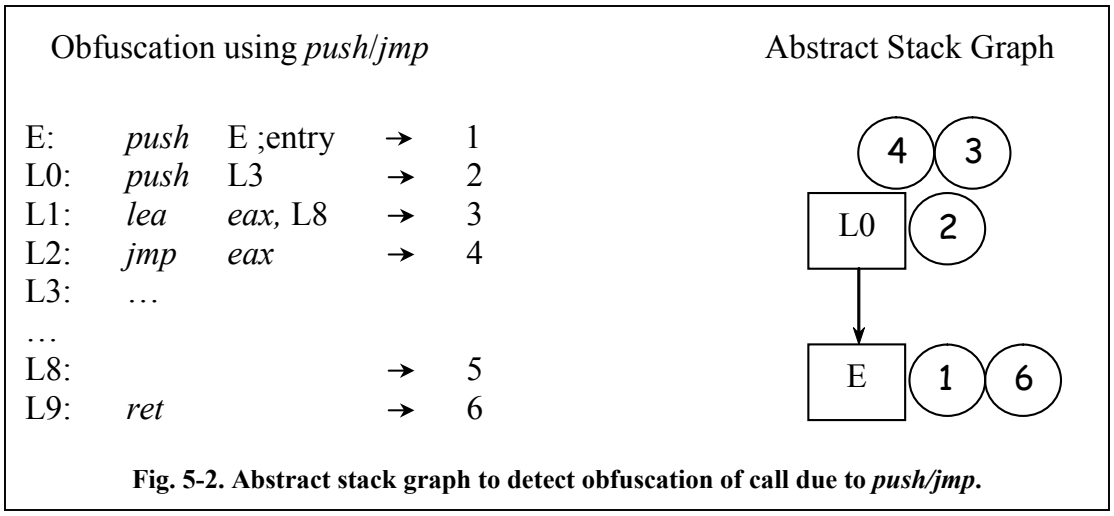


Fig. 5-2. Abstract stack graph to detect obfuscation of call due to *push/jmp*.

That the *jmp* instruction actually performs a *call* becomes known from the abstract stack graph at the entry point of the call. When an address is not known to be the entry point of a procedure, the abstract stack graph at the *ret* instruction, program point 6, discloses the obfuscation. During normal execution the top of the stack at this program point contains the return address *L3* pushed by the *push* instruction at label *L0*. In the abstract stack graph, the top of stack at program point 6 is *E*. That the *ret* instruction is returning from an obfuscated call is detected because *E* is not the address of a *call* instruction.

5.1.2 Obfuscation using *push/ret* or *push/pop*

Fig. 5-3 shows two different types of obfuscations of a *call*. They differ in how control is transferred to the target address. In the first, the target address is pushed on the stack and a *ret* instruction pops this address from the stack and transfers control. In the second, the target address is pushed on the stack, it is then popped into a register, and an indirect jump is performed to the address in the register. The labels and program points in

the two examples have been chosen such that both examples have the same stack and abstract stack graph. In both examples the actual transfer of control is done at instruction labeled *L3*, i.e. at program point 5.

Obfuscation using *push*/*ret*

```
E:   push  E ;entry  →  1
L0:  push  L4        →  2
L1:  lea   eax, L8   →  3
L2:  push  eax       →  4
L3:  ret             →  5
L4:  …
…
L8:                  →  6
L9:  ret             →  7
```

Obfuscation using *push*/*pop*

```
E:   push  E ;entry  →  1
L0:  push  L5        →  2
L1:  lea   eax, L8   →  3
L2:  push  eax       →  4
L3:  pop   ebx       →  5
L4:  jmp   ebx       →  6
L5:  …
…
L8:                  →  7
L9:  ret             →  8
```

Abstract Stack Graph

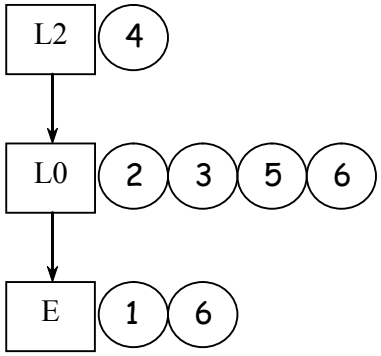


Fig. 5-3. Abstract stack graph to detect obfuscation of call due to *push/ret* or *push/pop*.

The instruction at *L3*, where a *ret* or a *pop ebx* is done is the address that was previously pushed onto the stack by instruction at *L2* and is the target of the control transfer. In the first example this is a *ret* instruction and in the second example it is *pop ebx*. The top of the abstract stack at program point 5 contains *L0*, the address of the instruction that pushed the target address on the stack. Thus, once again, when a *ret* statement is encountered (at program point 7 in case of first example and at program

point 8 in case of second example) it can be determined that it was reached due to an obfuscated call.

Now the *push* instruction itself can be substituted by a sequence of instructions that eventually achieve the same semantics. For example, in Fig. 5-2 the instruction at address *L0* which is *push L3* (assume *L3* is a 4 byte address) can be substituted by the following sequence of instructions:

mov ebp, esp

sub esp, 4

mov [ebp - 4], L3

In such cases where the stack operation itself is further obfuscated by instruction substitution, the abstract stack graph cannot be used to detect the obfuscation since it is limited to observing the evaluations of only those operations that can be mapped to stack *push* and *pop* instructions, where each is performed as a unit operation. It cannot model situations where the *push* and *pop* instructions themselves may be decomposed into multiple instructions.

5.2 Detecting Obfuscated Parameters

When analyzing a program for malicious behavior it is often useful to know the parameters being passed to a function. A program may be deemed malicious depending on the parameter. For instance, 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 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, the order is changed to match. At the entry point, the parameter addresses are connected by compensating for the pushed return address. Fig. 5-4 contains a sample normal code. In this program, the arguments to the function are pushed immediately before the *call* instruction.

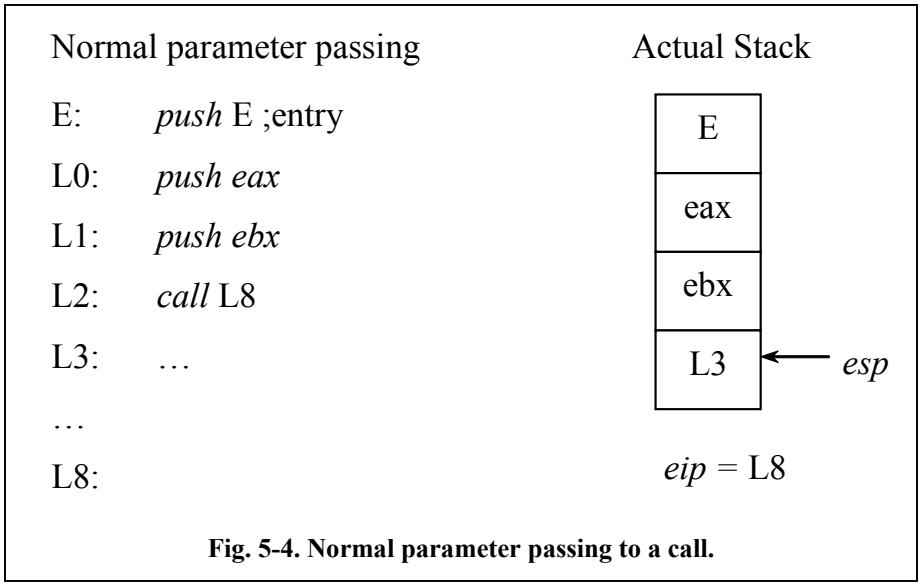


Fig. 5-4. Normal parameter passing to a call.

5.2.1 Obfuscation using Out of Turn *push*

Fig. 5-5 contains an example of what is termed as "out-of turn push". Instructions at *L0* and *L1* push parameters in registers *eax* and *ebx* onto the stack. These are intended to be parameters to *call L6*, but they are pushed before the instruction *call L4*. This gives the appearance that the parameters are being passed to the function at *L4*. The abstract stack graph for the program can be used to detect where the parameters to a function are

assembled. At program point 6, immediately after *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 location where the parameters for the function are pushed.

Obfuscation using Out of turn *push*

```
E:    push E ;entry  →   1
L0:   push   eax     →   2
L1:   push   ebx     →   3
L2:   call   L4      →   4
L3:   call   L6      →   6
L4:   …
L5:   ret            →   5
L6:   …
L5:   ret            →   7
```

Abstract Stack Graph

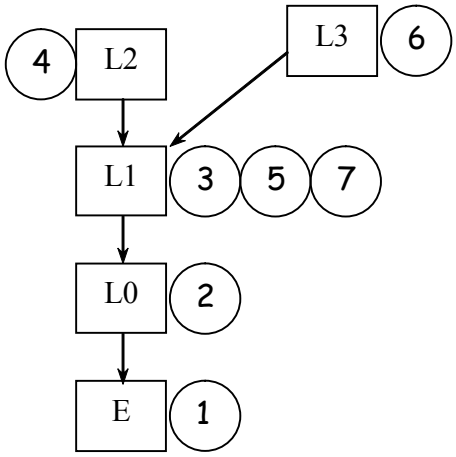


Fig. 5-5. Abstract stack graph to detect obfuscation of parameters due to out of turn *push*.

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 5, the *ret* instruction, the top of the abstract stack contains *L2*. Thus the *ret* instruction will be seen to return from a call made by the *call* instruction at address label *L2*. Now at program point 6, 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, the top of the

abstract stack contains *L3*. Thus, the *ret* instruction will be seen to return from the call made by the *call* instruction at address label *L3*.

5.2.2 Obfuscation using Redundant *push/pop*

Introducing redundant push and pop instructions can obfuscate the parameters. Consider the program in Fig. 5-6. 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

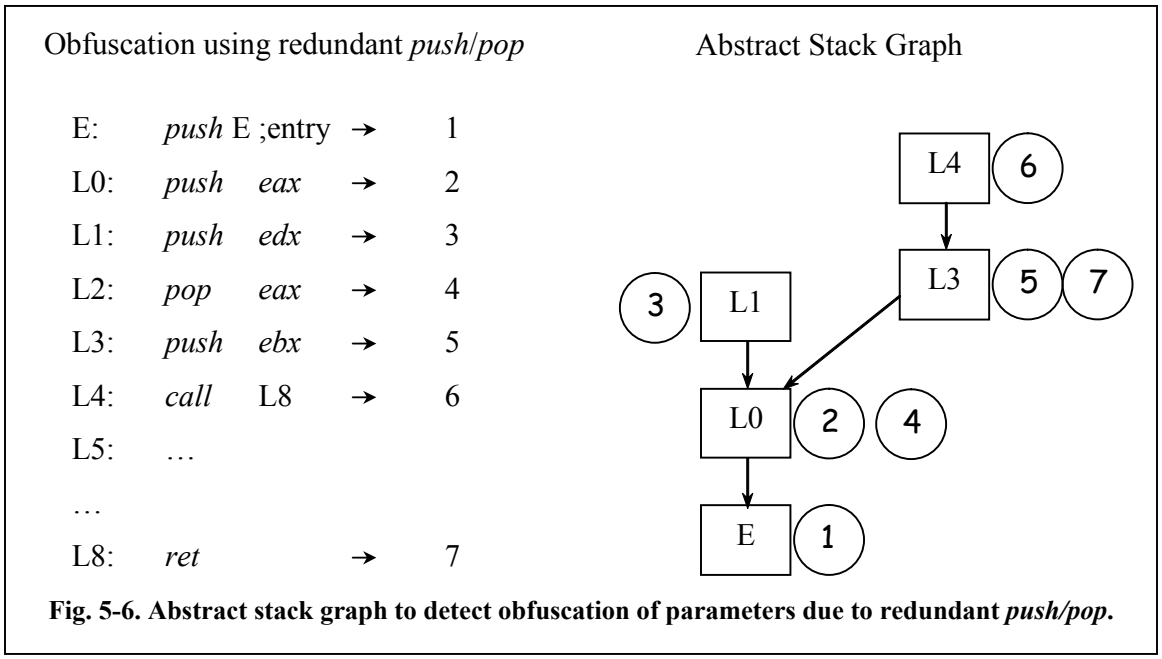


Fig. 5-6. Abstract stack graph to detect obfuscation of parameters due to redundant *push/pop*.

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.

5.2.3 Obfuscation due to Redundant Control

Fig. 5-7 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 to logically follow one direction, i.e., either it is always taken and never falls through, or it is never taken and always falls through. 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 instruction at *L0* results in *edx* containing zero. Hence the instruction at *L1* always evaluates to *true* and the branch is taken to *L3*.

Obfuscation using redundant control

```
E:    push  E ;entry   →  1
L0:   xor   edx edx    →  2
L1:   beqz  L3         →  3
L2:   jmp   L5         →  4
L3:   push  eax        →  5
L4:   push  ebx        →  6
L5:   call  L8         →  7
…
L8:   ret              →  8
```

Abstract Stack Graph

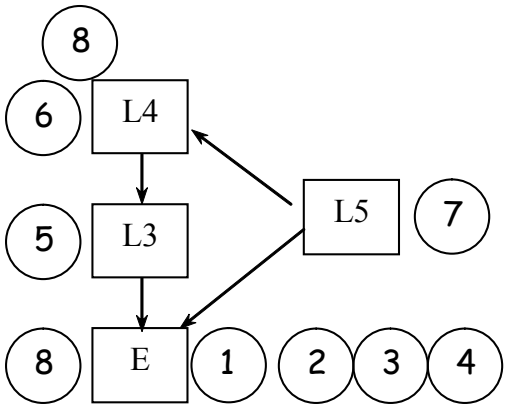


Fig. 5-7. Abstract stack graph fails to detect obfuscation due to redundant control.

The abstract stack graph shown here cannot be used to detect this redundant control unless we are able to compute contents of registers or memory locations. This example shows the limitation of the abstract stack graph method where a model for retrieving contents of registers is required to detect the obfuscation. At program point 7, where the *call* to *L8* is made, the abstract stack state could be L5|L4|L3|E or L5|E. This means the branch at *L1* could either be taken or not taken. But, if we were to determine the contents of register *edx* at *L0*, then we can determine that the branch at *L1* is always

taken. This information would render a new abstract stack graph in which the edge from node *L5* to *E* would no longer be present. The only abstract stack state possible would be L5|L4|L3|E which includes the address of the instructions *L3* and *L4* that push the arguments for *call* at *L5*, hence detecting the redundant control.

5.3 Detecting Obfuscated *ret*

A *ret* statement typically pops the top of the stack and returns control to address it pops which is basically reversing a *call*'s steps. 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. 5-8. After *ret* is executed, control transfers to the instruction immediately after the *call*.

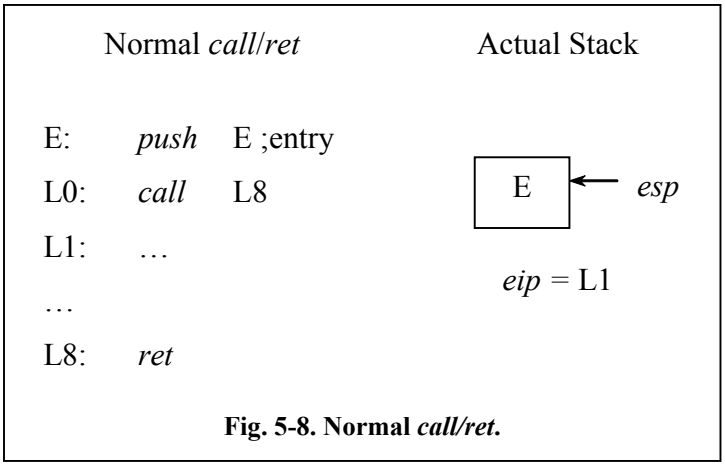


Fig. 5-8. Normal *call/ret*.

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 original *call* instruction. The two we detect are pop to return and return elsewhere.

5.3.1 Using *pop* to return

In the example in Fig. 5-9, the effect of a *ret* instruction is achieved by popping an address at the top of stack into a register and jumping it. The abstract stack at program

point 2 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.

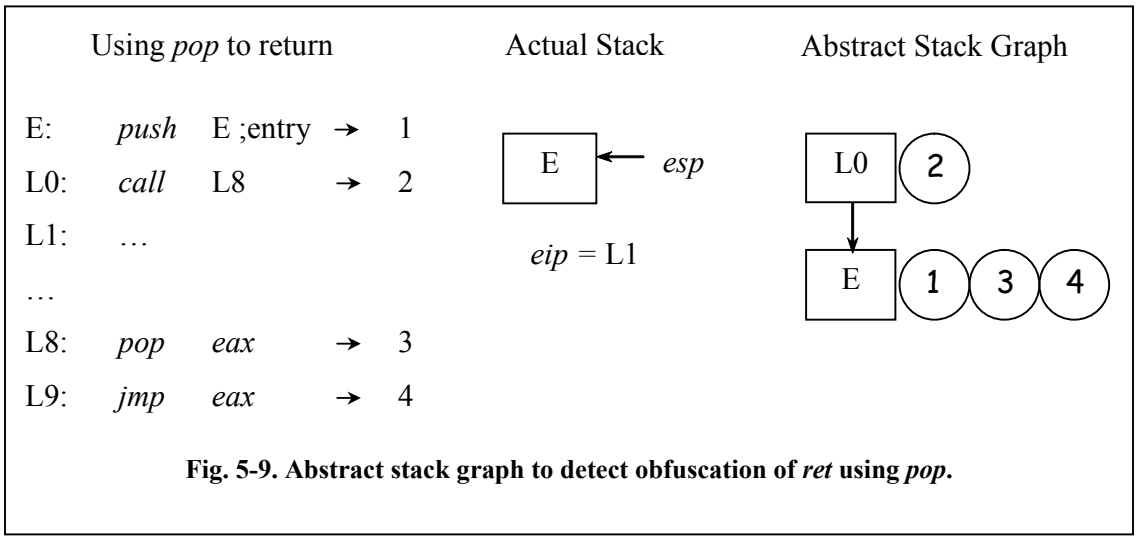


Fig. 5-9. Abstract stack graph to detect obfuscation of *ret* using *pop*.

5.3.2 Returning elsewhere

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. 5-10, the instruction at *L0* makes the *call* to *L3*. Immediately after the *call* instruction, 2 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 2 bytes to the return address to generate a new return address so that the *ret* instruction transfers control to 2 bytes after the original return address. This is obfuscating *ret* to return elsewhere. The abstract stack graph may be augmented to detect this obfuscation. Along with each location in the stack an additional tag, *modified*, may be maintained. When a

value is pushed on the stack, *modified* is set to *false*. If an instruction may change the contents of the stack, and we can determine the stack offset that is being changed, then we can change the tag of that location to *modified*. If the value at the top of the stack at a *ret* instruction is modified, it implies that *ret* is returning elsewhere.

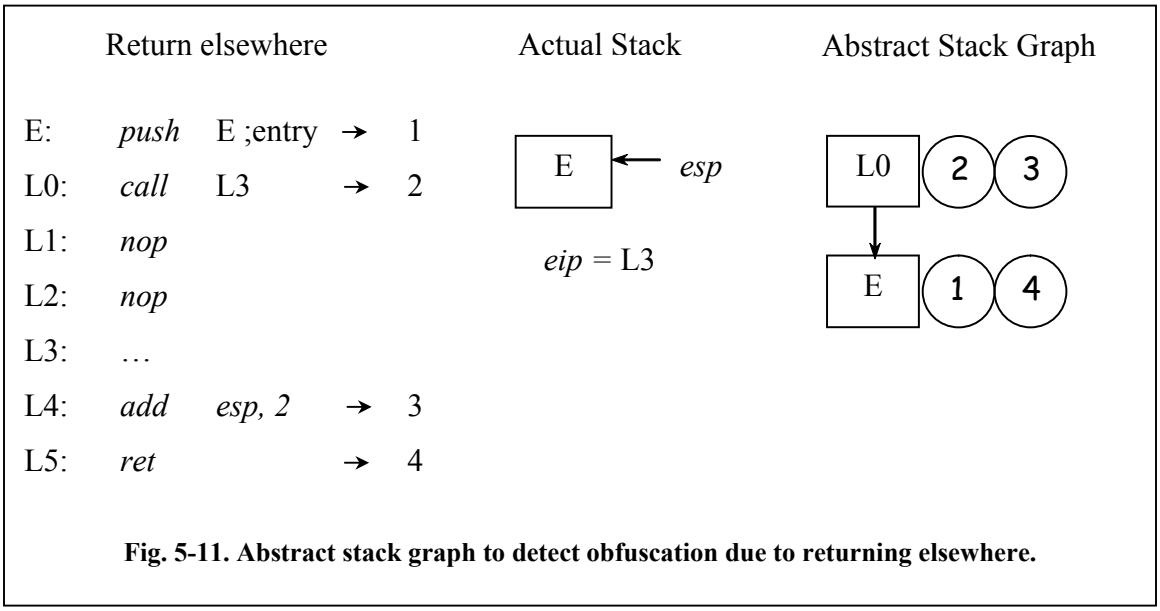


Fig. 5-11. Abstract stack graph to detect obfuscation due to returning elsewhere.

Another method of obscuring the *ret* instruction is by using *branch functions*. A branch function does not behave like "normal" function in that it typically does not return control to the instruction following the *call* instruction, but instead branches to some other location in the program that depends, in general, on where it was called from. Given such a branch function, an unconditional branch in a program (a *jump* instruction) can now be replaced by a *call* to the branch function. Branch functions serve two distinct purposes. The first is to obscure the flow of control in the program by sufficiently obscuring the computation of the target address within the branch function. The second is to create opportunities for misleading the dissembler by inserting junk bytes at the point immediately after each *call* of the branch function.

The branch function takes an argument and a return address from the callee. The return address is the address of the instruction immediately following the *call*. The callee passes, as an argument to the branch function, the offset from the return address to the eventual target address of the branch function. The branch function adds the value of its argument to the return address, so that the return address becomes the address of the original target. The code for this is as follows:

xchg eax, esp

add [*8 + esp*]*, eax*

pop eax

ret

The first instruction exchanges the contents of register *eax* with the word at the top of the stack, effectively saving the contents of *eax* (this is required because if any of the condition code flags is live at the call point, they have to be saved by the caller just before the call, and restored at the target) and at the same time loading the displacement to the target (passed to the branch function as an argument on the stack) into *eax*. The second instruction adds up this displacement to the return address (which resides a word below the top of the stack) and the result is placed back on the stack. The third instruction restores the previously saved value of *eax*, and the fourth instruction has the effect of branching to the address computed by the function and now placed on top of the stack. This too is the case of obfuscation by returning elsewhere and can be detected using the abstract stack graph as discussed above.

5.3.3 Abusing Call

The *call* instruction can also be "abused" to jump to a particular instruction. In Fig. 5-12, 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 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 stack 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*.

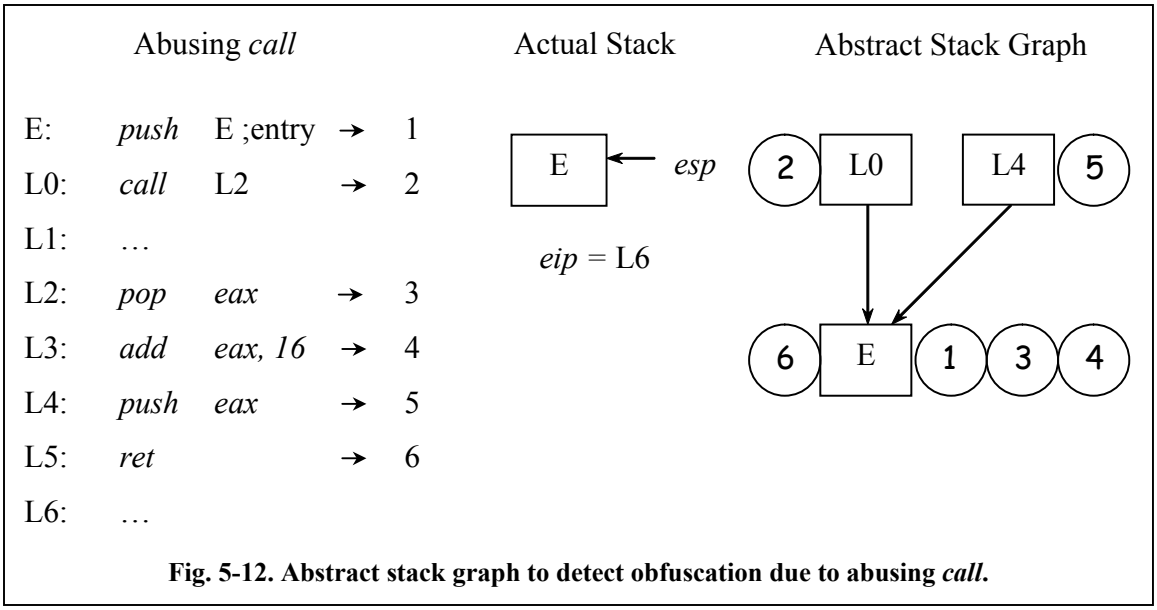


Fig. 5-12. Abstract stack graph to detect obfuscation due to abusing *call*.

6 Implementation and Results

This chapter presents a demonstration of the proposed algorithm and highlights its limitations. A prototype tool, DOCs (Detector for Obfuscated Calls) has been implemented using the Eclipse 2.1 framework. The goal is to demonstrate the use of DOCs as a means to detect call obfuscations in known virus programs (w32.evol).

6.1 DOCs Implementation details

DOCs has been implemented using the Eclipse 2.1 framework [35]. The Eclipse Platform is designed for building integrated development environments (IDEs) that can be used to create diverse applications. It is an open platform for tool integration built by an open community of tool providers. The Eclipse Platform is built on a mechanism for discovering, integrating, and running modules called *plug-ins*. A tool provider writes a tool as a separate plug-in that operates on files in the workspace and surfaces its tool-specific UI in the workbench. When the Platform is launched, the user is presented with an IDE composed of the set of available plug-ins.

DOCs has been implemented as a plug-in and is hence extendable. It is written in the Java programming language. A screen shot when the plug-in is used to open an assembly file (.asm extension) is shown in Fig. 6-1.

6.2 Capabilities of DOCs

DOCs provides the ability to open any number of projects at the same time. The navigator view helps to browse and open files in a project which are displayed in the file view. DOCs takes as input an assembly file, and constructs an abstract stack graph on user selection. The user can now select an option from the choices view to detect obfuscated calls, obfuscated returns, *call-ret* sites and manipulated call sites (same as detecting obfuscated returns).

6.3 Demonstration with test programs

DOCs was used with a few sample assembly files to detect the following:

- Valid *call-ret* sites
- Non-contiguous *call-ret* sites
- Obfuscated calls
- Obfuscated returns

6.3.1 Detecting valid *call-ret* sites

Fig. 6-1 shows a screen shot in which a test assembly file is opened in the file view and the option to detect *call-ret* sites has been selected. The instructions highlighted in red within the file view show valid call-ret sites. The numbering "(0)" at the end of these highlighted lines denotes that for the *call* site at address 101F, the *ret* site is at address 110B.

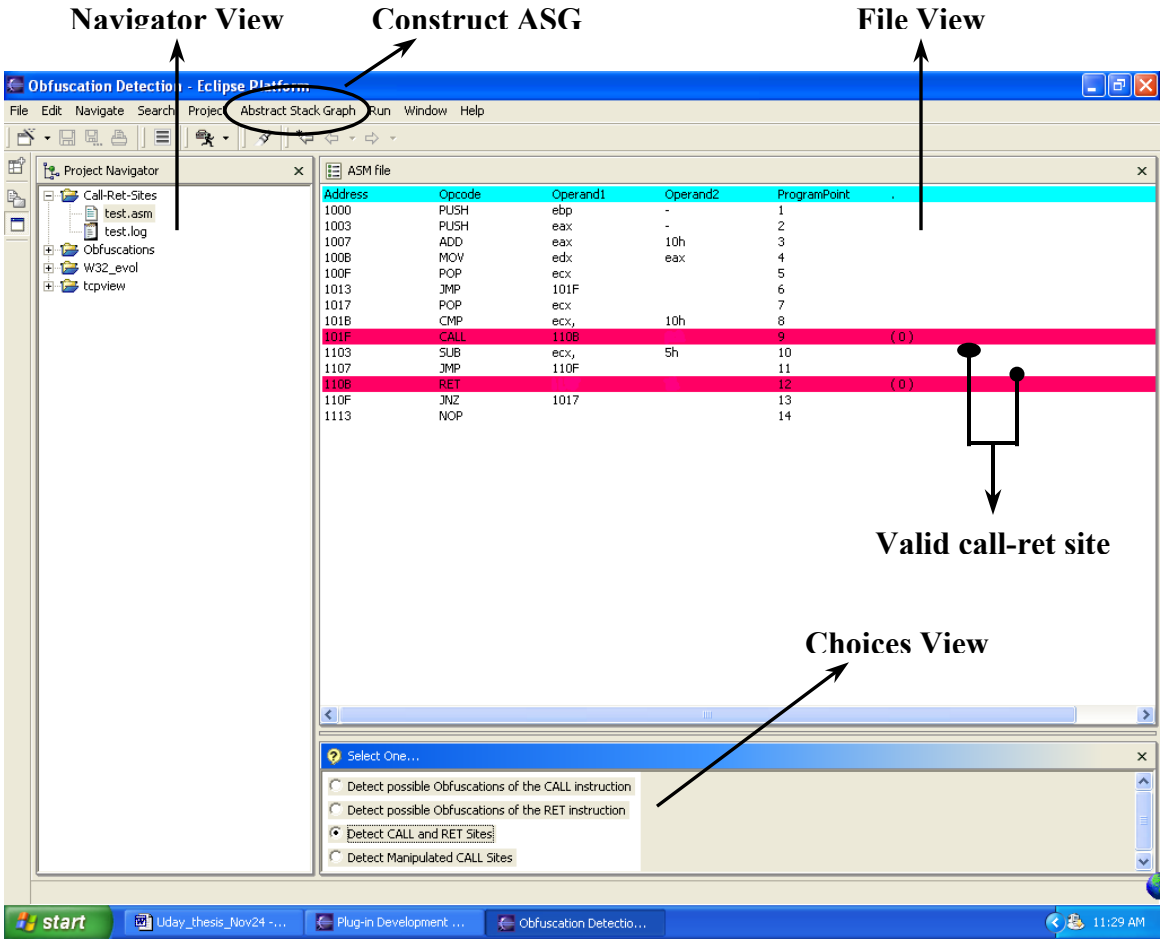


Fig. 6-1. Detecting valid call-ret sites.

In order to flag the instruction addresses as valid *call-ret* sites requires construction of the abstract stack graph which has been constructed for the selected assembly file and is as shown in Fig. 6-2. It is evident from Fig. 6-2 that when program point 12 is reached, the *ret* instruction is returning from a node which is a *call* instruction and hence is a valid *call-ret* site.

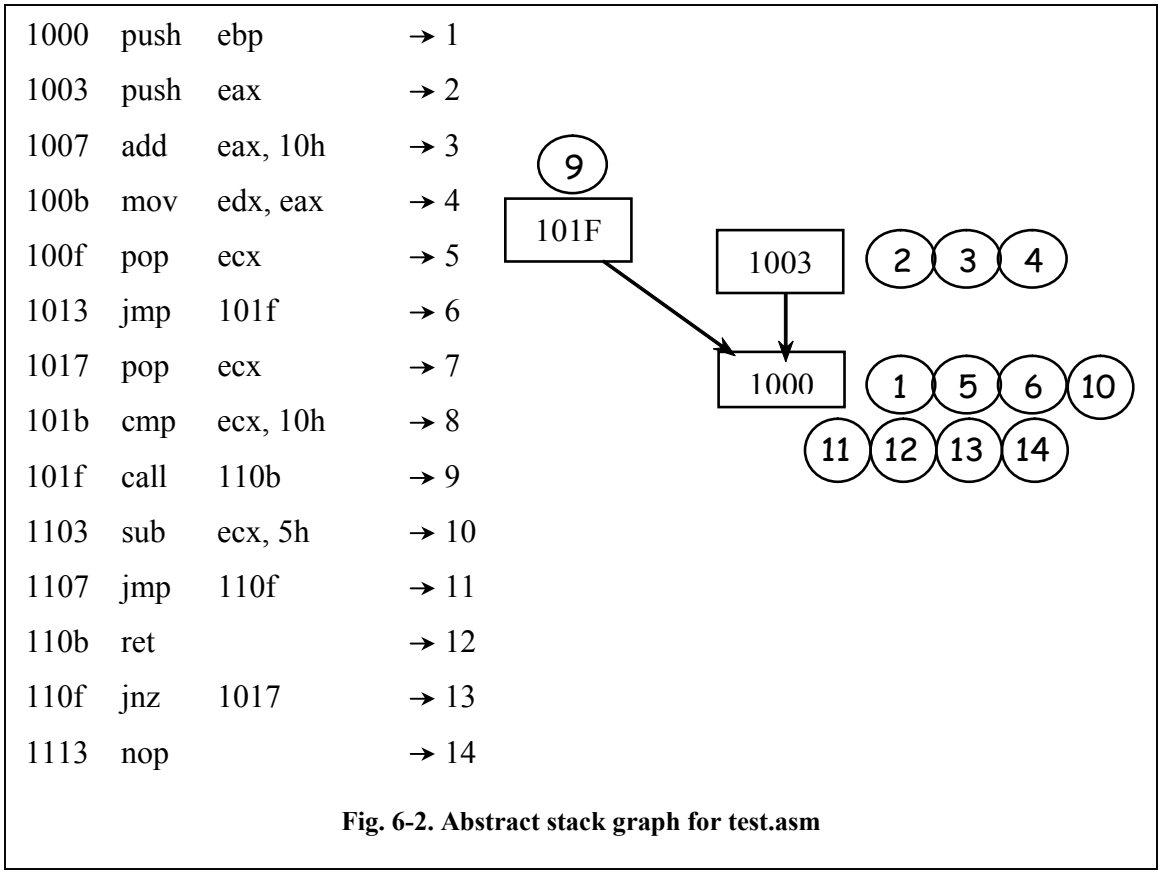


Fig. 6-2. Abstract stack graph for test.asm

6.3.2 Detecting non-contiguous *call-ret* sites

DOCs can also detect non-contiguous *call-ret* sites as shown for a sample assembly file in Fig. 6-3. The entry point of the code begins at address 300f. At address 3014 a *call* to address 3000 is made. Within this code, at address 3004 now a *call* to address 2000 is made, and so on. For each of these calls their corresponding *ret* sites lie before the *call* site. The *call* at address 3014 finally returns at address 300B. 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. A usual linear scan would have rendered incorrect *call-ret* sites.

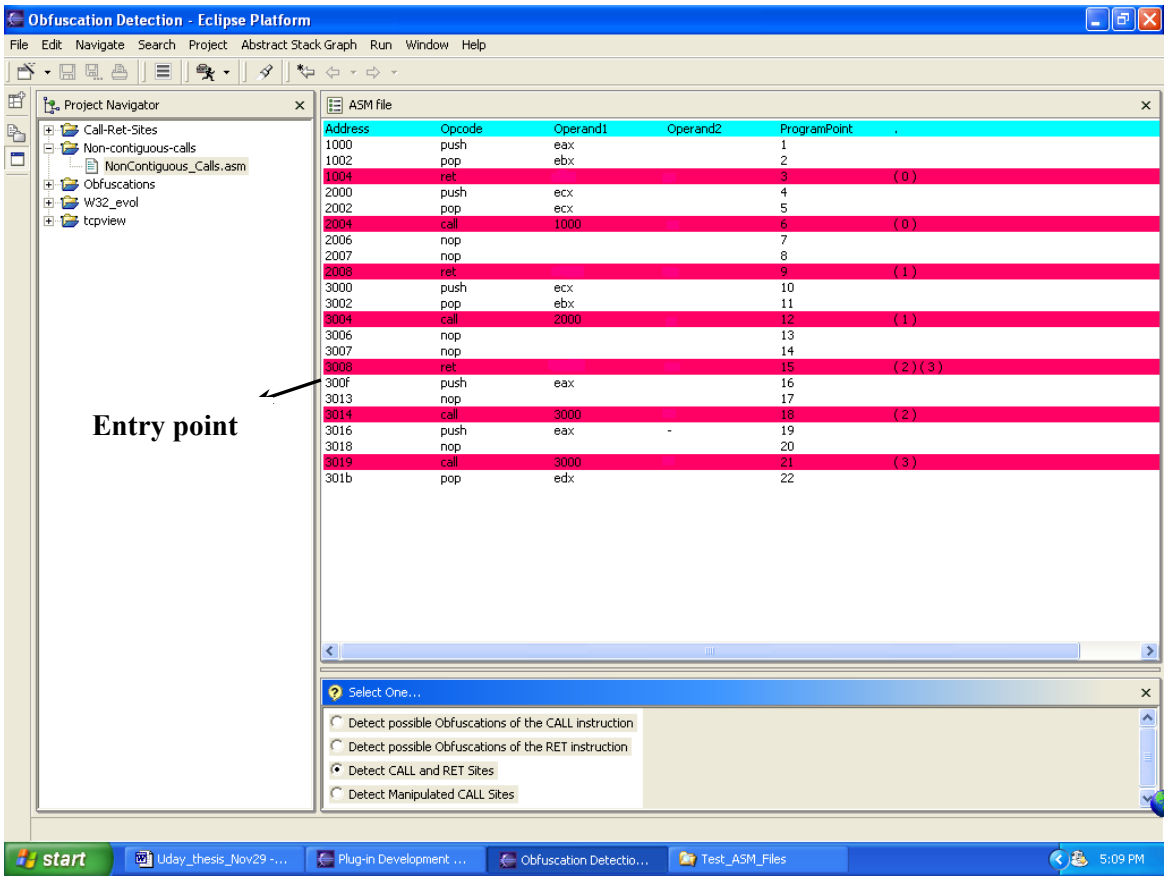


Fig. 6-3. Detecting non-contiguous *call-ret* sites.

6.3.3 Detecting obfuscated calls

Fig. 6-4 shows a screen shot in which another assembly file (test1.asm) is opened in the file view and the option to detect obfuscated calls has been selected. Instructions highlighted within the file view show possible obfuscation of the *call* instruction. The *ret* site at address 200F is associated with a *push* at address 2000 and the *ret* site at address 2213 is associated with the *push* at address 2017. The entry point of the code begins from the first instruction at address 1fff. A simple linear scan performed on this code would render incorrect *entry-exit* blocks. For example, a popular disassembler, IDAPro, marks the instructions between addresses 1fff and 200f as comprising a block of code while the instructions between addresses 2103 through 2213 are marked as another block of code.

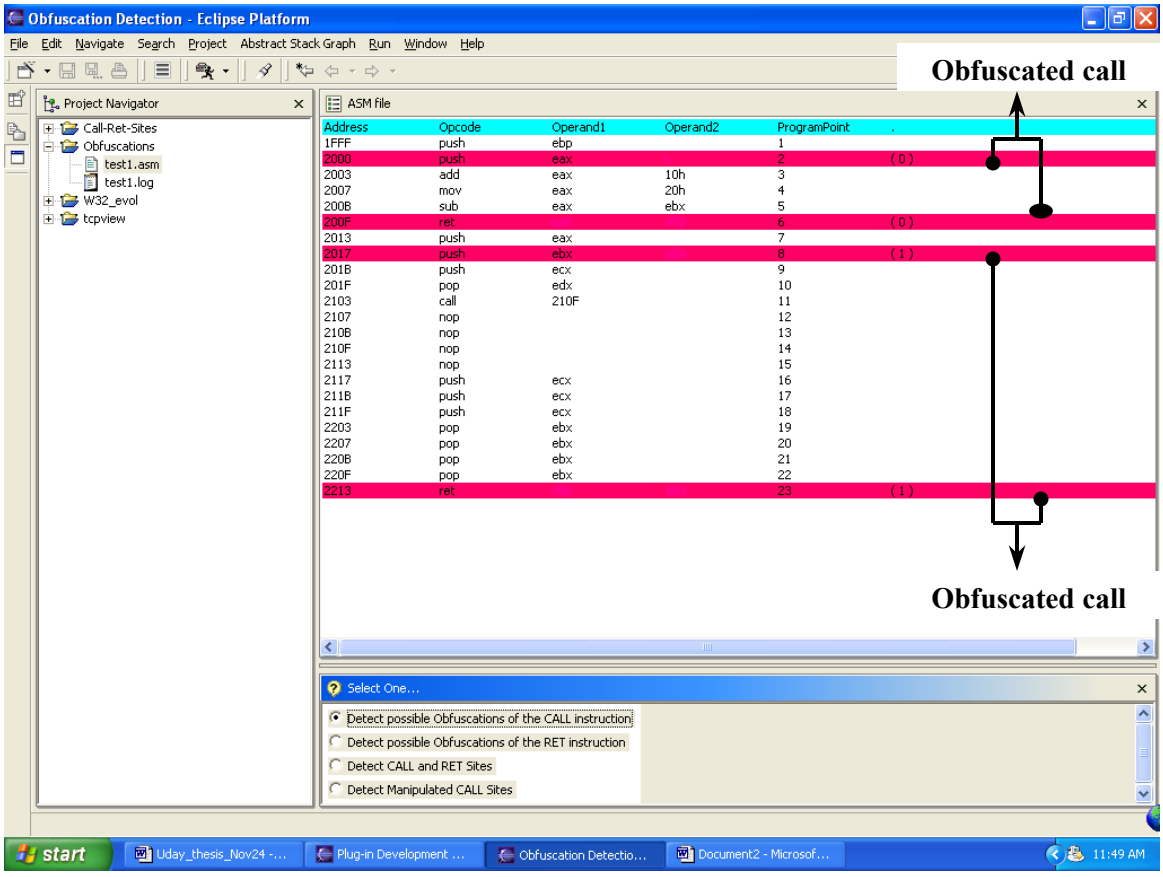


Fig. 6-4. Detecting possible obfuscations of the *call* instruction.

In order to flag the instruction addresses as obfuscated *call* requires construction of the abstract stack graph which has been constructed for the selected assembly file *test1.asm* and is as shown in Fig. 6-5. From the abstract stack graph when program points 6 and 23 are reached, the *ret* instruction is returning from a node which is a *push* instruction hence detecting obfuscated calls.

```
1FFF  push  ebp          → 1
2000  push  eax          → 2
2003  add   eax, 10h     → 3
2007  mov   eax, 20h     → 4
200B  sub   eax, ebx     → 5
200F  ret                → 6
2013  push  eax          → 7
2017  push  ebx          → 8
201B  push  ecx          → 9
201F  pop   edx          → 10
2103  call  110F         → 11
2107  nop                → 12
210B  nop                → 13
210F  nop                → 14
2113  nop                → 15
2117  push  ecx          → 16
211B  push  ecx          → 17
211F  push  ecx          → 18
2203  pop   ebx          → 19
2207  pop   ebx          → 20
220B  pop   ebx          → 21
220F  pop   ebx          → 22
2213  ret                → 23
```

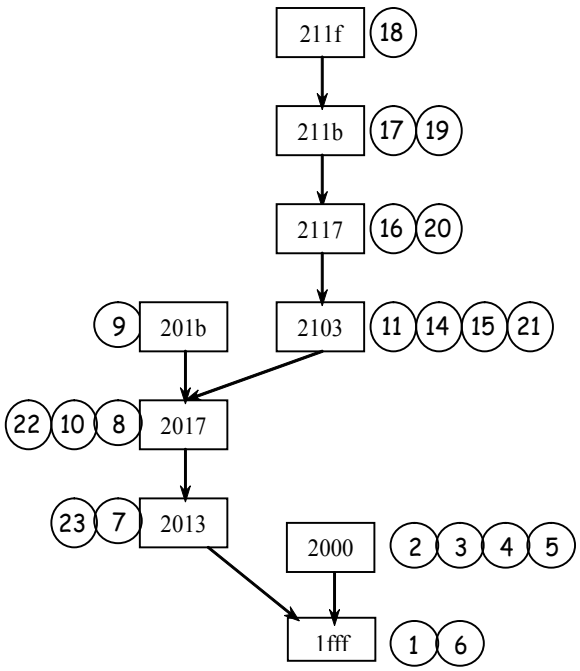


Fig. 6-5. Abstract stack graph for test1.asm

6.3.4 Detecting obfuscated returns

Fig. 6-6 shows a screen shot in which the same assembly file *test1.asm* is opened in the file view and this time the option to detect obfuscated returns has been selected. Instructions highlighted in red within the file view show possible obfuscation of the *ret* instruction. The *pop* instruction at address 220F is associated with a *call* instruction at address 2103. This information too is retrieved from the abstract stack graph shown in Fig. 6-5. From the abstract stack graph when program point 22 is reached, the *pop* instruction is popping from the node which is a *call* instruction at address 2103 hence detecting the obfuscated *ret*.

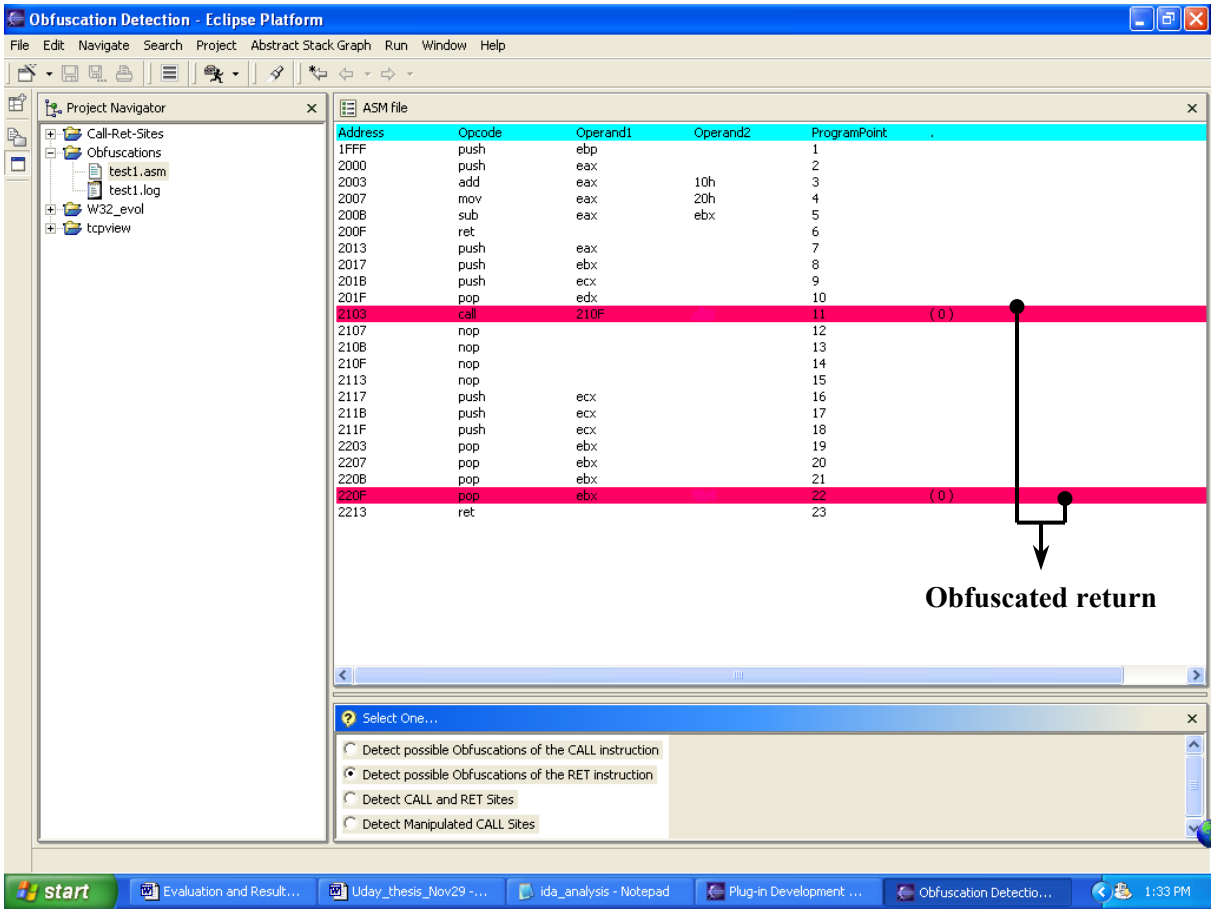


Fig. 6-6. Detecting possible obfuscations of the *ret* instruction.

6.4 Demonstration with w32.evol virus

In case of sample test programs the prototype tool successfully detected all possible obfuscated calls, returns, valid *call-ret* sites and non-contiguous *call-ret* sites. To demonstrate its use for detecting obfuscations in real virus code, it was used to detect possible obfuscations in w32.evol virus. The prototype tool detected 20 possible obfuscated calls and 2 possible obfuscations of the return instruction. This has been verified by manual analysis of the virus, prior to detection by the tool. Fig. 6-7 shows a screen shot in which two of the possible obfuscated *call* sites are highlighted. The *call* to

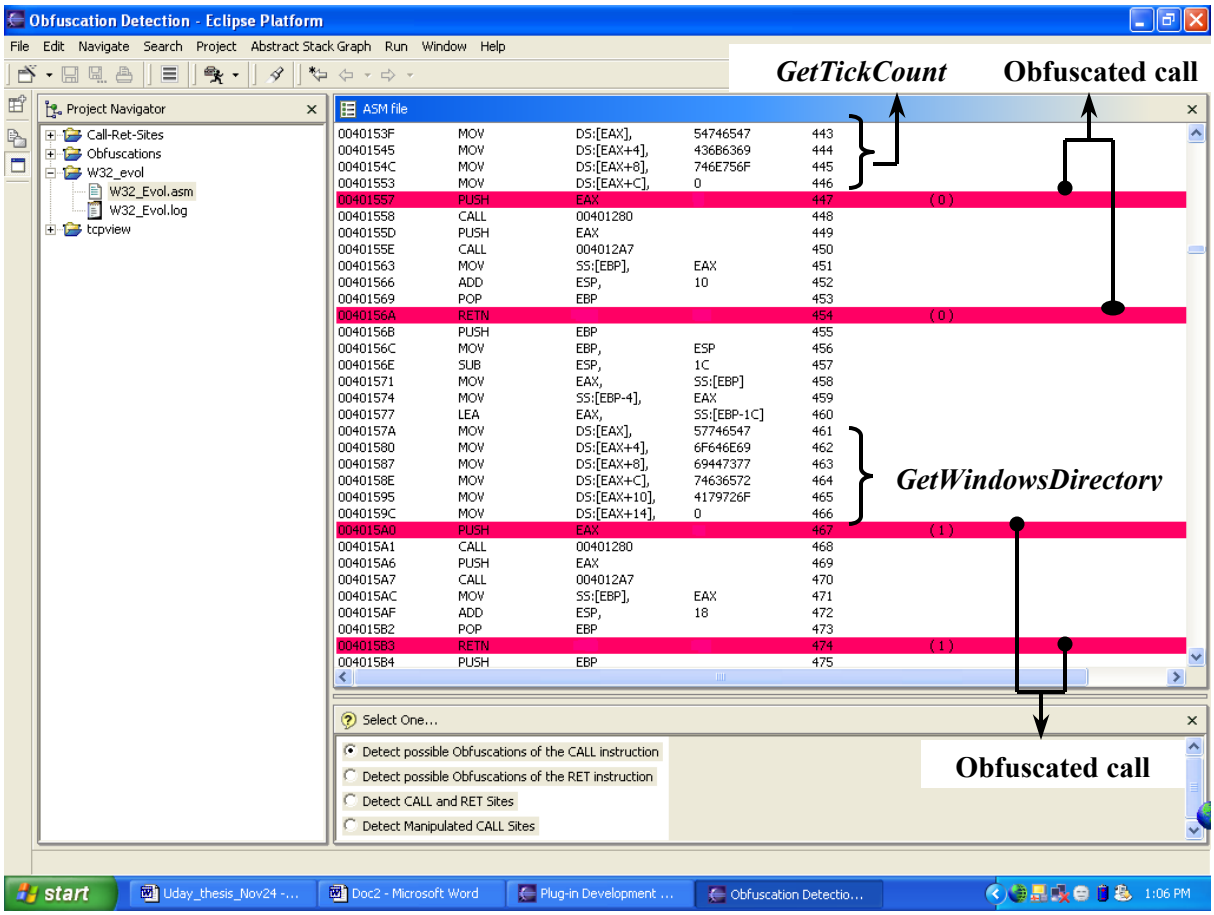


Fig. 6-7. Detecting possible obfuscations of the call instruction in w32.evol.

two of the kernel functions *GetTickCount*( ) and *GetWindowsDirectory*( ) is obfuscated and Fig. 6-7 shows these strings being pushed on the stack. The process by which control is transferred to each of these kernel functions becomes clearer from Fig. 6-8. A detailed explanation can be found in Appendix A.

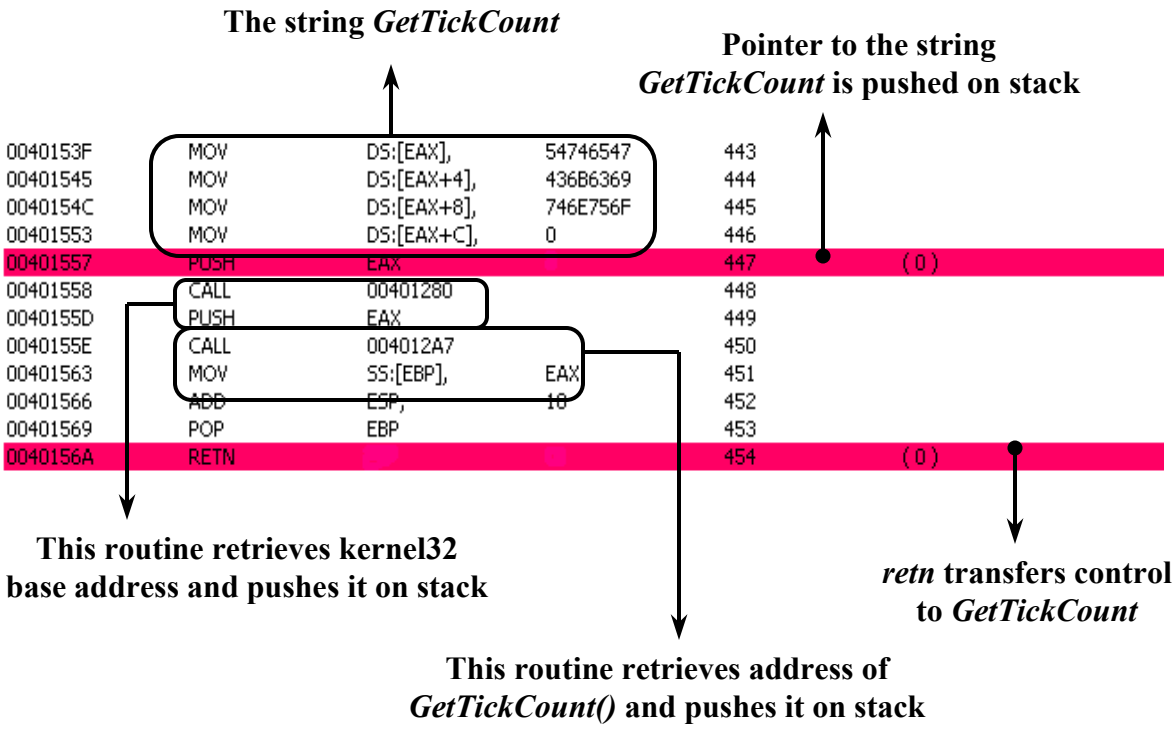


Fig. 6-8. Obfuscation of *call* to kernel function *GetTickCount* ( )

The instructions from addresses 0040153F through 00401557 push a pointer to the string "*GetTickCount*" on the stack which is the required kernel function intended to be called. The instruction at address 00401558 calls a routine which retrieves kernel32 base address and saves it in register *eax*. The instruction at address 0040155D stores this value on the stack. The stage is now set to call a kernel function called *GetProcAddress*( ) which takes as parameters the string "*GetTickCount*" and kernel32 base address that have previously been pushed on the stack. This kernel function retrieves the procedure address

(in this case the address of *GetTickCount*) and saves it in register *eax*. The instruction at address 0040155E calls a routine that in turn makes a *call* to *GetProcAddress*( ) to retrieve the address of *GetTickCount*( ) in register *eax*. The instruction at address 00401563 stores this value on the stack. The instructions at addresses 00401566 and 00401569 adjust the stack pointer to point to this address and the *retn* instruction at address 0040156A transfers control to *GetTickCount*( ).

Fig. 6-9 shows a screen shot in which the two possible obfuscated *ret* sites are highlighted. Each of the *call* instructions at addresses 004017AA and 004017C1 do not have a matching *ret* instruction but rather are popped by a *pop* instruction immediately following each.

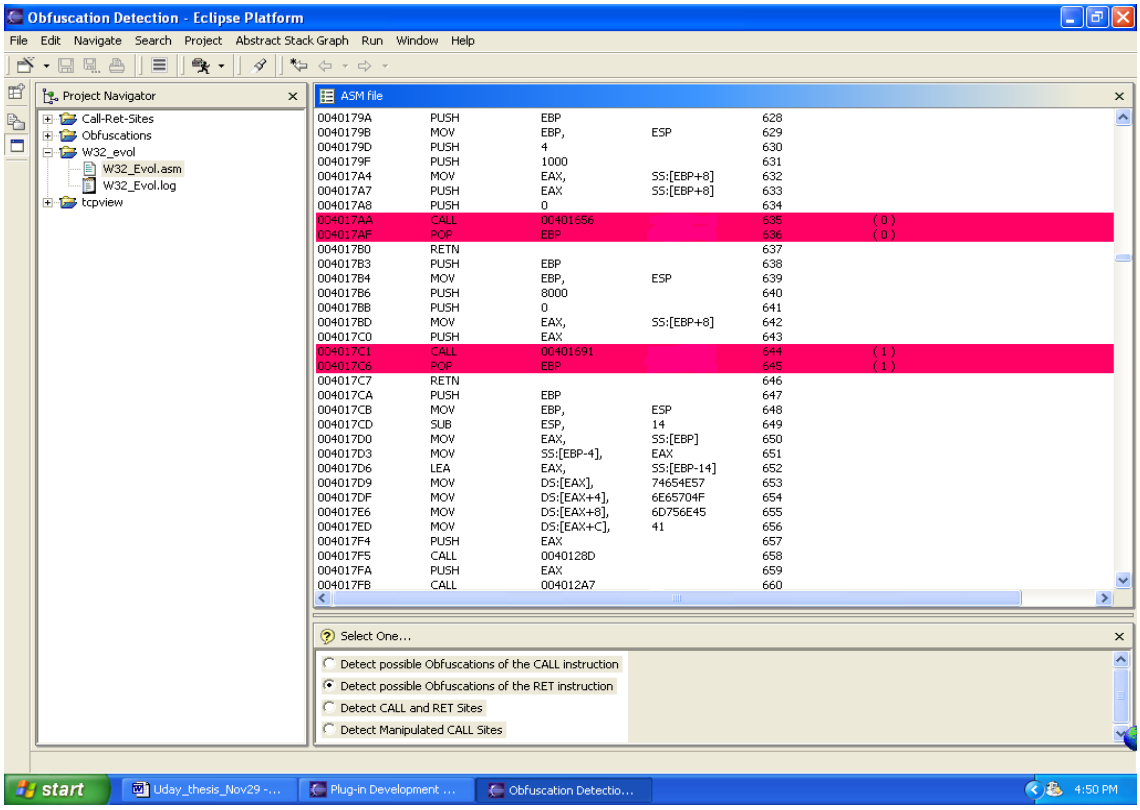


Fig. 6-9. Detecting possible obfuscations of the *ret* instruction in w32.evol.

6.5 Limitations

The solution proposed here is a partial solution 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 instructions that perform unit *push* and *pop* operations are handled. Instructions such as *pusha*, *popa* that increment and decrement the stack by more than one unit (usually 4 bytes) are not handled.

There is no model to retrieve contents of memory locations or contents of registers which limits the ability of the tool since stack operations done via memory are not being handled. While analyzing w32.evol for possible obfuscation of the *call* instruction, the prototype tool detects calls to each of the kernel functions as being obfuscated except the obfuscated call to *GetProcAddress*( ), which requires retrieving value (in this case the address of *GetProcAddress*) that is being pushed. Also DOCs cannot be used to analyze code rich in indirect control transfers such as jump through registers or memory. The idea of abstract locations from [8] can be used to overcome this limitation.

7 Conclusion and Future Work

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. Ten different obfuscations were shown to be detectable, and the methods for doing so outlined. These are obfuscations in common use by virus writers.

The abstract stack graph and the algorithm for constructing an abstract stack graph are partial solutions for detecting obfuscations in binaries. A more complete solution would consider additional instructions, provide a method of modeling actual memory locations, and track the contents of memory locations and content of registers.

Some work done elsewhere may one day provide a way of achieving this functionality. Balakrishnan and Reps have proposed a method for abstract interpretation of non-stack related instructions [8]. Their effort is aimed at discovering “something similar to C variables” by analyzing the memory accesses of a binary executable. Since their analysis assumes that a program conforms to a ‘standard compilation model,’ their

model of stack use is static. An activation record is associated with each procedure. The stack is a set of activation records that are linked together during interprocedural analysis. Adapting Balakrishan and Reps' algorithm to use an abstract stack graph may help create a complete system for detecting obfuscations. The adaptation may also help create a disassembler for obfuscated programs that cannot be fooled easily.

8 Appendix A: Obfuscation in Win32.Evol

The following piece of code is extracted from w32.evol virus.

```
00401208      CMP EAX, 77E00000
; checking kernel32 base address to identify Windows 2000 operating system
0040120D      JNZ SHORT 00401261
; did not find proper Win32 platform, so exiting
0040120F      MOV DWORD PTR DS:[ESI], 5500000F
; these are the four bytes beginning at 77E89B15 within kernel32.dll. This is corrupt and
; should actually be 550001F2. These bytes are being pushed on the stack.
00401215      MOV DWORD PTR DS:[ESI+4], 5151EC8B
; the next four bytes are also pushed on the stack. These 8 bytes are used for identifying
; the address of the Windows API module GetProcAddress( ) which is at 77E89B18
; within kernel32.dll
0040121C      ADD EAX, 80000      ; EAX = 77E00000 + 80000 = 77E80000
; since the required API modules begin from 77E80000
00401221      MOV EDI, EAX        ; use EDI to iterate between 77E80000
00401223      MOV ECX, 20000; and 77E80000 + 20000 = 77EA0000. ECX is counter
00401228      MOV EDX, DWORD PTR DS:[ESI]; compare the bytes 550001F2 and
0040122A      CMP EDX, DWORD PTR DS:[EDI]; contents of address in EDI
0040122C      JNZ SHORT 00401236          ; if condition fails, go to increment EDI
0040122E      MOV EDX, DWORD PTR DS:[ESI+4]
; else compare next four bytes 5151EC8B
00401231      CMP EDX, DWORD PTR DS:[EDI+4]
```

```
00401234        JE SHORT 00401243
00401236        ADD EDI, 1
; increment EDI and loop until the marker string bytes ‘eVOL’ are found on the stack
00401239        SUB ECX, 1
0040123C        CMP ECX, 0
0040123F        JNZ SHORT 00401228
00401241        JMP SHORT 00401261
00401243        ADD EDI, 3
; EDI = 77E89B15 + 3 = 77E89B18 which is the address of GetProcAddress() Win32
; API function. Further for ADD EDI, n EDI would contain an address 77E89B18 – n.
; The virus writer can use this and simply substitute the first 8 bytes occurring at
; (77E89B18 – n) address in place of the 8 bytes at instructions 0040120F and 00401215.
00401246        MOV ECX, EAX
; ECX = 77E80000 which is kernel32.dll base address. This requires to be passed as a
; parameter to GetProcAddress()
00401248        MOV EBX, DWORD PTR SS:[EBP]
0040124B        ADD EBX, 10
0040124E        MOV DWORD PTR DS:[EBX], 4C4F5665
; push string marker ‘eVOL’ on stack
00401254        MOV DWORD PTR DS:[EBX+4], ECX
; push kernel32.dll base address
00401257        MOV DWORD PTR DS:[EBX+10], EDI
; push address of GetProcAddress( )
```

```
0040125A      MOV EAX, 1
0040125F      JMP SHORT 00401263
00401261      XOR EAX, EAX        ; destroying the kernel32 base address
00401263      POP EDI             ; exiting
00401264      POP ESI
00401265      ADD ESP, 8
00401268      POP EBP
00401269      RETN  ; returns back to the main function from where this was called
```

The main function now calls a routine that pushes the parameters for *GetProcAddress*( ) on the stack and this routine in turn calls another routine that obfuscates the call to *GetProcAddress*( ).

```
0040153C      LEA EAX, DWORD PTR SS:[EBP-14]
; EBP holds contents of ESP
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          ; the string "GetTickCount"
00401557      PUSH EAX    ; pointer to the string is pushed on the stack. This is Arg2.
00401558      CALL 00401280; this routine retrieves kernel32.dll base address into eax
; which requires a search on the stack for the string marker 'eVOL'
0040155D      PUSH EAX            ; this is Arg1 pushed on the stack
0040155E      CALL 004012A7       ; this routine obfuscates call to GetProcAddress( )
00401563      MOV DWORD PTR SS:[EBP], EAX; Control returns back from 4012A7.
```

```
00401566    ADD ESP, 10          ; EAX now holds the address of GetTickCount().
00401569    POP EBP              ; these instructions from 00401563 to
0040156A    RETN                 ; 40156A transfer control to GetTickCount().
004012A7    PUSH EBP
004012A8    MOV EBP, ESP
004012AA    SUB ESP, 4    ; make space on stack
004012AD    MOV EAX, DWORD PTR SS:[EBP]
004012B0    MOV DWORD PTR SS:[EBP-4], EAX
004012B3    CALL 0040126A ; this routine retrieves the address of string marker
'eVOL' on the stack
004012B8    MOV EAX, DWORD PTR DS:[EBX+10]
; the contents of EBX+10 on the stack is the address of GetProcAddress( ) which is
;moved into EAX
004012BB    MOV DWORD PTR SS:[EBP], EAX
; the address of GetProcAddress( ) is pushed on the stack below the actual top of stack
004012BE    POP EBP; this pops top of stack and not the address of GetProcAddress( )
004012BF    RETN                 ; transfers control to GetProcAddress( )
```

Each of the required Win32 API function is called in the same way. Detection of this type of *call* obfuscation can be automated with the help of the abstract stack graph. The prototype tool can successfully detect calls to each of these kernel functions as being obfuscated, though, it fails to detect obfuscation of call to *GetProcAddress()*, which requires retrieving a value (in this case the address of *GetProcAddress*) that is being pushed. The idea of abstract locations from [8] can be used to achieve this.

9 Appendix B

Pseudo code to construct an abstract stack graph:

```
struct AbstractStackGraph
{
    abstract_node  N;
    List  predecessor_abstract_nodes;
    List  successor_abstract_nodes;
}asg;
struct WorkListElement
{
    unsigned int ip;
// ip holds the address of the next instruction to be executed.
    unsigned int asp;
// asp is the abstract stack pointer that holds the address of an instruction,
// which is the top of an abstract stack.
int num_of_successors;     // This is the number of successors for the instruction in
// asp, which is also a node in the abstract stack graph.
}wle;
List  elements;// This holds a list of worklist elements that are objects of type wle.
wle.asp = E;    // E is the address of entry instruction
wle.ip = E + instLength(E);
// The function instLength(A) returns the length of an instruction at address A.
wle.num_of_successors = 0;
```

```
struct AbstractNode
{
    unsigned int inst_addr;
    List program_points;
} abstract_node;
```

```
W = { <wle > }          // Work List              V = { }                    // Visited List
while ( W != NULL )
{
        retrieve w from W, where w ∈ W;
        if ( w !∈ V )
        {
        add w to V;
        // abstract_interpret( ) interprets the instruction specified by wle.ip,
        // modifies the abstract stack graph accordingly and either returns null, or
        // a new work list element or a list of work list elements. A list is returned
        // whenever an instruction is interpreted as a branch instruction or jump to
        // a case table.
        elements = abstract_interpret(wle);
        add elements to W;
        }
}
abstract_interpret(wle)
{
        instruction = getInstruction(wle.ip);
        prog_point = getProgramPoint(wle.asp);
        successor_prog_points = getSuccessorProgPoints(wle.asp);
        switch(instruction)
        {
```

```
case “push”:  // New abstract node is created and is made to point toward
              // all those nodes (in the so far formed abstract stack graph)
              // whose list of program points constitutes prog_point. Then
              // do the following:
      If no new change to the abstract stack graph, return(null);
      else    {
      wle.asp = wle.ip;
      wle.num_of_successors = sizeof(successor_abstract_nodes);
      // successor_abstract_nodes is associated with the new wle.asp
      wle.ip += instLength(wle.ip);
      return(<wle>);
      }  // end of else statement
case “pop”:   // The program point associated with wle.ip is added to the
              // list of program points of each successor abstract node
              // associated with wle.asp i.e. successor_prog_points. This
              // could mean to add the program point of wle.ip to more
              // than one abstract node, as wle.asp might have more than
              // one successor nodes. Then do the following:
If no new change to the abstract stack graph, return(null);
else    {
      temp_ip = wle.ip;
      temp_asp = wle.asp;
      wle.ip += instLength(wle.ip);
```

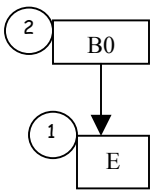
```
                for each ( successor abstract node, s_a_n, of temp_asp to which the
                program point of temp_ip was added )          {
                wle.asp =  s_a_n;
                wle.num_of_successors = sizeof(successor_abstract_nodes);
                // successor_abstract_nodes is associated with the new wle.asp
                wle_list = add(<wle>);
                }  // end of for loop
                return(<wle_list>);
                }  // end of else statement
        case "beqz":    // A branch statement has two possible destinations. One is
                        // the address to where it branches to and the other is to the
                        // fall through address. This basically just changes wle.ip.
                        wle.ip = getOperand(instruction);
                        wle_list = add(<wle>);
                        wle.ip += instLength(wle.ip);
                        wle_list = add(<wle>);
                        return(<wle_list>);
         case "jmp":  // A jmp statement jumps to a single destination address.
                        wle.ip = getOperand(instruction);
                        return(<wle>);
          // Similarly other cases can be defined …
        }  // end of switch statement.
}  // end of abstract_interpret( )
```

Example showing step-by-step process of constructing an abstract stack graph:

```
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
```

Sample program.

1. W = { <B0, E, 0> } V = { }



Retrieving <B0, E, 0> from W => W = { } and V = { <B0, E, 0> }

abstract_interpret(<B0, E, 0>) => *push eax*

return(<C1, B0, 1>)

W = { <C1, B0, 1> }

---

2. Retrieving <C1, B0, 1> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1> }

abstract_interpret(<C1, B0, 1>) => *sub ecx, 1h*

return(<C2, B0, 1>)

W = { <C2, B0, 1> }

---

3. Retrieving <C2, B0, 1> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1> }

abstract_interpret(<C2, B0, 1>) => *beqz B2*

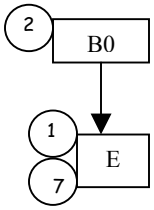
return(<B2, B0, 1>, <B1, B0, 1>)

W = { <B2, B0, 1>, <B1, B0, 1> }

---

4. Retrieving <B2, B0, 1> from W => W = { <B1, B0, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1> }



abstract_interpret(<B2, B0, 1>) => *pop eip*

return(<B4, E, 0>)

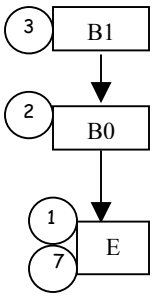
W = { <B1, B0, 1>, <B4, E, 0> }

---

5. Retrieving <B1, B0, 1> from W => W = { <B4, E, 0> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>,

<B1, B0, 1> }



abstract_interpret(<B1, B0, 1>) => *push ebx*

return(<B3, B1, 1>)

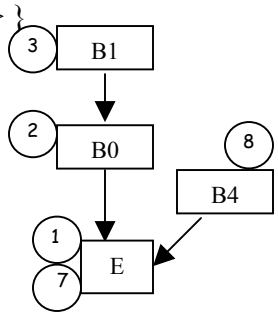
W = { <B4, E, 0>, <B3, B1, 1> }

---

6. Retrieving <B4, E, 0> from W => W = { <B3, B1, 1> }

and V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0> }



abstract_interpret(<B4, E, 0>) => *push esi*

return(<B5, B4, 1>)

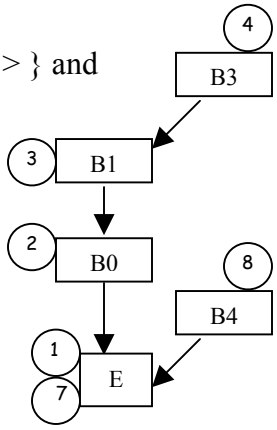
W = { <B3, B1, 1>, <B5, B4, 1> }

---

7. Retrieving <B3, B1, 1> from W => W = { <B5, B4, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0>, <B3, B1, 1> }



abstract_interpret(<B3, B1, 1>) => *push ecx*

return(<C4, B3, 1>)

W = { <B5, B4, 1>, <C4, B3, 1> }

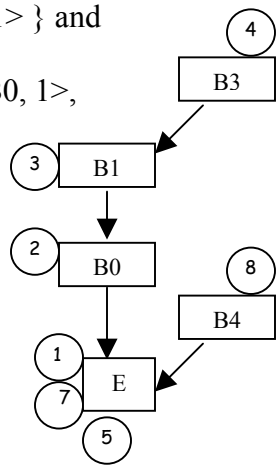
---

8. Retrieving <B5, B4, 1> from W => W = { <C4, B3, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>,

<B1, B0, 1>, <B4, E, 0>, <B3, B1, 1>,

<B5, B4, 1> }



abstract_interpret(<B5, B4, 1>) => *pop eip*

return(<C7, E, 0>)

W = { <C4, B3, 1>, <C7, E, 0> }

---

9. Retrieving <C4, B3, 1> from W => W = { <C7, E, 0> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>,

<B1, B0, 1>, <B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1> }

abstract_interpret(<C4, B3, 1>) => *dec ecx*

return(<C5, B3, 1>)

W = { <C7, E, 0>, <C5, B3, 1> }

---

10. Retrieving <C7, E, 0> from W => W = { <C5, B3, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>,

<B1, B0, 1>, <B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0> }

abstract_interpret(<C7, E, 0>) => *beq B0*

return(<B0, E, 0>, <B6, E, 0>)

W = { <C5, B3, 1>, <B0, E, 0>, <B6, E, 0> }

---

11. Retrieving <C5, B3, 1> from W => W = { <B0, E, 0>, <B6, E, 0> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>,

<B1, B0, 1>, <B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0> }

abstract_interpret(<C5, B3, 1>) => *beqz B1*

return(<B1, B3, 1>, <C6, B3, 1>)

W = { <B0, E, 0>, <B6, E, 0>, <B1, B3, 1>, <C6, B3, 1> }

---

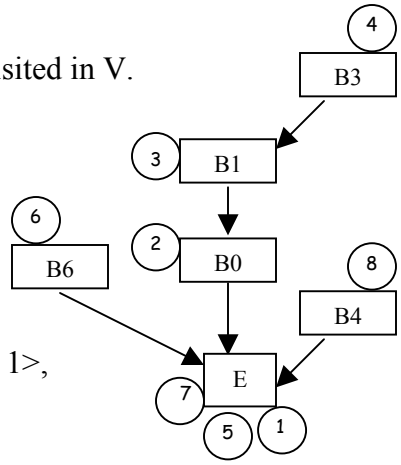
12. Retrieving <B0, E, 0> from W, but this is visited in V.

Retrieving <B6, E, 0> from W =>

W = { <B1, B3, 1>, <C6, B3, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>,

<C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,



<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0> }

abstract_interpret(<B6, E, 0>) => *call abc*

return(null)

W = { <B1, B3, 1>, <C6, B3, 1> }

---

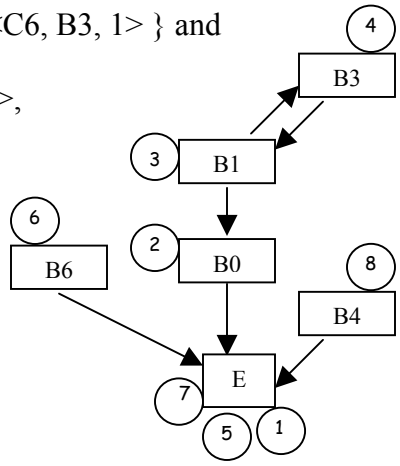
13. Retrieving <B1, B3, 1> from W => W = { <C6, B3, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0>,

<B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0>, <B1, B3, 1> }



abstract_interpret(<B1, B3, 1>) => *push ebx*

return(<B3, B1, 2>)

W = { <C6, B3, 1>, <B3, B1, 2> }

---

14. Retrieving <C6, B3, 1> from W => W = { <B3, B1, 2> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3 1> }

abstract_interpret(<C6, B3, 1>) => *jmp B5*

return(<B5, B3, 1>)

W = { <B3, B1, 2>, <B5, B3, 1> }

---

15. Retrieving <B3, B1, 2> from W => W = { <B5, B3, 1> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2> }

abstract_interpret(<B3, B1, 2>) => *push ecx*

/*Adds nothing new to the abstract stack graph*/

return(null)

---

16. Retrieving <B5, B3, 1> from W => W = { } and

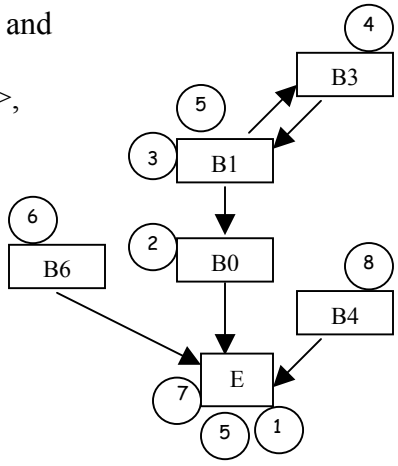
V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0>,

<B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0>, <B1, B3, 1>,

<C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1> }



abstract_interpret(<B5, B3, 1>) => *pop eip*

return(<C7, B1, 2>)

W = { <C7, B1, 2> }

---

17. Retrieving <C7, B1, 2> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2> }

abstract_interpret(<C7, B1, 2>) => *beq B0*

return(<B0, B1, 2>, <B6, B1, 2>)

W = { <B0, B1, 2>, <B6, B1, 2> }

---

18. Retrieving <B0, B1, 2> from W => W = { <B6, B1, 2> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

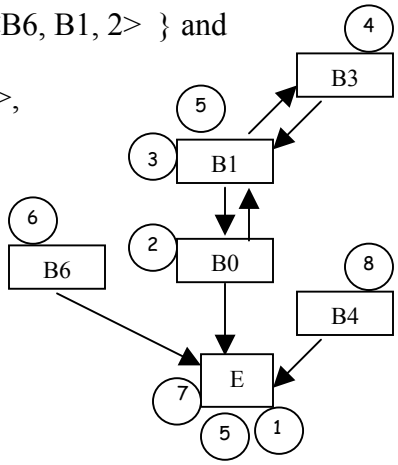
<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0>,

<B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0>, <B1, B3, 1>,

<C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2> }



abstract_interpret(<B0, B1, 2>) => *push eax*

return(<C1, B0, 2>)

W = { <B6, B1, 2>, <C1, B0, 2> }

---

19. Retrieving <B6, B1, 2> from W => W = { <C1, B0, 2> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>,

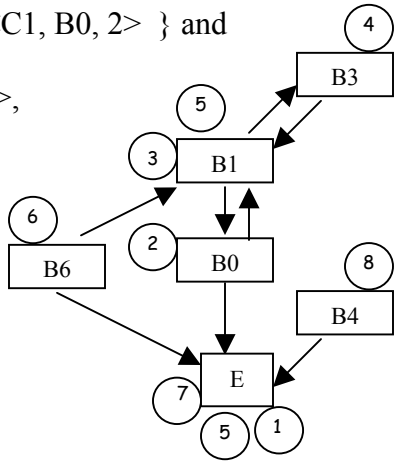
<B2, B0, 1>, <B1, B0, 1>, <B4, E, 0>,

<B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0>, <B1, B3, 1>,

<C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2> }



abstract_interpret(<B6, B1, 2>) => *call abc*

return(null)

W = { <C1, B0, 2> }

---

20. Retrieving <C1, B0, 2> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>,

<B5, B3, 1>, <C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>, <C1, B0, 2> }

abstract_interpret(<C1, B0, 2>) => *sub ecx, 1h*

return(null)

W = { <C2, B0, 2> }

---

21. Retrieving <C2, B0, 2> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>, <C1, B0, 2>, <C2, B0, 2> }

abstract_interpret(<C2, B0, 2>) => *beqz B2*

return(<B2, B0, 2>, <B1, B0, 2>)

W = { <B2, B0, 2>, <B1, B0, 2> }

---

22. Retrieving <B2, B0, 2> from W => W = { <B1, B0, 2> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

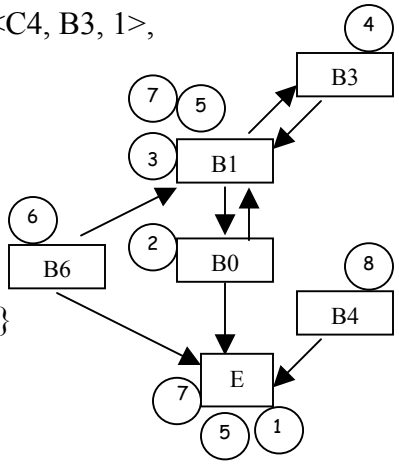
<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>,

<C7, E, 0>, <B6, E, 0>, <B1, B3, 1>,

<C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>,

<C1, B0, 2>, <C2, B0, 2>, <B2, B0, 2> }



abstract_interpret(<B2, B0, 2>) => *pop eip*

return(<B4, B1, 2>)

W = { <B1, B0, 2>, <B4, B1, 2> }

---

23. Retrieving <B1, B0, 2> from W => W = { <B4, B1, 2> } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>, <C1, B0, 2>, <C2, B0, 2>,

<B2, B0, 2>, <B1, B0, 2> }

abstract_interpret(<B1, B0, 2>) => *push ebx*

/*Adds nothing new to the abstract stack graph*/

return(null)

---

24. Retrieving <B4, B1, 2> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>, <C1, B0, 2>,

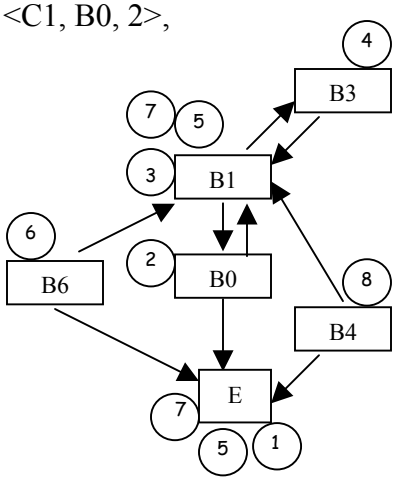
<C2, B0, 2>, <B2, B0, 2>, <B1, B0, 2>,

<B4, B1, 2> }

abstract_interpret(<B4, B1, 2>) => *push esi*

return(<B5, B4, 2>)

W = { <B5, B4, 2> }



---

25. Retrieving <B5, B4, 2> from W => W = { } and

V = { < B0, E, 0>, <C1, B0, 1>, <C2, B0, 1>, <B2, B0, 1>, <B1, B0, 1>,

<B4, E, 0>, <B3, B1, 1>, <B5, B4, 1>, <C4, B3, 1>, <C7, E, 0>,

<B6, E, 0>, <B1, B3, 1>, <C6, B3, 1>, <B3, B1, 2>, <B5, B3, 1>,

<C7, B1, 2>, <B0, B1, 2>, <B6, B1, 2>, <C1, B0, 2>, <C2, B0, 2>,

<B2, B0, 2>, <B1, B0, 2>, <B4, B1, 2>, <B5, B4, 2> }

abstract_interpret(<B5, B4, 2>) => *pop eip*

/*Adds nothing new to the abstract stack graph*/

return(null)

At step 25, the work list *W* is empty. The complete abstract stack graph is obtained at step 24.

The reason for tracking the *num_of_successors* information for the elements of *W* is that whenever a branch instruction is encountered, there are two possible paths that can be taken. Information has to be passed along both of these possible paths. In our case, the

elements of the work list W are being passed. These elements denote the state of the abstract stack graph.

From the above example run we see at step 3 there is a branch instruction. At step 4, we have a partial abstract stack graph constructed with *wle.ip* = B2 and *wle.asp* = B0 with B0 having a single successor in the abstract stack graph. Now whenever instruction at B2 is visited again due to a loop, at step 22, we again have the same *wle.ip* = B2 and *wle.asp* = B0 but now with B0 having two successors in the abstract stack graph. As can be noticed, the state of the graph has been updated and hence it is required that this updated graph be passed. If we hadn't introduced the *num_of_successors* information for an element, we would have concluded <B2, B0> as already visited and the abstract stack graph would have been incomplete.

The reason for this is the occurrence of *pop* in between a loop. Now, at step 4 whenever B0 is pointing toward E, which means B0 is the top of the abstract stack, and due to the *pop* instruction at B2, the top of stack would now be E. Hence, at step 22 whenever the instruction at B2 is reached again, the state of the abstract stack graph has changed wherein B0 now points toward E and B1. Due to the *pop* at B0, the top of the abstract stack can now be either E or B1. If this information were not considered (i.e. *num_of_successors)* then the abstract stack graph would be incomplete.

Bibliography

[1] "2004-03-30, News Alert, Netsky Climbs to 2nd Worst Malware since 1995," http://www.mi2g.com/, Last accessed November 29, 2004.

[2] "Teso, Burneye Elf Encryption Program," https://teso.scene.at, Last accessed November 29, 2004.

[3] "W32.Cabanas," http://securityresponse.symantec.com/avcenter/venc/data/w32.cabanas.html, Last accessed November 29, 2004.

[4] "W32/Chiton," http://www.virusbtn.com/resources/viruses/indepth/gemini.xml, Last accessed November 29, 2004.

[5] "W32/Gemini," http://www.virusbtn.com/resources/viruses/indepth/gemini.xml, Last accessed November 29, 2004.

[6] "W95.Bistro," http://securityresponse.symantec.com/avcenter/venc/data/w95.bistro.html, Last accessed November 29, 2004.

[7] "Z0mbie," http://z0mbie.host.sk, Last accessed November 29, 2004.

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

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

[10] S. Cho, "Win32 Disassembler," http://www.geocities.com/~sangcho/disasm.html, Last accessed November 29, 2004.

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

[12] C. Cifuentes and M. V. Emmerik, "Uqbt: Adaptable Binary Translation at Low Cost," in *IEEE Computer org*, 2000.

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

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

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

[16] R. N. Horspool and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *The Computer Journal*, vol. 23, pp. 223-229, 1979.

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

[18] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *USENIX Secuirty 2004*, San Diego, 2004.

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

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

[21] N. Mehta and S. Clowes, "A Security Microcosm Attacking and Defending Shiva," http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-asia-03/bh-asia-03-clowes.pdf, Last accessed June 30 2004.

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

[23] A. S. Murawski, "About the Undecidability of Program Equivalence in Finitary Languages with State," in *ACM Transactions on Computational Logic*, 2004.

[24] R. Muth, S. Debray and S. Watterson, "Alto: A Link-Time Optimizer for Compaq Alpha," *Software - Practice and Experience*, vol. 31, pp. 67-101, 2001.

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

[26] R. L. Sites, A. Chernoff**,** M. B. Kerk, M. P. Marks and S. G. Robinson, "Binary Translation," *Communications of the ACM*, vol. 36, pp. 69-81, 1993.

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

[28] P. Szor, "Coping with Cabanas," *Virus Bulletin*, pp. 10-12, 1997.

[29] P. Szor, "Hps," *Virus Bulletin*, 1998, http://www.peterszor.com/hps.pdf.

[30] P. Szor, "Attacks on Win32 - Part Ii," in *Virus Bulletin Conference*, Orlando, 2000.

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

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

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

[34] L. Zeltser, "Reverse Engineering Malware," http://www.zeltser.com/sans/gcih-practical/revmalw.html, Last accessed June 30 2004.

Abstract

A common approach to detecting malicious code is to examine the calls a binary makes to the operating system. Knowing this, malicious code programmers hide the calls using a variety of obfuscations. 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 work presents a method to statically detect calls in binary code. The main idea is to use abstract interpretation to detect where the normal *call-ret* calling 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. A linear algorithm is defined for calculating the abstract stack graph. Methods for using the abstract stack graph are shown to detect ten different obfuscations. The technique is demonstrated by implementing a prototype tool called DOCs using several test programs and a metamorphic virus called w32.evol.

Biographical Sketch

Mr. Eric Uday Kumar was born in Hyderabad, India on September 24, 1979. He graduated with a bachelor's degree with distinction in Computer Science in 2002 from the B. V. Raju Institute of Technology, (affiliated to Jawaharlal Nehru Technological University), Andhra Pradesh, India. He entered the master's program in Computer Science at the University of Louisiana at Lafayette in Fall 2002. Following completion of this degree, he will pursue a Ph.D. in the area of computer security.