Adapting Call-string Approach for x86 Obfuscated Binaries

Davidson R. Boccardo[1], **Arun Lakhotia**[2], **Aleardo Manacero Jr**[3], **Michael Venable**[2]

[1]Electrical Engineering Dept., Sao Paulo State University (UNESP)
Ilha Solteira - São Paulo - Brazil

[2]Center for Advanced Computer Studies, University of Louisiana at Lafayette
Lafayette - Louisiana - USA

[3]Computer Science and Statistics Dept., Sao Paulo State University (UNESP)
São José do Rio Preto - São Paulo - Brazil

{drb3065,arun,mpv}@louisiana.edu, aleardo@ibilce.unesp.br

Abstract. *Call-string technique, a classic technique for interprocedural analysis, cannot be applied to binaries that do not follow stack conventions used by high-level language compilers. Examples are programs that make obfuscated procedure calls using push and return instructions, which is a technique largely used to hide malicious code. In this paper it is shown that a technique equivalent to call-string, the abstract stack graph (ASG), may be used to identify such obfuscations. An ASG contains nodes representing statements that push some element on the stack. An edge in the graph represents the next instruction that pushes a value on the abstract stack along some control flow path. For a program that manipulates stack using only call and return instructions, its ASG is equivalent to its call-graph. Since the ASG represents stack operations by any instruction it becomes a suitable substitute for the call-graph for interprocedural analysis of obfuscated binaries.*

Resumo. *A técnica 'call-string', técnica clássica para análise interprocedural, não pode ser aplicada a binários que não seguem padrões de uso da pilha utilizados por compiladores de linguagens de alto nível. Exemplos são programas que ofuscam chamadas de procedimento usando uma combinação de instruções 'push' e 'ret', que é uma técnica extremamente utilizada para esconder código malicioso. Neste artigo, uma técnica equivalente à 'call-string' é demonstrada, em que um grafo abstrato da pilha pode ser utilizado para identificar estas ofuscações. Um grafo abstrato da pilha contem nós representando instruções que realizam inserção na pilha. Uma aresta neste grafo representa a próxima instrução que realiza a inserção na pilha abstrata ao longo de um caminho do fluxo de controle. Para um programa que manipula a pilha utilizando somente instruções 'call' e 'ret', seu grafo abstrato da pilha é equivalente ao seu grafo de chamadas. Desde que o grafo abstrato da pilha representa operações na pilha por qualquer instrução, o mesmo torna-se um substituto apropriado para o grafo de chamadas para análise interprocedural de binários ofuscados.*

1. Introduction

Recently, research activity has increased in the area of binary analysis [Larus and Schnarr 1995, Cifuentes and Fraboulet 1997, Cifuentes et al. 1998, Amme et al. 2000, Goodwin 1997, Schwarz et al. 2001, Debray et al. 1998, Srivastava and Wall 1993, Venkitaraman and Gupta 2004, Bergeron et al. 2001, Balakrishnan 2007, Guo et al. 2005, Reps et al. 2006, Reps and Balakrishnan 2008, Christodorescu and Jha 2003, Lakhotia et al. 2005, Venable et al. 2005, Kinder et al. 2009]. For Commercial Off-The Shelf (COTS) programs or other third-party programs in which the source code is not available to the analyst, analysis for malicious (hidden) behavior can be performed reliably only on binaries. Even when the source code is available, analyzing the binary is the only true way to detect hidden capabilities, as demonstrated by Thompson in his Turing Award Lecture [Thompson 1984].

Current methods for analyzing binaries are modeled on methods for analysis of source code, where a program is decomposed into a collection of procedures, and the analyses are classified into two types: intraprocedural and interprocedural. In intraprocedural analysis, the entire program is treated as one function, leading to very significant over-approximation. In interprocedural analysis, procedures are taken into account and complications can arise when ensuring that calls and returns match one another, where information may flow along a call node to a procedure and then be propagated by a return node to another call node calling the same procedure.

Classical interprocedural analysis may be performed by procedure-inlining followed by an intraprocedural analysis, or using the functional approach through procedure summaries, or by providing the calling-context using the call string approach [Sharir and Pnueli 1981].

Since a binary, albeit disassembled, is not syntactically rich, the identification of procedure boundaries, parameters, procedure calls, and returns is done by making assumptions. Such assumptions consist of the sequence of instructions used at a procedure entry (prologue), at a procedure exit (epilogue), the parameter passing convention, and the conventions to make a procedure call. When a binary violates the convention, the analysis fails.

This paper presents a method for performing interprocedural analysis when a binary does not follow the standard compilation model of manipulating the stack. For example, a binary may not use the *call* instruction, instead it may simulate a *call* by a combination of two *push* and one *ret* instruction. Such non-standard methods of making a call are explicitly used by malicious programs to defeat automated analysis [Boccardo et al. 2007, Christodorescu and Jha 2003, Lakhotia and Singh 2003, Szor and Ferrie 2001]. Such obfuscations may also be used for the purpose of hiding intellectual property [Linn and Debray 2003, Collberg et al. 1997, Wroblewski 2002]. The method presented here is applicable even when a binary is not deliberately obfuscated. This is because the standard compilation models are really not industry standard. The standards are compiler specific, and may even vary with optimization levels of the compiler.

More specifically this paper demonstrates that the Abstract Stack Graph (ASG), introduced earlier by [Lakhotia et al. 2005], can be used to adapt Sharir and Pnueli's [Sharir and Pnueli 1981] call-string approach to perform context-sensitive interprocedural analysis of programs with non-standard manipulation of stack, including obfuscation of calls. It is obvious from the construction of the ASG ([Lakhotia et al. 2005]) that the call-graph (CG) and the ASG are isomorphic for the same program when the program uses standard compilation model. We use this to show that even when procedure calls are obfuscated, the necessary structure of the CG is preserved in the ASG. Thus, a call-string of Sharir and Pnueli, which is a finite length path in a call-graph, maps to what we term as a stack-string, a finite length path in an ASG.

The benefit of using ASG over CG is immediate. Interprocedural analysis methods developed using call-string approach, which are restricted to a standard compilation model, may be made more general simply by switching to using stack-strings. For instance, Balakrishnan and Reps's WYSINWYX system develops higher level abstractions of binaries, such as determining the memory layout of variables [Balakrishnan 2007, Balakrishnan and Reps 2004]. This system utilizes Sharir and Pnueli's call-string approach for context-sensitive interprocedural analysis. The applicability of this system can be expanded to a larger class of programs by using stack-string, instead of call-string.

The costs and issues of constructing ASG are similar to those of constructing CG. [Lakhotia et al. 2005] have presented an algorithm that constructs a precise ASG for programs that manipulate the stack pointer by adding/subtracting a constant and in which the address of control transfer can be computed to be a constant. This class of programs is similar to the class of programs containing only direct calls (no indirect calls). [Venable et al. 2005] has extended Lakhotia *et al.*'s algorithm to remove this restriction on the class of language. Their algorithm does not decompose a program into procedures, as this decomposition is not assumed to be known. This yields a program that is resource intensive and context-insensitive. Our goal is the improvement of Venable *et al.*'s algorithm by using an ASG constructed by the algorithm to guide the construction of the ASG. This issue is analogous to constructing CGs for programs with indirect calls [Lakhotia 1993], in which a CG is used to guide the construction of CG. Research results from solving the issue for constructing CGs [Zhang and Ryder 2007, Milanova et al. 2004] may thus be borrowed for constructing ASGs for analogous constraints.

The remaining sections of this paper are arranged as follows. Section 2 presents related work in the area of interprocedural analysis and binary analysis. Section 3 presents an overview of the call-string approach and also highlights its drawbacks. Section 4 describes how to adapt call-string using ASG to overcome the drawbacks come from call-string approach, when interprocedural analysis is made on non conventional binaries. Section 5 contains our concluding remarks.

2. Related Works

Precise and efficient context-sensitive interprocedural data-flow analysis of high-level languages has been an active area of research. Most of these efforts, represented by [Sagiv et al. 1996, Cousot and Cousot 2002, Muller-Olm and Seidl 2004], are focused on special classes of problems for high-level languages. The general strategy falls within the two approaches proposed by Sharir and Pneuli [Sharir and Pnueli 1981], namely the call-string approach or the procedure summaries approach.

Interprocedural analysis of binaries has also received attention for post-compile time optimization [Srivastava and Wall 1993] and for analyzing binaries with the intent to detect vulnerabilities not visible in the source code, such as those due to memory mapping of variables [Balakrishnan 2007]. Goodwin uses procedure summary approach to interprocedural analysis to aid link-time optimization [Goodwin 1997]. In contrast, Balakrishnan [Balakrishnan 2007] uses the call-string approach. As mentioned earlier, these methods assume a certain compiler model to identify code segments related to performing procedure calls.

On a tangential direction there has been significant work in obfuscation of programs with the intent to thwart static analysis [Linn and Debray 2003, Collberg and Thomborson 2002]. Such obfuscations may be used by benign as well as malicious programs for the same purpose, to make it difficult for an analyst to detect its function or its underlying algorithm. The obfuscation techniques attack various phases in the analysis of binary [Lakhotia and Singh 2003].

A metamorphic virus, a virus that transforms its own code as it propagates, may use procedure call obfuscations to enable its transformation operation. The Win32.Evol virus, for example, uses call-obfuscation just for this purpose. A side-effect of this is that the virus defeats any interprocedural analysis that depends on a traditional compiler model [Lakhotia and Singh 2003]. Increase in obfuscation efforts have also triggered attempts to analyze obfuscated code. There have been efforts to use semantics based methods for detecting malware [Dalla Preda et al. 2007, Christodorescu and Jha 2003, Bergeron et al. 2001]. Term-rewriting has been proposed to normalize variants of a metamorphic malware [Walenstein et al. 2006]. None of these works address analysis of obfuscated programs that do not conform to the standard compilation model.

3. Interprocedural Analysis

Analyzing a procedure is classically represented as a control flow graph containing nodes and edges. Nodes represent computational elements while edges represent transfer of control. Program analysis algorithms propagate information along edges of this graph. For interprocedural analysis, each procedure call is treated as a node with edges to all the procedures that can be called from the call site. Similarly, a return statement is represented as a node with edges to all the nodes where control may be transferred after a procedure terminates. We use the phrase ‘call edges’ and ‘return edges’ for edges that represent transfer of control due to call and return statements, respectively.

Propagating information to all the successors of a node in the graph leads to context-insensitive analysis. Information may flow along a call edge to a procedure and then be propagated by a return edge to another call site calling that same procedure. Thus, incorrect combinations of call and return edges creates spurious pathways in the information flow.

3.1. Call-string approach

Sharir and Pnueli’s call-string approach for context-sensitive interprocedural analysis involves tagging information with an encoded history of calls along which it is propagated. When information flows along a call-edge, the corresponding call site is added to the history. The history is propagated as the tagged information is used to compute other information. Finally, at the return edge, information is propagated back to only the call sites in the history, and in turn the last call site is removed from the history.

The context-sensitive flow of information by maintaining call strings comes at a price. There may be an exponential, if not infinite, number of interprocedurally valid paths, paths in which the call and return edges are correctly paired. Thus, the amount of information to be maintained explodes.

The information space is made manageable by capping the history being maintained to up to some *k* most recent call sites. This ensures context-sensitive flow of information between the most recent *k* sites, but context-insensitive flow between call and return sites that are more than *k* call sites apart.

A call-graph is a labeled graph in which each node represents a procedure, each edge represents a call, and the label on the edge represents a call site. A call string is a sequence of call-sites $(L_1 L_2 ... L_n)$ such that call site L_1 belongs to the entry procedure, and there exists a path in the call-graph consisting of edges $L_1, L_2, ..., L_n$. A call string can be saturated when the encoded history of the procedure calls exceeds the limit *k* imposed during analysis. Its representation is given as $(*L_1 L_2 ... L_k)$, where the parameter *k* is the bound of the call string size and represents the set $\{cs_k \mid cs_k \in CS_k$, $cs = \pi L_1 L_2 ... L_k$ and $|\pi| \geq 1\}$.

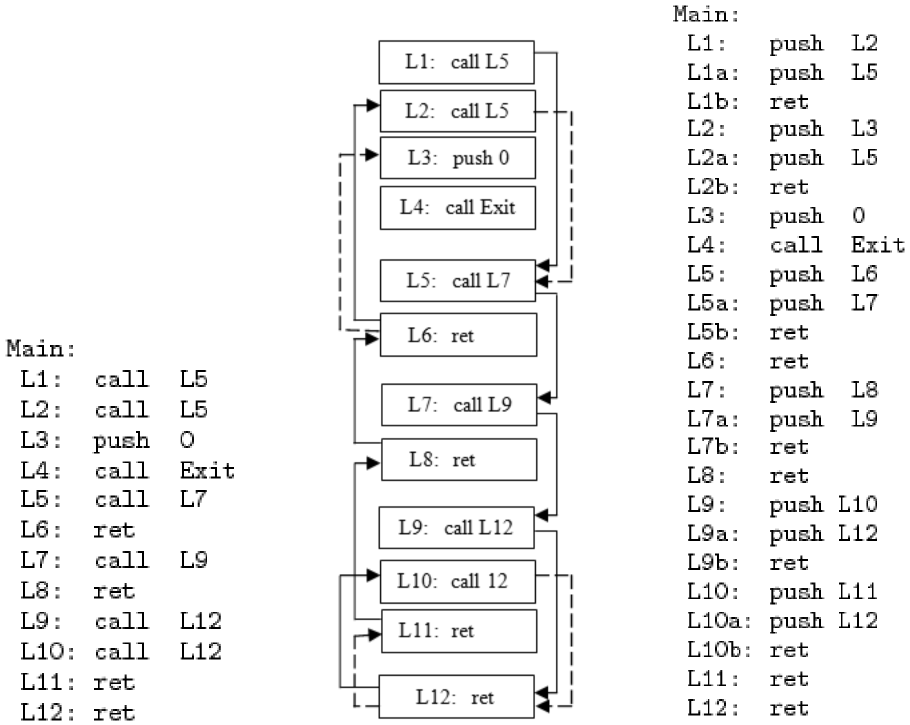
The call-string approach can be used to perform context-sensitive interprocedural analysis for binaries, so long as the Interprocedural Control Flow Graph (ICFG) can reliably be constructed. When this graph cannot be constructed, such as for obfuscated or optimized code, the approach breaks down.

Figure 1(a) contains a sample code that presents the motivation. It is a simplified program showing only the call and return structure. Figure 1(b) shows the ICFG of this program. The edges of the graph represent call and return edges. Context-sensitive interprocedural analysis algorithms require pairing the edges such that information flowing from one call node is not propagated to another call node [Sharir and Pnueli 1981]. Figure 1(c) shows an obfuscated version of the sample program. It is generated by replacing every *call* instruction by a sequence of two *push* instructions and a *ret* instruction, where the first *push* pushes the address of the instruction after the *call* instruction (the return address of the procedure call), the second *push* pushes the target address of the call, and the *ret* instruction causes the execution to jump to the target address of the call. Since the program has no *call* instruction, partitioning it using classical algorithms will yield only one procedure (consisting of the entire code). Furthermore, the *ret* instructions will be treated as if they were returning to the caller, thus generating an incorrect ICFG. As a consequence, any resulting analysis based on this ICFG will also be incorrect.

The obfuscation shown in Figure 1(c) is naïve and presented to demonstrate the concept. More obfuscations, although still trivial, may be performed by shuffling the two *push* instructions among other code. More complex obfuscations may be achieved by not using *push* and *ret* instructions; but instead using move, increment, and decrement operations directly on the stack pointer to perform equivalent functions.

[Lakhotia et al. 2005] identify the following four types of obfuscation related to call and return statements.

1. *A call simulated by other means.* The semantics of a '*call addr*' instruction is as follows: the address of the instruction after the *call* instruction is pushed on the stack and the control is transferred to the address *addr*, the target of the call. Win32.Evol achieves the same semantics by a combination of two *push* instructions and a *ret* instruction. There are other ways to achieve the equivalent behavior.



```
Main:
 L1:   call  L5
 L2:   call  L5
 L3:   push  0
 L4:   call  Exit
 L5:   call  L7
 L6:   ret
 L7:   call  L9
 L8:   ret
 L9:   call  L12
 L10:  call  L12
 L11:  ret
 L12:  ret
```

```
Main:
 L1:    push  L2
 L1a:   push  L5
 L1b:   ret
 L2:    push  L3
 L2a:   push  L5
 L2b:   ret
 L3:    push  0
 L4:    call  Exit
 L5:    push  L6
 L5a:   push  L7
 L5b:   ret
 L6:    ret
 L7:    push  L8
 L7a:   push  L9
 L7b:   ret
 L8:    ret
 L9:    push L10
 L9a:   push L12
 L9b:   ret
 L10:   push L11
 L10a:  push L12
 L10b:  ret
 L11:   ret
 L12:   ret
```

(a) Sample code. (b) ICFG. (c) Obfuscated version.

Figure 1. Example motivating context-sensitive analysis of obfuscated code.

2. *A call instruction may not make a call.* The call instruction performs two actions - pushing a return address on the stack and transfer of control. A program may use the instruction primarily for control transfer, and discard the return address later, as done by Win32.Evol. The program may also use the instruction as a means to retrieve the address from memory of a certain point in code, as it is done by some worms.
3. *A return may be simulated by other means.* A *ret* instruction is complementary to a *call*. It pops the return address (typically pushed by a *call* instruction) from the stack and transfers control to that address. The same semantics may be achieved by using other statements. For instance, the return address may be popped into a register and a *jmp* instruction may be used to transfer control to the address in that register.
4. *A return instruction may not return back to a call site.* A program may utilize the *ret* instructions to transfer control to another instruction, completely unrelated to any *call* instruction. For instance, the *ret* instruction can be used to simulate a *call* instruction, as outlined earlier.

4. Using ASG in place of CG

We now show the relationship between ASG and call-graph (CG), and how an ASG may be used in place of CG for interprocedural context-sensitive analysis.

The concept of ASG from [Lakhotia et al. 2005] is developed by first introducing the notion of abstract stack. An abstract stack is an abstraction of the real (concrete) program's stack. While the concrete stack keeps actual data values that are pushed and popped in a LIFO (Last In First Out) sequence, the abstract stack stores the addresses of the instructions that *push* and *pop* values in a LIFO sequence.

An ASG is a concise representation of all, potentially infinite, abstract stacks at all points in the program. A path (sequence of nodes beginning from the abstract stack's top toward its bottom) in the graph represents a specific abstract stack. An ASG is represented by a labeled graph in which each node represents an instruction that manipulates the stack pointer to effectively push some data on the stack, and the edges represent potential traces that push values onto the stack.

Lemma 4.1 *Paths in ASG preserve call-strings of CG for programs that do not manipulate instructions in the stack, except when using the 'call' and 'ret' instructions.*

Proof The nodes of the ASG for such a program will consist of only the call sites. An edge in the ASG from a call-site L_j to a call-site L_i exists iff there is an execution path from L_i to L_j, with no other *call* instruction along the path. Assume that L_i is a statement in procedure P_i, and L_j is in procedure P_j. Assume also that L_j calls procedure P_k. Thus, in the CG exists an edge from P_i to P_j, with the annotation L_i, and an edge from P_j to P_k with the annotation L_j. This implies that an edge L_i to L_j in ASG corresponds to an edge P_i to P_j with annotation L_i, and vice-versa. A call-string will thus correspond to a path in the ASG. ▮

Therefore, a call-string of Sharir and Pnueli, which is a finite length path in a call-graph, can be mapped to what we term as a stack-string, a finite length path in an ASG. Formally, a stack-string can be defined as a path in the ASG of program locations ($L_1L_2...L_n$) such that program location L_1 is the first element pushed on the stack, and there exists a path in the ASG consisting of program locations $L_1L_2...L_n$ such that L_n is the top of the stack. Analogous to Sharir and Pnueli's saturated call-string we define a saturated stack-string as a string whose encoded history of the program locations exceeds some limit *k*. It is represented as $(*L_1L_2...L_k)$, where the parameter *k* is the bound of the stack-string size and represents the set $\{ss_k \mid ss_k \in SS_k, ss = \pi L_1L_2...L_k$ and $|\pi| \geq 1\}$.

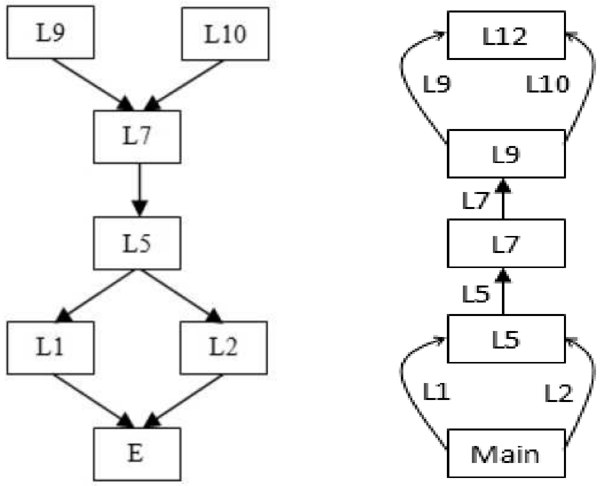
Figure 2 shows the ASG and CG for the code of Figure 1(a). The correspondence between ASG and CG is obvious. The nodes in the ASG represent the edges (call-sites) in the call graph. An edge in the ASG represents the next instruction that pushes a value on the abstract stack along some control flow path. The corresponding called functions are represented side by side of the call-site.

Now consider programs that use other instructions to manipulate stack, but do not attempt to obfuscate *call* and *ret*.

Corollary 4.2 *For any program that does not obfuscate 'call' and 'ret' instructions, an ASG path containing at least one 'call' instruction maps to a unique path in the CG. Also, a call-string in CG of this program corresponds to one or more ASG paths (that can be mapped to the CG).*

Proof Follows from the previous lemma. If on an ASG path, instructions other than the $call$ instructions are removed it will correspond to a call string. The second part follows by contradiction. ▮

The above discussion implies the ASG can be used as a substitute for programs that do not obfuscate $call$ and ret instructions. When performing interprocedural analysis,



(a) Abstract stack graph. (b) Call-graph.

Figure 2. Abstract stack graph and call-graph for code of Figure 1(a).

values may be tagged with k-length paths in the ASG, instead of the CG. Of course, the tags would have to take into account the non-call instructions to preserve equivalence in using call-strings over CG.

The real value, though, comes in the application of ASG for analysis of obfuscated programs. Since CGs cannot be constructed for obfuscated programs (without deeper analysis), it is rather difficult to theoretically offer an argument that ASGs are a suitable replacement for CGs of obfuscated programs. Hence, we will make the case of use of ASG by example.

Figure 3 shows the ASG for the obfuscated code of Figure 1(c). It is evident that all paths in the ASG of the non-obfuscated version (Figure 2(a)) can be mapped to paths in ASG of the obfuscated version. The obfuscated version has extra nodes (represented by the suffix a), representing *push* instructions used to *push* the address of the procedure being called onto stack.

The similarity of the graphs of Figures 3 and 2(a) suggests that paths in the ASG may be treated as a replacement for call-string, even for obfuscated programs. Instead of computing, propagating, and updating call-string over CG, an interprocedural analysis algorithm may construct, propagate, and update call-strings over ASG. When an ASG can be computed before the analysis, all possible calling contexts for a statement can be determined from the top of stacks reaching that point and the ASG [Lakhotia et al. 2005, Venable et al. 2005]. When the computation of ASG may require performing other analysis, as is likely in obfuscated programs, the two analysis may be performed in lock-step.

There is just one more optimization step that may be valuable when using an ASG as a replacement for CG. Even for non-obfuscated code an ASG may have more nodes than just call sites. Thus, a *k* length path in the ASG may have fewer call sites than its corresponding *k* length call-string. Since the computational resources needed may increase non-linearly with *k*, simply increasing *k* may not be an option. Instead one

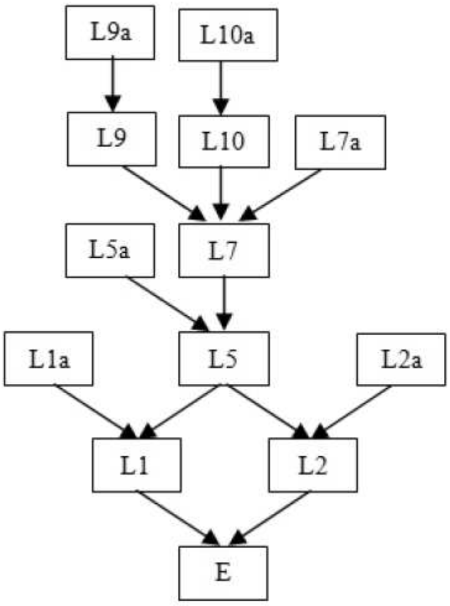


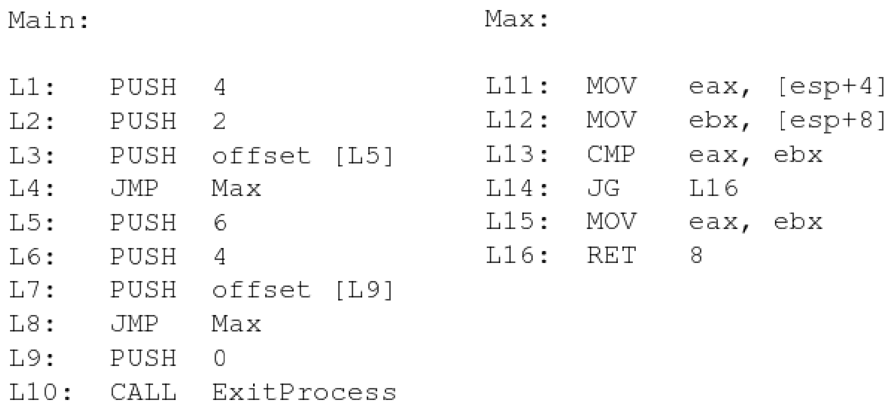
Figure 3. Abstract stack graph for the obfuscated code of Figure 1(c)

may reduce the number of nodes in the ASG by creating 'blocks' of nodes, as is done in control flow graphs (CFG). A block is a sequence of nodes in an ASG with a single entry and a single exit. Using ASG made up of blocks of instructions, instead of individual instructions, will enable propagation of the calling contexts for larger *k*.

A prototype has been constructed that uses the previous ideas to perform context-sensitive analysis on obfuscated programs. This prototype has been implemented over Venable's context-insensitive algorithm for detecting obfuscated calls [Venable et al. 2005]. The prototype is written in Java utilizing the Eclipse framework. Eclipse is an extensible development environment with a rich set of tools to aid in development. Programs developed on Eclipse are written as plugins to the Eclipse platform and can take advantage of the robust Eclipse framework to decrease development time.

In the following examples, we explain the context-sensitive analysis process of obfuscated code using stack-strings. We will only consider instructions that involve stack operations. Figure 4 contains a sample assembly obfuscated program with two contexts. The program consists of two functions: *Main* and *Max*. The function *Max* takes as input two numbers and returns as output the larger of the two numbers. The function *Main* pushes the two arguments onto the stack, but instead of calling *Max* directly, it pushes the return address onto the stack and jumps to *Max*. The function *Max* is called twice by the function *Main*. This obfuscation technique can effectively hide the boundary between the two procedures and results in a less accurate ICFG. Analysis methods relying on the flow graph may, in effect, produce less accurate results as well.

After careful inspection, one may observe that in order to perform context-sensitive analysis we have to match the return node at L_{16} to the nodes L_5 or L_9 (return sites). Using stack-string we can correctly perform these matches. The context-sensitive analysis process of obfuscated code using stack-strings follows.



```
Main:                          Max:

L1:   PUSH  4                  L11:  MOV   eax, [esp+4]
L2:   PUSH  2                  L12:  MOV   ebx, [esp+8]
L3:   PUSH  offset [L5]        L13:  CMP   eax, ebx
L4:   JMP   Max                L14:  JG    L16
L5:   PUSH  6                  L15:  MOV   eax, ebx
L6:   PUSH  4                  L16:  RET   8
L7:   PUSH  offset [L9]
L8:   JMP   Max
L9:   PUSH  0
L10:  CALL  ExitProcess
```

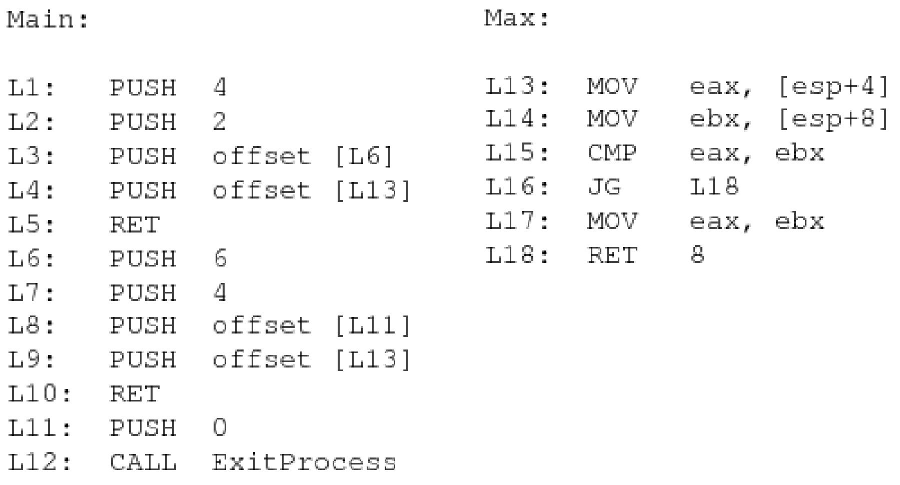
Figure 4. Obfuscated call using *push/jmp* instructions.

Upon entry, the stack-string is $\perp$, i.e., signaling that the context is currently empty. Instruction L_1 pushes a value onto the stack, consequently changing the stack-string to $\perp - L_1$. Instructions L_2 and L_3 perform in a manner similar to L_1. At the point L_3 the stack-string context is $\perp - L_1 - L_2 - L_3$. Instruction L_4 transfers the control to the destination of the jump and the stack-string context is left unchanged.

The next instruction evaluated is the target of the jump, or L_{11} in this case. Instructions L_{11} to L_{15} have no effect on the stack-string context. The *ret 8* instruction at L_{16} implicitly pops the return address off the top of the stack (L_5) based on the current stack-string context $\perp - L_1 - L_2 - L_3$, and continues execution at that address. It also changes the stack-string context $\perp - L_1 - L_2 - L_3$ to $\perp$. Evaluation continues at L_5, which pushes a value onto the stack, consequently changing the stack-string to $\perp - L_5$. Instructions L_6 and L_7 perform in a manner similar to L_5. At this point the current stack-string context is $\perp - L_5 - L_6 - L_7$. Similarly to the instruction of L_4, L_8 transfers the control to the function *Max* and the context is left unchanged. At this point, the analysis contains two stack-contexts: $\perp - L_5 - L_6 - L_7$ and $\perp - L_1 - L_2 - L_3$. This provides context-sensitivity in the analysis, in which pieces of code are analyzed separately for different data flow values at different stack-string contexts, consequently, leading to more precise results. At instruction L_{16}, the *ret 8* implicitly pops the return address off the top of the stack (L_9) on the stack-string context $\perp - L_5 - L_6 - L_7$ and continues execution at that address. It also changes the stack-string context $\perp - L_5 - L_6 - L_7$ to $\perp$. Evaluation continues at L_9, which proceeds to the end of the program.

Figure 5 shows the same code, but using the *push/ret* obfuscation. Instructions L_3 (L_8) and L_4 (L_9) push the return address and the target address onto the stack. L_5 (L_{10}) consists of a *ret* instruction that causes execution to jump to the function *Max*. Analysis methods that rely on the correctness of a ICFG will surely fail when analyzing such code.

During the interpretation, at instruction L_5, the stack-string context is $\perp - L_1 - L_2 - L_3 - L_4$. The *ret* instruction implicitly pops the return address off the top of the stack (L_{13}) on the current stack-string context $\perp - L_1 - L_2 - L_3 - L_4$, alters the stack-string context to $\perp - L_1 - L_2 - L_3$ and continues execution at that address. As in the previous example, we have two contexts for this program. At instruction L_{13}, the analysis contains two stack-contexts: $\perp - L_1 - L_2 - L_3$ coming from the return instruction at L_5



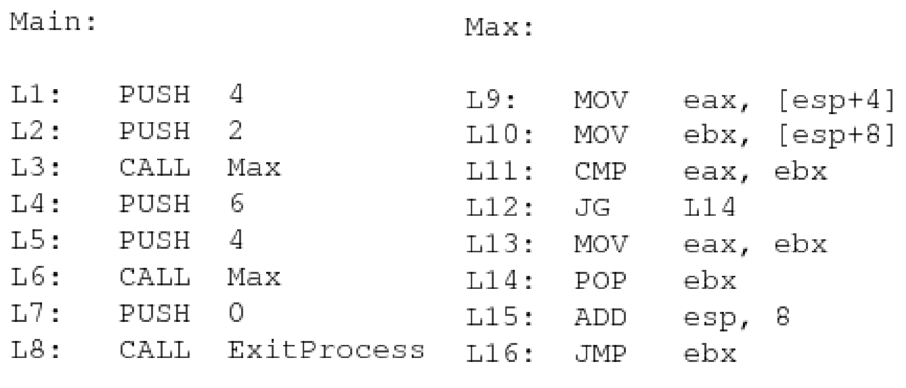
```
Main:                              Max:

L1:   PUSH  4                      L13:  MOV   eax, [esp+4]
L2:   PUSH  2                      L14:  MOV   ebx, [esp+8]
L3:   PUSH  offset [L6]            L15:  CMP   eax, ebx
L4:   PUSH  offset [L13]           L16:  JG    L18
L5:   RET                          L17:  MOV   eax, ebx
L6:   PUSH  6                      L18:  RET   8
L7:   PUSH  4
L8:   PUSH  offset [L11]
L9:   PUSH  offset [L13]
L10:  RET
L11:  PUSH  0
L12:  CALL  ExitProcess
```

Figure 5. Obfuscated call using *push/ret* instructions.

and $\perp - L_6 - L_7 - L_8$ from the return instruction at L_{10}. Once again, this allows pieces of code be analyzed separately from different contexts, providing context-sensitivity and thus more accurate results. The *ret 8* instruction at L_{18} returns to instruction L_6 (address of the top of the stack in the stack-context is $\perp - L_1 - L_2 - L_3$), and to L_{11} (address of the top of the stack in the stack-context is $\perp - L_6 - L_7 - L_8$).

In Figure 6, the function *Max* is invoked in the standard way, however it does not return in the typical manner. Instead of calling *ret*, the function pops the return address from the stack and jumps to that address (lines $L_{14} - L_{16}$).

At instruction L_{14}, the stack-string contexts are $\perp - L_1 - L_2 - L_3$ and $\perp - L_4 - L_5 - L_6$. The *pop* instruction at L_{14} pops the value from the top of the stack accordingly with the correct context, *i.e.*, L_4 for the context $\perp - L_1 - L_2 - L_3$ and L_7 for the context $\perp - L_4 - L_5 - L_6$. At instruction L_{15}, the stack-string contexts are $\perp - L_1 - L_2$ and $\perp - L_4 - L_5$ due to the *pop* instruction. The *add* instruction at L_{15} adds eight to *esp*, changing the stack-string contexts to $\perp$. L_{16} is an indirect jump to the address in *ebx*, and thus analysis continues at L_4 or L_7 depending of the current context.



```
Main:
                                 Max:
L1:   PUSH  4
L2:   PUSH  2                    L9:   MOV   eax, [esp+4]
L3:   CALL  Max                  L10:  MOV   ebx, [esp+8]
L4:   PUSH  6                    L11:  CMP   eax, ebx
L5:   PUSH  4                    L12:  JG    L14
L6:   CALL  Max                  L13:  MOV   eax, ebx
L7:   PUSH  0                    L14:  POP   ebx
L8:   CALL  ExitProcess          L15:  ADD   esp, 8
                                 L16:  JMP   ebx
```

Figure 6. Obfuscated return using *pop/jmp* instructions.

5. Concluding remarks

Context-sensitive interprocedural analysis when guided by a call graph is limited only to those binaries in which the call-graph can be constructed and in which stack manipulation is performed using standard compilation model(s). This precludes applying these analysis on obfuscated, optimized, or hand-written code. As a result, malware forensic tools based on such analysis can easily be thwarted.

We demonstrate how an abstract stack graph may be used as a replacement for the call-graph to perform interprocedural analysis. Since an ASG can be constructed for programs that obfuscate calls or use stack manipulation operations in non-standard ways, this adaptation makes it feasible to extend interprocedural analysis to a larger class of binaries. The adaptation is simple enough to directly impact interprocedural analysis algorithms based on call graph [Balakrishnan 2007, Guo et al. 2005].

In order to make ASG fully functional some work still have to be done, such as to extend them to identify situations where the stack pointer may be manipulated indirectly by passing data through memory. Besides that, it is easy to conclude that ASG is a powerfull technique to perform interprocedural analysis.

References

Amme, W., Braun, P., Zehendner, E., and Thomasset, F. (2000). Data dependence analysis of assembly code. In *Int. J. Parallel Proc.*

Balakrishnan, G. (2007). *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI.

Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction, Springer-Verlag*, pages 5–23, New York, NY.

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M. M., Lavoie, Y., and Tawbi, N. (2001). Static detection of malicious code in executable programs. In *Int. J. of Req. Eng.*

Boccardo, D. R., Manacero Jr., A., and Falavinha Jr., J. N. (2007). Implications of obfuscated code in the development of av detectors (in portuguese). In *XXXIV Seminario Integrado de Software e Hardware, SEMISH*, pages 2277–2291, Rio de Janeiro, Brazil.

Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proc. of the 12th USENIX Security Symposium*.

Cifuentes, C. and Fraboulet, A. (1997). Intraprocedural static slicing of binary executables. In *Proc. Int. Conf. on Software Maintenance (ICSM)*, pages 188–195.

Cifuentes, C., Simon, D., and Fraboulet, A. (1998). Assembly to high-level language translation. In *Proc. Int. Conf. on Software Maintenance (ICSM)*, pages 228–237.

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland.

Collberg, C. S. and Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation tools for software protection. *IEEE Trans. on Soft. Eng.*, 28(8):735–746.

Cousot, P. and Cousot, R. (2002). Modular static program analysis. In *CC'02*, pages 159–178.

Dalla Preda, M., Christodorescu, M., Jha, S., and Debray, S. (2007). A semantics-based approach to malware detection. In *Proc. Principles of Programming Languages (POPL)*, pages 377–388, New York, NY, USA. ACM.

Debray, S. K., Muth, R., and Weippert, M. (1998). Alias analysis of executable code. In *Proc. Principles of Programming Languages (POPL)*, pages 12–24.

Goodwin, D. W. (1997). Interprocedural dataflow analysis in an executable optimizer. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 122–133, New York, NY, USA. ACM.

Guo, B., Bridges, M. J., Triantafyllis, S., Ottoni, G., Raman, E., and August, D. (2005). Practical and accurate low-level pointer analysis. In *3nd Int. Symp. on Code Gen. and Opt.*

Kinder, J., Veith, H., and Zuleger, F. (2009). An abstract interpretation-based framework for control flow reconstruction from binaries. In *10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, pages 214–228.

Lakhotia, A. (1993). Constructing call multigraphs using dependence graphs. In *Proc. Principles of Programming Languages (POPL)*, pages 273–284.

Lakhotia, A., Kumar, E. U., and Venable, M. (2005). A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968.

Lakhotia, A. and Singh, P. K. (2003). Challenges in getting 'formal' with viruses. *Virus Bulletin*, pages 15–19.

Larus, J. R. and Schnarr, E. (1995). Eel: Machine-independent executable editing. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 291–300.

Linn, C. and Debray, S. (2003). Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conf. on Computer and Communications Security (CCS)*.

Milanova, A., Rountev, A., and Ryder, B. G. (2004). Precise call graphs for c programs with function pointers. In *Autom. Softw. Eng.*, pages 11(1): 7–26.

Muller-Olm, M. and Seidl, H. (2004). Precise interprocedural analysis through linear algebra. In *Proc. Principles of Programming Languages (POPL)*, pages 330–341.

Reps, T. and Balakrishnan, G. (2008). Improved memory-access analysis for x86 executables. In *Proc. Int. Conf. on Compiler Construction*.

Reps, T., Balakrishnan, G., and Lim, J. (2006). Intermediate-representation recovery from low-level code. In *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Charleston, SC.

Sagiv, M., Reps, T., and Horwitz, S. (1996). Precise interprocedural dataflow analysis with applications to constant propagation. In *Theor. Comput. Sci.*, pages 167(1–2): 131–170.

Schwarz, B., Debray, S. K., and Andrews, G. R. (2001). PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. Workshop on Binary Translation (WBT)*.

Sharir, M. and Pnueli, A. (1981). *Two approaches to interprocedural data flow analysis*. S.S. Muchnick and N.D. Jones, editors, Program Flow Analysis: Theory and Applications, chapter 7, pages 189-234. Prentice-Hall, Englewood Cliffs, NJ.

Srivastava, A. and Wall, D. (1993). A practical system for intermodule code optimization at linktime. *Journal of Programming Languages*, 1(I):1–18.

Szor, P. and Ferrie, P. (2001). Hunting for metamorphic. In *Proc. Virus Bull. Conf.*

Thompson, K. (1984). Reflections on trusting trust. *Commun. ACM*, 27(8):761–763.

Venable, M., Chouchane, M. R., Karim, M. E., and Lakhotia, A. (2005). Analyzing memory accesses in obfuscated x86 executables. In *DIMVA'05*, pages 1–18.

Venkitaraman, R. and Gupta, G. (2004). Static program analysis of embedded executable assembly code. In *CASES '04: Proc. of the 2004 Int. Conf. on Compilers, architecture, and synthesis for embedded systems*, pages 157–166, New York, NY, USA. ACM.

Walenstein, A., Mathur, R., Chouchane, M. R., and Lakhotia, A. (2006). Normalizing metamorphic malware using term-rewriting. In *6th IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM)*.

Wroblewski, G. (2002). General method of program code obfuscation. In *Proc. Int. Conf. on Software Engineering Research and Practice (SERP)*.

Zhang, W. and Ryder, B. G. (2007). Automatic construction of accurate application call graph with library call abstraction for java. *Journal of Software Maintenance*, pages 19(4): 231–252.