UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

TESE DE DOUTORADO

"Context-Sensitive Analysis of x86 Obfuscated Executables"

DAVIDSON RODRIGO BOCCARDO

Ilha Solteira – SP
outubro/2009

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"Context-Sensitive Analysis of x86 Obfuscated Executables"

DAVIDSON RODRIGO BOCCARDO

Orientador: Prof. Dr. Aleardo Manacero Júnior

Tese apresentada à Faculdade de Engenharia - UNESP – Campus de Ilha Solteira, para obtenção do título de Doutor em Engenharia Elétrica.
Área de Conhecimento: Automação.

Ilha Solteira – SP
outubro/2009

FICHA CATALOGRÁFICA

Elaborada pela Seção Técnica de Aquisição e Tratamento da Informação
Serviço Técnico de Biblioteca e Documentação da UNESP - Ilha Solteira.

Boccardo, Davidson Rodrigo.
B664c Context-sensitive analysis of x86 obfuscated executables / Davidson Rodrigo Boccardo. -- Ilha Solteira : [s.n.], 2009.
104 f.

Tese (doutorado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Automação, 2009

Orientador: Aleardo Manacero Júnior
Bibliografia: p. 99-104

1. Análise estática. 2. Interpretação abstrata. 3. Ofuscação de código.

unesp UNIVERSIDADE ESTADUAL PAULISTA
CAMPUS DE ILHA SOLTEIRA
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA

CERTIFICADO DE APROVAÇÃO

TÍTULO: Context-Sensitive Analysis of x86 Obfuscated Executables

AUTOR: DAVIDSON RODRIGO BOCCARDO
ORIENTADOR: Prof. Dr. ALEARDO MANACERO JUNIOR

Aprovado como parte das exigências para obtenção do Título de DOUTOR em ENGENHARIA ELÉTRICA , Área: AUTOMAÇÃO, pela Comissão Examinadora:

Prof. Dr. ALEARDO MANACERO JUNIOR
Departamento de Cienc Comp e Estatistica / Instituto de Biociências, Letras e Ciências Exatas de São José do Rio Preto

Prof. Dr. SERGIO AZEVEDO DE OLIVEIRA
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. FRANCISCO VILLARREAL ALVARADO
Departamento de Matemática / Faculdade de Engenharia de Ilha Solteira

Rodolfo Azevedo
Prof. Dr. RODOLFO JARDIM DE AZEVEDO
Instituto de Computação / Universidade Estadual de Campinas

Prof. Dr. ANDRÉ LUIZ MOURA DOS SANTOS
Centro de Ciências e Tecnologias / Universidade Estadual do Ceara

Data da realização: 09 de outubro de 2009.

Dedication

I dedicate this work to my family.

Acknowledgements

I would like to thank everyone of the Sao Paulo State University, Department of Computer Science and Statistics and Department of Electrical Engineering for teaching quality, dedication and incentive in academics.

I would like to thank my advisors Aleardo Manacero Júnior and Arun Lakhotia for their guidance and encouragement over these years. They have taught me how to achieve my goals and how to develop my ideas. Sincere thanks to André Luiz Moura dos Santos, Rodolfo Jardim de Azevedo, Sérgio Azevedo de Oliveira and Francisco Villarreal Alvarado for being on my dissertation committee.

I also would like to thank the Brazilian Ministry of Education (CAPES) for the financial support through of my doctorate.

A warm thanks for the colleagues from the Software Research Lab (SRL), especially Michael Venable and Anshuman Singh for dedicating their time to cooperate with the implementation and formal semantics while I was visiting the Center of Advanced Computer Studies at Lafayette.

A special thanks to my parents and family who helped me out on my problems and doubts in all these years. I would also like to thank the Biessenberger family for helping me while I was visiting the Center of Advanced Computer Studies at Lafayette.

Resumo

Ofuscação de código tem por finalidade dificultar a detecção de propriedades intrínsecas de um algoritmo através de alterações em sua sintaxe, entretanto preservando sua semântica. Desenvolvedores de software usam ofuscação de código para defender seus programas contra ataques de propriedade intelectual e para aumentar a segurança do código. Por outro lado, programadores maliciosos geralmente ofuscam seus códigos para esconder comportamento malicioso e para evitar detecção pelos anti-vírus.

Nesta tese, é introduzido um método para realizar análise com sensitividade ao contexto em binários com ofuscamento de chamada e retorno de procedimento. Para obter semântica equivalente, estes binários utilizam operações diretamente na pilha ao invés de instruções convencionais de chamada e retorno de procedimento.

No estado da arte atual, a definição de sensitividade ao contexto está associada com operações de chamada e retorno de procedimento, assim, análises interprocedurais clássicas não são confiáveis para analisar binários cujas operações não podem ser determinadas. Uma nova definição de sensitividade ao contexto é introduzida, baseada no estado da pilha em qualquer instrução. Enquanto mudanças em contextos à chamada de procedimento são intrinsicamente relacionadas com transferência de controle, assim, podendo ser obtidas em termos de caminhos em um grafo de controle de fluxo interprocedural, o mesmo não é aplicável para mudanças em contextos à pilha.

Um *framework* baseado em interpretação abstrata é desenvolvido para avaliar contexto baseado no estado da pilha e para derivar métodos baseado em contextos à chamada de procedimento para uso com contextos baseado no estado da pilha. O método proposto não requer o uso explícito de instruções de chamada e retorno de procedimento, porém depende do conhecimento de como o ponteiro da pilha é representado e manipulado.

O método apresentado é utilizado para criar uma versão com sensitividade ao contexto de um algoritmo para detecção de ofuscamento de chamadas de Venable *et al.*. Resultados experimentais mostram que a versão com sensitividade ao contexto do algoritmo gera resultados mais precisos, como também, é computacionalmente mais eficiente do que a versão sem sensitividade ao contexto.

Abstract

A code obfuscation intends to confuse a program in order to make it more difficult to understand while preserving its functionality. Programs may be obfuscated to protect intellectual property and to increase security of code. Programs may also be obfuscated to hide malicious behavior and to evade detection by anti-virus scanners.

We introduce a method for context-sensitive analysis of binaries that may have obfuscated procedure call and return operations. These binaries may use direct stack operators instead of the native *call* and *ret* instructions to achieve equivalent behavior. Since definition of context-sensitivity and algorithms for context-sensitive analysis has thus far been based on the specific semantics associated to procedure call and return operations, classic interprocedural analyses cannot be used reliably for analyzing programs in which these operations cannot be discerned. A new notion of context-sensitivity is introduced that is based on the state of the stack at any instruction. While changes in calling-context are associated with transfer of control, and hence can be reasoned in terms of paths in an interprocedural control flow graph (ICFG), the same is not true for changes in stack-context.

An abstract interpretation based framework is developed to reason about stack-context and to derive analogues of call-strings based methods for the context-sensitive analysis using stack-context. This analysis requires the knowledge of how the stack, rather the stack pointer, is represented and on the knowledge of operators that manipulate the stack pointer.

The method presented is used to create a context-sensitive version of Venable *et al.*'s algorithm for detecting obfuscated calls. Experimental results show that the context-sensitive version of the algorithm generates more precise results and is also computationally more efficient than its context-insensitive counterpart.

List of Figures

List of Tables

List of abbreviations

ASG	abstract stack graph
AST	abstract syntax tree
BDD	binary decision diagram
CFG	control flow graph
CG	call graph
COTS	commercial off-the shelf
DOC	detector of obfuscated calls
LIFO	last in first out
ICFG	interprocedural control flow graph
RIC	reduced interval congruence
SEH	structured exception handling
VSA	value set analysis

Mathematical notation

$\langle X, \sqsubseteq_X \rangle$	partially ordered set upon domain X
$\sqcup$	join operator
$\sqcap$	meet operator
$\wp(X)$	powerset of X
$\bigsqcup$	least upper bound (lub)
$\bigsqcap$	greatest lower bound (glb)
$\bot$	least element
$\top$	greatest element
lfp	least fixed point
(C, α, γ, A)	Galois connection between domains C and A
α	abstraction map
γ	concretization map
$\triangledown$	widening operator
X^*	set of finite sequences over X
$\epsilon \in X^*$	empty sequence
$(x\ i)$	represents the i^{th} element of the sequence x
$a.x$	inserts a in the head of the sequence x
$(rest\ a.x)$	removes a from the sequence $a.x$
$Y{\downarrow}X$	X^{th} element of the pair Y
$s \in \Sigma$	program state
$\sigma \in \Sigma^*$	trace (sequence of program states)
I	set of instructions
$\llparenthesis$	set of instructions that open contexts
$\rrparenthesis$	set of instructions that close contexts
ν	context string
Π	maps a trace to its context string
π	represents the effect of an individual program state on the accumulated context string
ν_k	k-context string
ν_ℓ	ℓ-context string

I^*	set of finite sequences over I
$\llparenthesis^*$	set of finite sequences of open contexts
$\llparenthesis^k$	k-abstraction of the set of finite sequences of open contexts
$\llparenthesis^\ell$	ℓ-abstraction of the set of finite sequences of open contexts
$\llparenthesis_T$	abstract syntax tree of the set of finite sequences of open contexts
α_k	maps a context string in $\llparenthesis^*$ to a k-context string in $\llparenthesis^k$
α_ℓ	maps a context string in $\llparenthesis^*$ to a ℓ-context string in $\llparenthesis^\ell$
ϕ	maps $\llparenthesis^*$ to $\llparenthesis_T$
$\hat{\llparenthesis}_{asm}$	open contexts for assembly programs
$\hat{\rrparenthesis}_{asm}$	close contexts for assembly programs
π_{asm}	version of function π for assembly programs
Π_{asm}	version of function Π for assembly programs
$\hat{\llparenthesis}^{\ell}_{asm}$	ℓ-abstraction of the set of finite sequences of open contexts for assembly programs
$\hat{\llparenthesis}^{k}_{asm}$	k-abstraction of the set of finite sequences of open contexts for assembly programs
F	concrete function F (F represents any given function)
$F^{\#}$	abstract function F

Contents

1 Introduction

An increase in the development of computer networks and internet technology has been noticed in recent years. Remote execution, distributed computing and code mobility have resulted in new computing abilities; however, they raise security and safety problems. Hosts and networks must be protected from malwares, and programs must be protected from malicious hosts. A malware may try to gain, steal or damage some information in a determined target (host). Software developers try to defend their program against malicious host attacks that usually aim to steal, modify or tamper with the code in order to take (economic) advantage of it. Both represent harmful threats to the security of computer networks.

Software protection and malware detection are two major applications of code obfuscation. A code obfuscation intends to confuse a program in order to make it more difficult to understand while preserving its functionality. Software developers use obfuscation techniques to hide intrinsic information of the algorithm in order to protect intellectual property and to increase security of code (by making it difficult for others to identify vulnerabilities). Malware writers, however, use obfuscation to hide malicious behavior in order to evade detection by anti-virus scanners (BOCCARDO; MANACERO JÚNIOR.; FALAVINHA JÚNIOR., 2007). Therefore, the design of techniques for analyzing obfuscated code is essentially due to the impossibility of determining if certain obfuscated code is malicious without its inspection.

Recently, research activity has increased in the area of binary analysis. These analyses have been motivated to port legacy applications to new platforms, link-time optimization of executables, verify whether an embedded application conforms to standards, identify security vulnerabilities that can be exploited by a hacker, analyze whether a binary may be malicious, and control flow reconstruction.

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). Hence, a safety analysis should be run at the binary level since the binary is the most accurate representation of a program behavior.

1.1 Motivation

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. Incorrect combination of call and return nodes creates spurious pathways in the information flow, 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.

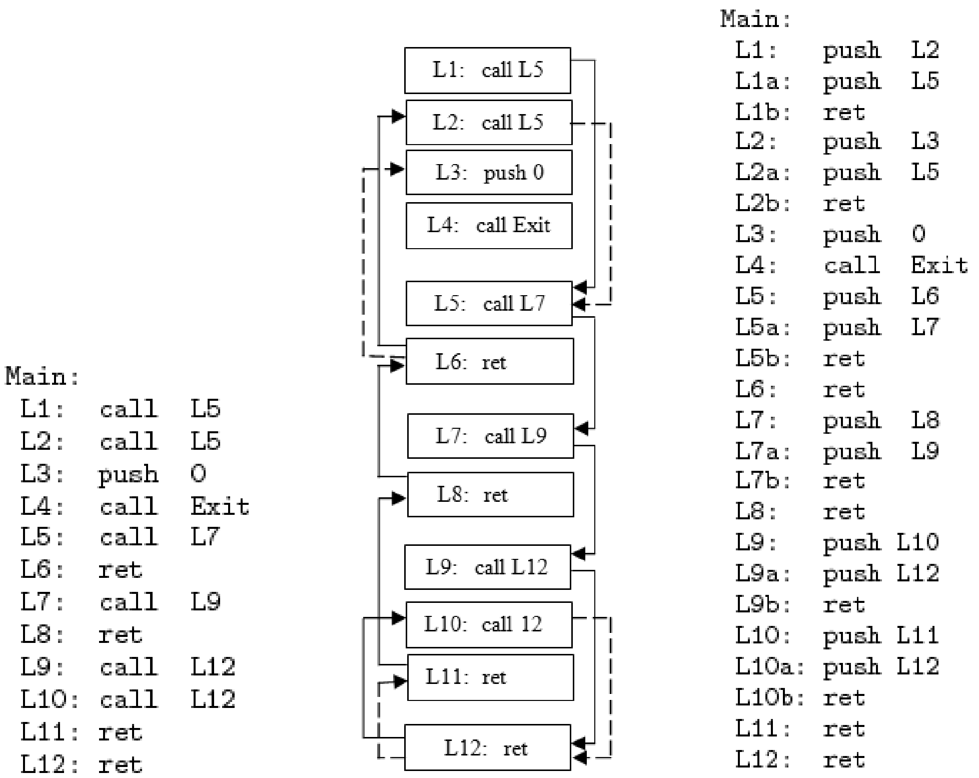
Classical interprocedural analysis may be performed either by procedure-inlining followed by an intraprocedural analysis, or by using the functional approach through procedure summaries, or by providing the calling-context using the call string approach (SHARIR; PNUELI, 1981). In the procedure-inlining approach, every call to a procedure is replaced by the body of that procedure. This technique is only feasible for non-recursive procedures, and the control-flow graph (CFG) may grow exponentially in terms of the nesting depth. In the functional approach through procedure summaries, an effect, a map from input values to the output values, for every procedure is calculated. The calculation of the effect requires the analysis of each procedure only a few times in case of recursive procedures. These procedure effects are then used to perform the analysis. In the call string approach, procedures are analyzed separately for different invocation flows to the beginning of its code (its calling contexts). This improves the analysis' precision for pieces of code that are executed more than once in different contexts. The analysis of different call sequences is made by simulating the call stack of an abstract machine which contains unclosed calling sequences.

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. These assumptions are often referred by researchers as a 'standard compilation model.' The 'standards' are compiler specific; they are not industry standards. Even for a given compiler, the 'standards' may vary depending on the optimization scheme selected. When a binary violates the 'standards', the current methods for context-sensitive interprocedural analysis fail.

Malware detection methods also make assumptions by observing the system calls made by the program (BERGERON et al., 1999). If the pattern of system calls matches a known malicious pattern of calls, then the file is deemed malicious. Symantec's Bloodhound technology, for example, uses classification algorithms to compare the system calls made by the program under inspection against a database of calls made by known viruses and clean programs (SYMANTEC, 1997). When a malware obfuscates its system calls, such malware detection methods fail.

This dissertation presents a method for performing context-sensitive analyses of binaries without requiring any particular convention for the layout of the procedure code in memory or the use of any particular conventions for procedure calls. More specifically, the proposed method does not require the use of explicit *call* and *ret* instructions, but depends upon the knowledge of how the stack pointer and instruction pointer are represented, which direction the stack grows, and the static identification of operators manipulating the stack pointer. Although it is not clear how one can obfuscate an instruction pointer, one may easily obfuscate a stack pointer by representing it using another register or a memory location. The proposed method requires that the register or memory location used to represent a stack pointer must be known. Similarly, even though in most architectures stack grows towards lower memory addresses, the convention can be altered if a programmer is representing his own stack. The intended analysis assumes the knowledge of this convention.

Figure 1(a) contains a sample code that presents the motivation. It is a simplified program, essentially showing only the call and return structure. Figure 1(b) shows the control flow graph (CFG) of this program. The graph is created by assuming that the target of a *call* instruction represents the entry point of a procedure and a *ret* instruction returns from call to the closest preceding entry point. The edges in this 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; PNUELI, 1981) via a mismatched return edge. In the graph, the type of arrow (solid or dashed) determines the correct pairing.



```
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
```

(a) Sample code.

(b) CFG.

```
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
```

(c) Obfuscated version.

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

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 execution to jump to the target address of the call. There are other ways to achieve the equivalent behavior (LAKHOTIA; KUMAR; VENABLE, 2005). Since such a program may not have a *call* instruction, it does not provide any clues in finding procedure entry points. Current technologies may infer that this program has only one procedure (consisting of the entire code) (IDAPRO, 2009). More importantly, most works on analysis of binaries will treat the *ret* instructions as though they are returning to the caller, thus generating an incorrect CFG. As a result, any analysis based on this CFG will also be incorrect. Such non-standard methods of making a call are explicitly used by malicious programs to defeat automated analysis (BOCCARDO; MANACERO JÚNIOR.; FALAVINHA JÚNIOR., 2007),(CHRISTODORESCU; JHA, 2003),(LAKHOTIA; SINGH, 2003),(SZÖR; FERRIE, 2001).

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; instead one may use move, increment, and decrement operations directly on the stack pointer to perform equivalent functions.

Binaries may not adhere to accepted conventions/assumptions because its creator, whether a compiler or a programmer, wishes to deter others from analyzing it. Such deliberate violation of assumptions, conventions, or for that matter standards, to make the binary harder is termed as obfuscation. It is becoming increasingly common to obfuscate code to protect intellectual property (LINN; DEBRAY, 2003),(COLLBERG; THOMBORSON; LOW, 1997),(WROBLEWSKI, 2002). However, the code may also be obfuscated to hide malicious intent (CHRISTODORESCU; JHA, 2003),(LAKHOTIA; SINGH, 2003),(SZÖR; FERRIE, 2001). Most malwares today use a variety of obfuscations to deter its disassembly, analysis, or reverse engineering.

The foundations of the approach presented in this dissertation come from previous work of our research group in analyzing programs with obfuscated calls (VENABLE et al., 2005),(LAKHOTIA; KUMAR; VENABLE, 2005). First, Lakhotia and Kumar (LAKHOTIA; KUMAR, 2004) described a way to detect stack related obfuscations using abstract stack graph. Their work addresses only the evaluation of operations that can be mapped to stack's *push* and *pop* instructions. Although that approach can be applied to several classes of programs, it fails in cases where the stack is manipulated through memory contents (registers, stack or heap). Venable *et al.* (VENABLE et al., 2005) developed an improved algorithm that could track stack manipulations where the stack pointer may be saved and restored in memory or registers. Venable *et al.*'s work assumed that the binary could not be decomposed into procedure boundaries. As a result, they essentially perform intraprocedural analysis on the entire program. The resulting analysis is expensive and leads to very significant over approximation. These limitations are overcome by the context-sensitive algorithm presented in this dissertation.

1.2 State-of-the-art

This section examines the state-of-the-art related to binary analysis focusing on interprocedural analysis and analysis of malicious/obfuscated programs, and also exposes their limitations.

Binary analyses have been motivated by several application fields such as to port legacy applications to new platforms (LARUS; SCHNARR, 1995), (CIFUENTES; FRABO-

ULET, 1997a, 1997b), (CIFUENTES; SIMON; FRABOULET, 1998), (MYCROFT, 1999), (AMME et al., 2000), link-time optimization (GOODWIN, 1997),(SCHWARZ; DEBRAY; ANDREWS, 2001),(DEBRAY; MUTH; WEIPPERT, 1998),(SRIVASTAVA; WALL, 1993), verify whether an embedded application conforms to standards (VENKITARAMAN; GUPTA, 2004), identify security vulnerabilities that can be exploited by a hacker (BERGERON et al., 1999, 2001), (BALAKRISHNAN, 2007), (MATTHEW et al., 2005), (REPS; BALAKRISHNAN; LIM, 2006), (BALAKRISHNAN; REPS, 2007), (REPS; BALAKRISHNAN, 2008), analyze whether a binary may be malicious (CHRISTODORESCU; JHA, 2003), (LAKHOTIA; KUMAR; VENABLE, 2005), (LAKHOTIA; KUMAR, 2004), (BACKES, 2004), (VENABLE et al., 2005), and control flow reconstruction (KINDER; VEITH; ZULEGER, 2009).

Since this dissertation is concerned with context-sensitive analysis, this section will focus on prior research related to the following categories: interprocedural analysis (in general), interprocedural analysis of binary programs, and analysis of malicious/obfuscated programs.

Context-sensitive interprocedural data-flow analysis of high-level languages has been an active area of research. Most of these efforts, represented by (REPS; HORWITZ; SAGIV, 1995), (SAGIV; REPS; HORWITZ, 1995), (COUSOT; COUSOT, 2002), (MÜLLER-OLM; SEIDL, 2004), (BALL; MILLSTEIN; RAJAMANI, 2005), (XIE; AIKEN, 2005), (GULWANI; TIWARI, 2007), have focused on special classes of problems for high-level languages. The general strategy they use falls within the two approaches proposed by Sharir and Pnueli (SHARIR; PNUELI, 1981), the call-string approach or the procedure summaries approach.

In the call-string approach data flow values are separated based on their calling context (SHARIR; PNUELI, 1981). The approximate call-string approach offers an efficient and flexible method for computing interprocedural analysis at the cost of precision. For non-recursive programs, call-strings are bounded by the length K, where K is the number of distinct call-sites in the longest call-chain. For recursive programs, and when the lattice of data flow values V is bounded, this method requires strings of length $K \times (|L| + 1)^2$. Recent work by Karkare and Khedker (KARKARE; KHEDKER, 2007),(KHEDKER; KARKARE, 2008) improves this bound to $K \times (|L| + 1)$. They achieved this improvement by terminating the call string construction when the data flow values stabilize, instead of using the length of the call-string.

In the procedure summary approach, a summary that represents the behavior of the procedure parametrized by any information about its input variables is calculated for each procedure. The construction of the summary is made by analyzing each procedure once or a few times in case of recursive procedures. Although this method guarantees precision, it is not efficient due to calculations of procedure summaries being high in time and complex in space (AHO et al., 2006). Moreover, there is no automatic way to efficiently construct or even represent these procedure summaries, and abstraction specific techniques are required. The original formalism proposed by Sharir and Pnueli (SHARIR; PNUELI, 1981) for computing procedure summaries was limited to finite lattices of dataflow facts.

Sagiv, Reps and Horwitz generalized the Sharir-Pnueli framework to build procedure summaries using context-free graph reachability (REPS; HORWITZ; SAGIV, 1995). Müller-Olm and Seidl (MÜLLER-OLM; SEIDL, 2004) subsumes the problem of linear constant propagation considered by Sagiv *et al.* (SAGIV; REPS; HORWITZ, 1995), but does not deal with aliasing. Ball *et al.* (BALL; MILLSTEIN; RAJAMANI, 2005) introduces auxiliary variables to record the input values of the procedure and uses predicates defined by both the program variables and the auxiliary variables. The result of the analysis can then be interpreted as a relation between the auxiliary variables (input values) and output values. However, the predicates might not be expressive enough to capture the precise summary. Xie and Aiken (XIE; AIKEN, 2005) specialized the summary generation for a particular problem in order to discover which contexts are relevant. They created summaries for checking correct use of locks using boolean satisfiability (SAT) procedures to enumerate all the relevant calling contexts. Recently, Gulwani and Tiwari (GULWANI; TIWARI, 2007) introduced a method for generating precise procedure summaries in the form of constraints on the input variables of the procedure that must be satisfied for some appropriate generic assertion involving output variables of the procedure to hold at the end of the procedure. Their method is based upon computing the weakest preconditions of a generic assertion. To guarantee termination of the analysis, they performed a second order unification to strengthen and simplify the weakest preconditions.

The call-string approach has two advantages as compared to the procedure summary approach. First, it is possible to deal with abstract domains of infinite cardinality. Secondly, it is easily possible to reduce the complexity of the analysis by selecting small values for k. The disadvantage is that analyses using the call string approach can be less precise than those using the procedure summary approach. By encoding the call strings into the analysis domain the updating of the call strings has to be done during the analysis, consequently, increasing the cost in time and space.

The classic interprocedural control flow graph (ICFG) based algorithms for computing function summary require *a priori* identification of procedure entries and exits. These methods cannot directly be adapted for our needs because call obfuscations prevent determination of procedure boundaries, violating the pre-requisite. Reps *et al.*'s weighted pushdown system based interprocedural analysis, which also computes function summaries (REPS et al., 2005), does not use ICFGs. Indeed our representation of context using the state of stack is analogous to Reps *et al.*'s use of stack of a pushdown automata (REPS et al., 2005). Lal and Reps improve the computation of the summary information (LAL; REPS, 2006) by taking advantage of the specific semantics associated to procedure call and return. Use of weighted pushdown systems for analysis of obfuscated binaries may be a productive avenue for future research.

The call-string approach follows the execution of a program. Algorithms based on this approach have classically been modeled to determine a change of context based on the semantics of procedure call/return and are described using ICFG. However, as we demonstrate from our adaptation, the call-string method does not require *a priori* knowledge of procedure boundaries, nor does it depend on the semantics of procedure invocation. As is done for context-sensitive computation of targets of indirect calls using points-to analysis, the call-graph used for call-string approach may be computed on the fly (EMAMI; GHIYA; HENDREN, 1994),(WHALEY; LAM, 2004),(ZHU, 2005),(ZHU; CALMAN, 2004),(WILSON; LAM, 1995). Determining transfer of control based on contents of memory or register is analogous to computing the points-to relation for higher languages. However, since memory addresses are linearly ordered, the resulting "points-to" sets in our problem context can be abstracted using a linear function. Thus, our method is analogous in spirit, though not in letter, to context-sensitive points-to analysis.

Interprocedural analysis of binaries has also received attention for post-compile time optimization (SRIVASTAVA; 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). 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, such as that supported by IDA Pro (IDAPRO, 2009). In contrast, we split the semantics of *call* and *ret* instructions. We model their affect on the "context" separate from their affect on the "transfer of control." The context is represented by the state of the stack and is modeled by an instruction's affect on the stack pointer. The transfer of control is analyzed using Balakrishnan and Reps' Value-Set Analysis (VSA) (BALAKRISHNAN;

REPS, 2004),(BALAKRISHNAN, 2007).

While our work is focused on deobfuscation of programs, there is an active body of work in the opposite direction. There has been significant work in obfuscation of programs with the intent to thwart static analysis (LINN; DEBRAY, 2003),(COLLBERG; 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 work by attacking various phases in the analysis of a binary (LAKHOTIA; SINGH, 2003). For example, 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, a metamorphic virus, 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; SINGH, 2003).

The rapid increase in using obfuscation techniques to spread malware has also triggered efforts to analyze obfuscated code. There have been efforts to use semantics based methods for detecting malware (DALLA PREDA et al., 2007),(CHRISTODORESCU; 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 specifically addresses analysis of obfuscated programs that do not conform to the standard compilation model.

1.3 Research Objectives

The goal of this research is to design a context-sensitive analysis based on program semantics and abstract interpretation framework resilient from *call* and *ret* obfuscations attacks. The objective is that such analysis may find a purpose in assisting obfuscated code analyzers by providing more reliable analysis results for obfuscated code.

1.4 Research Contributions

The main contributions of this dissertation may be summarized as follows:

- It introduces the concept of stack context, used *in lieu* of calling-context, to perform context-sensitive analysis of a binary program when the binary may be obfuscated or does not adhere to a standard compilation model.

- It adapts for use with stack context prior work on performing context-sensitive analysis using calling-contexts. Using abstract interpretation, a k-context abstraction is derived that generalizes Sharir and Pnueli's k-suffix call-strings abstractions (SHARIR; PNUELI, 1981). Unlike Sharir and Pnueli's formulation this generalization does not require transfer of control, an intrinsic part of semantics of procedure call and return. Similarly, an ℓ-context abstraction is derived that generalizes for use with stack-context Emami *et al.*'s strategy of abstracting calling-contexts by reducing cycles due to recursion (EMAMI; GHIYA; HENDREN, 1994), thus leading the way to the use of binary decision diagrams (BDDs) for making the analysis scalable (ZHU, 2005),(WHALEY; LAM, 2004).

- It presents a concrete application of the proposed method by creating a context-sensitive version of Venable *et al.*'s algorithm (VENABLE et al., 2005) that detects obfuscated calls. The resulting analysis is shown to be sound.

- It presents empirical results comparing the context-sensitive and insensitive versions of Venable *et al.*'s algorithm. The empirical results show that the context-sensitive analysis requires significantly less time and also yields better (more precise) results.

1.5 Organization

Chapter 2 provides the background necessary for designing the context-sensitive analysis of obfuscated executables. The background consists of domain theory, followed by a brief introduction to abstract interpretation. An overview of disassembly methods, code obfuscation, abstract stack graph (ASG) and Venable's algorithm (VENABLE et al., 2005) is also presented.

Chapter 3 introduces context-trace semantics, a trace semantics in which context is made explicit, followed by a generalization of Sharir and Pnueli's (SHARIR; PNUELI, 1981) k-suffix method for abstracting calling-contexts and a generalization of Emami *et al.*'s method of abstracting calling-contexts by reducing recursive cycles (EMAMI; GHIYA; HENDREN, 1994). It also presents our algorithm for analysis of binaries, which adapts the concept of context-trace to binaries and also summarizes the use of Balakrishnan and Reps Value-Set Analysis (BALAKRISHNAN; REPS, 2004),(BALAKRISHNAN, 2007) to help with determining transfer of control in assembly programs. Proving aspects are also shown by using the abstract interpretation theory.

Chapter 4 presents empirical evaluation of the context sensitive and insensitive versions of Venable's algorithm. The experiments show that the context-sensitive version of the algorithm generates more precise results and is also computationally more efficient than the context-insensitive version. Experimental results also show that ℓ-context abstraction is more efficient and precise than k-context abstraction.

Chapter 5 summarizes the major contributions of this dissertation and briefly describes future works that we would like to explore.

2 Preliminaries

This chapter includes the background necessary to develop our proposed algorithm. Section 2.1 presents the domain theory background, recalling the basic notions of sets, functions and partial orderings (DAVEY; PRIESTLEY, 1990),(GIERZ et al., 1980),(GRÄTZER, 1978). Section 2.2 provides the abstract interpretation framework (COUSOT; COUSOT, 1977, 1979). Section 2.3 presents disassembly methods and their relationship to static analysis. Section 2.4 provides information about code obfuscation. Section 2.5 summarizes the Abstract Stack Graph (ASG) from Lakhotia *et al.* (LAKHOTIA; KUMAR; VENABLE, 2005). Finally, section 2.6 presents a description of Venable's algorithm.

2.1 Domain Theory

2.1.1 Sets

A set is a collection of elements. The standard notation $x \in C$ declares that x is an element of the set C. The cardinality of a set C represents the number of its elements, and it is represented as $|C|$. Let C and A be two sets, where C is a subset of A represented as $C \subseteq A$, if every element of C belongs to A. When $C \subseteq A$ and there exists at least one element of A that does not belong to C, we say C is properly contained in A, represented $C \subset A$. Two sets C and A are equal, represented $C = A$, if C is a subset of A and vice-versa, *i.e.*, $C \subseteq A$ and $A \subseteq C$. Two sets C and A are different, represented $C \neq A$, if there exists an element in C (in A) that does not belong to A (to C). Let $\emptyset$ represent the empty set, namely the set without any element. In this case, for every element x we have $x \notin \emptyset$ and for every set C we have $\emptyset \subseteq C$.

The set $C \cup A$ represents elements belonging to C or to A, and it is called the union of C and A. Its definition is given as $C \cup A \overset{\text{def}}{=} \{x \mid x \in C \vee x \in A\}$. The set $C \cap A$ containing the elements belonging both to C and A identifies the intersection of C and A, and it is defined as $C \cap A \overset{\text{def}}{=} \{x \mid x \in C \wedge x \in A\}$. Two sets C and A are disjoint if their intersection is the empty set, *i.e.*, $C \cap A = \emptyset$. Let $C \setminus A$ denote the set of elements of C that do not belong to A, formally $C \setminus A \overset{\text{def}}{=} \{x \mid x \in C \land x \notin A\}$. The powerset $\wp(C)$ of a set C is defined as the set of all possible subsets of $C :\overset{\text{def}}{=} \{A \mid A \subseteq C\}$.

An important concept that can be defined in terms of sets is pairing. For two objects x and y, their pairing is written (x, y). Pairing is useful for defining another set construction, the product construction. For sets C and A, their product $C \times A$ is the set of all pairs built from $C \times A$, and it is defined as $C \times A \overset{\text{def}}{=} \{(x, y) \mid x \in C \land y \in A\}$. Both pairing and products can be generalized from their binary formats to n-tuples and n-products.

A form of union construction on sets that keeps the members of the respective sets C and A separate is called disjoint union, and it is defined as $C + A \overset{\text{def}}{=} \{(0, x) \mid x \in C\} \cup \{(1, y) \mid y \in A\}$. Ordered pairs are used to “tag” the members of C and A so that it is possible to examine a member and determine its origin.

2.1.2 Functions

Let C and A be two sets. A function f from C to A is a relation between C and A such that for each $x \in C$ there exists exactly one $y \in A$ such that $(x, y) \in f$. In this case, we write $f\ x = y$. Usually the notation $f : C \rightarrow A$ is used to denote a function from C to A, where C is the domain of f, and A is the co-domain of function f. The set $f\ X \overset{\text{def}}{=} \{f\ x \mid x \in X\}$ is the image of $X \subseteq C$ under f. In particular, the image of the domain, *i.e.*, $f\ C$, is called the range of f. The set $f^{-1}(X) \overset{\text{def}}{=} \{y \in C \mid f\ y \in X\}$ is called the reverse image of $X \subseteq A$ under f. If there exists an element $x \in C$ such that the element $f\ x$ is not defined, the function f is said to be partial, otherwise it is total.

Functions can be combined using the composition operation. For $f : C \rightarrow A$ and $g : A \rightarrow E$, $g \circ f$ is the function with domain C and codomain E such that $\forall x \in C$, $g \circ f\ x = g(f\ x)$. Composition of functions is associative: for f and g as given above and $h : E \rightarrow F$, $h \circ (g \circ f) = (h \circ g) \circ f$. The notation f^n represents the composition of the function f n times.

Functions may be classified by their mappings. Some classifications are:

- *one-one*: function $f : C \rightarrow A$ is an *one-one* (or *injective*) function if and only if $\forall x_1 \in C$ and $x_2 \in C$, $f\ x_1 = f\ x_2$ implies $x_1 = x_2$;
- *onto*: function $f : C \rightarrow A$ is an *onto* (or *surjective*) function if and only if $A = \{y \mid \textit{there exists some } x \in C \textit{ such that } f\ x = y\}$;
- *bijective*: function $f : C \rightarrow A$ is a *bijective* function if f is both *injective* and

surjective;

- *identity*: function $f : C \to C$ is an *identity* function for C if and only if $\forall x \in C$, $f\ x = x$.

2.1.3 Partial ordering

A *partially ordered set* (or *poset*) formalizes the intuitive concept of ordering, sequencing, or arranging of the elements of a set. A poset consists of a set along with a binary relation that describes, for certain pairs of elements in the set, the requirement that one of the elements must precede the other. For a domain C, a binary relation $\mathcal{R} \subseteq C \times C$ (or $\mathcal{R} : C \times C \to \mathbb{B}$) is represented by the infix symbol $\sqsubseteq_C$, or just $\sqsubseteq$, if the domain of usage is clear. For $x, y \in C$, we read $x \sqsubseteq y$ as “x is less defined than y.”

Definition 2.1.1 *A binary relation $\sqsubseteq: C \times C \to \mathbb{B}$ is a ‘partial ordering’ upon a set C, also represented by $\langle C, \sqsubseteq \rangle$, if and only if $\sqsubseteq$ is:*

- *reflexive: $\forall x \in C : x \sqsubseteq x$;*
- *antisymmetric: $\forall x, y \in C : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$;*
- *transitive: $\forall x, y, z \in C : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.*

A partially ordered set of elements can be represented by an acyclic graph, where $x \sqsubseteq y$ when an arc exists from element x to element y and x is beneath y on the page. For example, Figure 2 shows the finite partially ordered set graph (Hasse diagram) for $\wp(\{x, y, z\})$, which is partially ordered by subset inclusion.

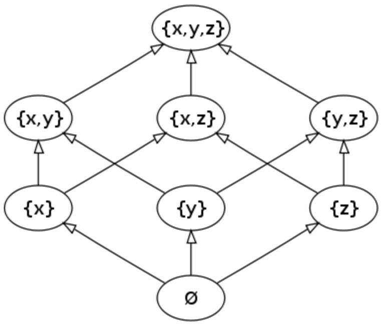


Figure 2: Hasse diagram of $\wp(\{x, y, z\})$.

Taking into account reflexivity, antisymmetry and transitivity, the graph of Figure 2 completely describes partial ordering.

Definition 2.1.2 *For a partial ordering $\sqsubseteq$ on C, $\forall x, y \in C$, the expression $x \sqcup y$ denotes the element in C (if it exists) such that:*

- $x \sqsubseteq x \sqcup y$ *and* $y \sqsubseteq x \sqcup y$.
- $\forall z \in C$, $x \sqsubseteq z$ *and* $y \sqsubseteq z \Rightarrow x \sqcup y \sqsubseteq z$.

The element $x \sqcup y$ is called the join of x and y. The join operation produces the smallest element that is larger than both of its arguments. A partial ordering might not have joins for all of its pairs. The partial ordering of Figure 2 has joins defined for all pairs.

An intersection set operation, the meet, is defined in a similar way. We write $x \sqcap y$ to denote the best defined element that is smaller than both x and y. The partial ordering of Figure 2 has meets defined for all pairs.

Definition 2.1.3 *A set C, partially ordered by $\sqsubseteq$, is a 'lattice' if and only if $\forall x, y \in C$, both $x \sqcup y$ and $x \sqcap y$ exist.*

The partial ordering of Figure 2 is an example of a lattice. For any set X, $\langle \wp(X), \subseteq \rangle$ is a lattice under the usual subset ordering $\subseteq$, in which join is the set union and meet is the set intersection. The concepts of join and meet can be generalized to operate over a (possibly infinite) set of arguments rather than just two.

Definition 2.1.4 *For a set C partially ordered by $\sqsubseteq$ and a subset X of C, $\bigsqcup X$ denotes the element of C (if it exists) that satisfies the following conditions:*

- $\forall x \in X$, $x \sqsubseteq_C \bigsqcup X$.
- $\forall y \in C$, $\forall x \in X$, $x \sqsubseteq_C y \Rightarrow \bigsqcup X \sqsubseteq_C y$.

The element $\bigsqcup X$ is called the *least upper bound* (*lub*) of X. In the following, the dual of *lub*, the *greatest lower bound (glb)* (denoted by $\bigsqcap X$), is defined.

Definition 2.1.5 *For a set C partially ordered by $\sqsubseteq$ and a subset X of C, $\bigsqcap X$ denotes the element of C (if it exists) that satisfies the following conditions:*

- $\forall x \in X$, $\bigsqcap X \sqsubseteq_C x$.

- $\forall y \in C, \forall x \in X, y \sqsubseteq_C x \Rightarrow y \sqsubseteq_C \bigsqcap X$.

Definition 2.1.6 *A set C partially ordered by $\sqsubseteq$ is a 'complete lattice' if and only if for all subsets X of C, both $\bigsqcup X$ and $\bigsqcap X$ exist.*

A complete lattice C is a partially ordered set $\langle C, \sqsubseteq \rangle$ such that all subsets have least upper bounds as well as greatest lower bounds in which the notation $\bot = \bigsqcap C$ is the *least element* and $\top = \bigsqcup C$ is the *greatest element.* The partial ordering of Figure 2 has the least element $\emptyset$ and the greatest element $\{x, y, z\}$. In programming language semantics, we use $A_\bot$ to define lifted domains, where $A_\bot = A \cup \{\bot\}$. The special value $\bot$ denotes *nontermination* or "no value at all."

Definition 2.1.7 *For a set C partially ordered by $\sqsubseteq_C$ and for a set A partially ordered by $\sqsubseteq_A$, the function $f : C \rightarrow A$ is monotone if for each $x, y \in C : x \sqsubseteq_C y$ implies that $f(x) \sqsubseteq_A f(y)$.*

2.1.4 Fixed points

Definition 2.1.8 *Let $f : C \rightarrow C$ be a function on a poset C, an element $c \in C$ is a fixed point of f if and only if $f\ c = c$. Further, c is the least fixed point of f if, for all $d \in C$, $f\ d = d$ implies $c \sqsubseteq d$.*

The least fixed point of a function $f : C \rightarrow C$, denoted $lfp^{\sqsubseteq}\ f$, is the fixed point which is less than or equal to all other fixed points according to some partial order. The notion of greatest fixed point, denoted $gfp^{\sqsubseteq}\ f$, is dually defined. Let us recall the well known Knaster-Tarski's fixed point theorem.

Theorem 2.1.9 *Let $(C, \sqsubseteq)$ be a complete lattice, then the least fixed point of a monotone function $f : C \rightarrow C$ exists and is defined to be $lfp^{\sqsubseteq}\ f = \bigsqcup_{n\ \in\ \mathbb{N}} f^n\ \bot$, where $f^n = f \circ f \circ ... \circ f$, n times.*

Hence, the least fixed point of a monotone function on a complete lattice can be computed as the limit of the iteration sequence obtained starting from the bottom of the complete lattice.

2.1.5 Galois connection

A Galois connection is a particular correspondence between two partially ordered sets. Sometimes calculations on a poset C may be too costly or even uncomputable, motivating the replacement of C by a simpler poset A.

Definition 2.1.10 *Two complete lattices $\langle C, \sqsubseteq_C \rangle$ and $\langle A, \sqsubseteq_A \rangle$ and two monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ such that:*

$$\forall c \in C : c \sqsubseteq_C \gamma(\alpha\ c) \textit{ and} \tag{2.1}$$

$$\forall a \in A : \alpha(\gamma\ a) \sqsubseteq_A a \tag{2.2}$$

form a Galois connection, equivalently represented by (C, α, γ, A).

This definition of Galois connection is equivalent to the one of adjunction between C and A, *i.e.*, $\exists \alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ such that $\forall a \in A, c \in C : \alpha\ c \sqsubseteq_A a \Leftrightarrow c \sqsubseteq_C \gamma\ a$. The function $\alpha(\gamma)$ is the left (right) adjoint of $\gamma(\alpha)$. It is enough to specify either α or γ map because in any Galois connection the left adjoint map α determines uniquely the right adjoint map γ and vice versa. Given the left adjoint α, the right adjoint is determined as $\gamma\ a = \bigsqcup_C \{c \in C \mid \alpha\ c \sqsubseteq_A a\}$; or given the right adjoint γ, the left adjoint is determined as $\alpha\ c = \bigsqcap_A \{a \in A \mid c \sqsubseteq_C \gamma\ a\}$. Two complete lattices, C and A, form a Galois connection (C, α, γ, A) iff α is additive or iff γ is coadditive. A Galois connection is called *Galois insertion* when α is surjective (or, equivalently, γ is injective). According to Galois theory, given a Galois connection (C, α, γ, A), a function $f^{\#} : A \rightarrow A$ is a sound approximation of a function $f : C \rightarrow C$ when $\alpha \circ f \sqsubseteq_A f^{\#} \circ \alpha$, or equivalently, $f \circ \gamma \sqsubseteq_C \gamma \circ f^{\#}$. When the abstraction and concretization maps are obvious from context, we denote $C \sqsubseteq A$ to mean that $\exists \alpha, \gamma$ such that (C, α, γ, A) is a Galois connection. We call $A_1 \sqsubseteq A_2 \sqsubseteq \ldots \sqsubseteq A_n$ a *chain* of Galois connections.

It is possible to have analyses for the individual components of a composite structure and we may want to combine them into an analysis for the whole structure. The following summarizes a catalogue of combination techniques of Galois connections from (NIELSON; NIELSON; HANKIN, 1999).

Sequential composition. A program analysis can be developed in stages starting with a complete lattice $(C_0, \sqsubseteq)$ fairly closely related to the semantics and introducing a complete lattice $(C_1, \sqsubseteq)$ related to C_0 by a Galois connection $(C_0, \alpha_1, \gamma_1, C_1)$. This step can then be repeated n times, stopping when certain analysis with the required computational properties is obtained. Formally, let $(C_0, \alpha_1, \gamma_1, C_1)$ and $(C_1, \alpha_2, \gamma_2, C_2)$ represent Galois connections, then $(C_0, \alpha_2 \circ \alpha_1, \gamma_2 \circ \gamma_1, C_2)$ is also a Galois connection, where $\alpha_2 \circ \alpha_1$ and $\gamma_2 \circ \gamma_1$ form an adjunction, *i.e.*, $\alpha_2(\alpha_1\ c_0) \sqsubseteq c_2 \Leftrightarrow \alpha_1\ c_0 \sqsubseteq \gamma_2\ c_2 \Leftrightarrow c_0 \sqsubseteq \gamma_1(\gamma_2\ c_2)$.

Independent attribute. This method is applied when a combination of several analyses of *individual* components of a structure is desired. Let $(C_1, \alpha_1, \gamma_1, A_1)$ and $(C_2, \alpha_2, \gamma_2, A_2)$ be Galois connections. The *independent attribute* will then give rise to a Galois connection $(C_1 \times C_2, \alpha, \gamma, A_1 \times A_2)$, where $\alpha(c_1, c_2) = (\alpha_1\ c_1, \alpha_2\ c_2)$, and $\gamma(a_1, a_2) = (\gamma_1\ a_1, \gamma_2\ a_2)$. To verify that this defines a Galois connection we calculate

$$\alpha(c_1, c_2) \sqsubseteq (a_1, a_2) \Leftrightarrow (\alpha_1\ c_1, \alpha_2\ c_2) \sqsubseteq (a_1, a_2)$$

$$\alpha_1\ c_1 \sqsubseteq a_1 \wedge \alpha_2\ c_2 \sqsubseteq a_2 \Leftrightarrow c_1 \sqsubseteq \gamma_1\ a_1 \wedge c_2 \sqsubseteq \gamma_2\ a_2$$

$$(c_1, c_2) \sqsubseteq (\gamma_1\ a_1, \gamma_2\ a_2) \Leftrightarrow (c_1, c_2) \sqsubseteq \gamma(a_1, a_2)$$

where α and γ forms an adjunction if and only if $(C_1 \times C_2, \alpha, \gamma, A_1 \times A_2)$ is a Galois connection.

Total function space. If C is a complete lattice then so is the *total function space* $S \rightarrow C$ for S being a set (NIELSON; NIELSON; HANKIN, 1999). Let (C, α, γ, A) be a Galois connection and S be a set, then we obtain a Galois connection $(S \rightarrow C, \alpha', \gamma', S \rightarrow A)$ by taking $\alpha'\ f = \alpha \circ f$ and $\gamma'\ g = \gamma \circ g$, in which f is a function from $S \rightarrow C$ and g is a function from $S \rightarrow A$. To verify that this defines a Galois connection we have that α' and γ' are monotone functions because α and γ are monotone functions; furthermore $\gamma'(\alpha'\ f) = \gamma \circ \alpha \circ f \sqsupseteq f$ and $\alpha'(\gamma'\ g) = \alpha \circ \gamma \circ g \sqsubseteq g$.

Monotone function space. The *monotone function space* between two complete lattices is a complete lattice (NIELSON; NIELSON; HANKIN, 1999). Let $(C_1, \alpha_1, \gamma_1, A_1)$ and $(C_2, \alpha_2, \gamma_2, A_2)$ be Galois connections. We then obtain the Galois connection $(C_1 \rightarrow C_2, \alpha, \gamma, A_1 \rightarrow A_2)$ by taking $\alpha\ f = \alpha_2 \circ f \circ \gamma_1$ and $\gamma\ g = \gamma_2 \circ g \circ \alpha_1$, where α and γ functions are monotone because α_2 and γ_2 are monotone functions. Next, we calculate $\gamma(\alpha\ f) = (\gamma_2 \circ \alpha_2) \circ f \circ (\gamma_1 \circ \alpha_1) \sqsupseteq f$, and $\alpha(\gamma\ g) = (\alpha_2 \circ \gamma_2) \circ g \circ (\alpha_1 \circ \gamma_1) \sqsubseteq g$ using the monotonicity of $f : C_1 \rightarrow C_2$ and $g : A_1 \rightarrow A_2$.

So far the constructions have shown how to combine Galois connections dealing with individual components of the data into Galois connections dealing with composite data. The next method shows how two analyses dealing with the same data can be combined into one analysis. This amounts to performing two analyses in parallel.

Direct product. Let $(C, \alpha_1, \gamma_1, A_1)$ and $(C, \alpha_1, \gamma_1, A_2)$ be Galois connections. The *direct product* of the two Galois connections will be the Galois connection $(C, \alpha, \gamma, A_1 \times A_2)$, where α and γ are given by $\alpha\ c = (\alpha_1\ c, \alpha_2\ c)$ and $\gamma(a_1, a_2) = \gamma_1\ a_1 \sqcap \gamma_2\ a_2$. To verify that this defines a Galois connection we have:

$$\alpha\ c \sqsubseteq (a_1, a_2) \Leftrightarrow \alpha_1\ c \sqsubseteq a_1 \wedge \alpha_2\ c \sqsubseteq a_2 \Leftrightarrow c \sqsubseteq \gamma_1\ a_1 \wedge c \sqsubseteq \gamma_2\ a_2 \Leftrightarrow c \sqsubseteq \gamma(a_1, a_2)$$

where α and γ is an adjunction if and only if $(C, \alpha, \gamma, A_1 \times A_2)$ is a Galois connection.

2.2 Abstract Interpretation

Abstract interpretation is a unified framework for static program analysis (COUSOT; COUSOT, 1977, 1979). Static program analysis is an extensively used technique for automatically predicting the set of values or behaviors arising dynamically at runtime when executing a program on a computer. Since these sets are in most cases not computable, approximations are necessary. This framework allows the systematic derivation of data flow analyses and provides methods to prove their correctness and termination. In most general terms, abstract interpretation is a theory of fixed point approximation.

Abstract interpretation has been applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science, such as the semantics, the proof, the static analysis, and the verification of safety and security of software or hardware computer systems. In particular, static analysis by abstract interpretation can automatically infer dynamic properties of computer programs. For the past years, it has been very successful in automatically verifying complex properties of real-time and safety-critical embedded systems (COUSOT, 1996).

An analysis may be derived in stages, starting from concrete semantics to abstracted semantics that satisfies computational properties. Soundness of the analysis is demonstrated by creating Galois connections between the domains of the successive stages. Galois connections may also be used to order two or more analyses on their precision.

The concrete semantics, represented by **S**, can be expressed by the least fixed point of a monotone function $f : C \rightarrow C$, in which C is the concrete domain, *i.e.*, the poset of mathematical objects on which the program runs. The ordering relation encodes relative precision, in which $c_1 \sqsubseteq c_2$ denotes that c_1 is a more precise (concrete) description than c_2. Approximation is encoded by an abstract domain $\langle A, \sqsubseteq_A \rangle$ which is a poset of abstract values that represent some approximated properties of concrete objects. Also in the abstract domain the ordering relation models relative precision, in which $a_1 \sqsubseteq a_2$ denotes that a_1 is a better approximation (*i.e.*, more precise) than a_2.

Concrete and abstract domains are related through a Galois connection (C, α, γ, A). The function $\alpha : C \rightarrow A$ is called the abstraction function, and the function $\gamma : A \rightarrow C$ is the concretization function. We say that A is an abstraction of C and C is a concretization of A. The abstraction and concretization maps express the meaning of the abstraction process, where $\alpha\ c$ is the abstract representation of c, and $\gamma\ a$ represents the concrete meaning of a. Thus, $\alpha\ c \sqsubseteq_A a$ and, equivalently, $c \sqsubseteq_C \gamma\ a$ means that a is a sound approximation in A of c.

Given a Galois connection (C, α, γ, A), the abstract semantics, represented by $\mathbf{S}^{\#}$, is obtained by replacing the function $f : C \rightarrow C$, used to compute $\mathbf{S}$, with a monotone abstract semantic function $f^{\#} : A \rightarrow A$ that correctly mimics the behavior of f in the domain properties expressed by A. Accordingly with the Galois theory, the function $f^{\#} : A \rightarrow A$ is a sound approximation of a function $f : C \rightarrow C$ when $\alpha \circ f \sqsubseteq_A f^{\#} \circ \alpha$, or equivalently, $f \circ \gamma \sqsubseteq_C \gamma \circ f^{\#}$. The abstract semantics are also a least fixed point of $f^{\#}$ (thanks to the fixed point transfer theorem of (TARSKI, 1955)). A function $f^{\#}$ defined as $\alpha \circ f \circ \gamma$ is a safe approximation of f. Further, when (C, α, γ, A) is a Galois insertion, the function $\alpha \circ f \circ \gamma$ is also the best approximation of f.

Following (COUSOT; COUSOT, 2004), a program may be formalized as a graph or as a transition system $\tau = \langle \Sigma, \Sigma_i, t \rangle$, where Σ is a set of states, $\Sigma_i \subseteq \Sigma$ denotes the set of initial states and $t \subseteq \Sigma \times \Sigma$ defines the transition relation between states. A finite partial trace $\sigma \in \Sigma^*$ is a sequence of program states $s_0 ... s_n$ such that $s_0 \in \Sigma_i$ and for all $i \in [0, n) :\ (s_i, s_{i+1}) \in t$. The set of all such finite partial traces is called the *trace semantics* of the program and is given by the least fixed point of the semantic transformer $\mathcal{F}$:

$$\mathcal{F}\ T = \Sigma_i \cup \{\sigma.s.s' \mid \sigma.s \in T \wedge \langle s, s' \rangle \in t\}$$

where T is a set of finite partial traces. The domain of this trace semantics is $\wp(\Sigma^*)$.

Hence, the least fixed point (*lfp*) of $\mathcal{F}$ is as follows:

$$lfp_{\perp}^{\sqsubseteq}\ \mathcal{F} = \bigsqcup_{n \geq 0} \mathcal{F}^n\ \perp$$

If *Abs* is an abstract domain, *i.e.*, $(\wp(\Sigma^*), \alpha, \gamma, Abs)$ is a Galois connection, then $\mathcal{F}^{\#} : Abs \rightarrow Abs$ is sound *w.r.t* $\mathcal{F}$. It can be shown that $\forall n \geq 0\ :\ \alpha(\mathcal{F}^n\ \perp) \sqsubseteq_{Abs} \mathcal{F}^{\#^n}\ \perp$. Therefore, it follows that $\alpha\ (lfp_{\perp}^{\sqsubseteq}\ \mathcal{F}) = \alpha\ (\bigsqcup_{n \geq 0} \mathcal{F}^n\ \perp) = \bigsqcup_{n \geq 0} \alpha(\mathcal{F}^n\ \perp) = \bigsqcup_{n \geq 0} \mathcal{F}^{\#^n}\ \perp = lfp_{\perp}^{\sqsubseteq}\ \mathcal{F}^{\#^n}$. This is known as the *fixed point transfer* theorem (COUSOT; COUSOT, 1979).

The usual Kleene iteration sequence is known to converge to the least fixed point of $\mathcal{F}$, however the sequence need not stabilize. Cousot and Cousot (COUSOT; COUSOT, 1977) describe ways to ensure that the sequence stabilizes, by parameterizing the sequence on the operator ∇, called *widening* operator. The *widening* operator determines the least upper bound operation. The precision and cost of the approximated fixed point is related to the choice of the *widening* operator.

The derivation of a static analyzer using abstract interpretation may be summarized as follows. The program state is classically represented by the domain $\Sigma = I \times Store$, where I is the domain of instructions and *Store* is the domain of stores. The analysis is derived from a chain of Galois connections linking the semantic domain $\wp((I \times Store)^*)$ to the analysis domain $I \rightarrow Abstore$, where *Abstore* is an abstraction of stores. The derivation may have the following stages:

1. The set $\wp((I \times Store)^*)$, called set of traces, is approximated to trace of sets, represented by $(\wp(I \times Store))^*$, *i.e.*, $\wp((I \times Store)^*) \sqsubseteq (\wp(I \times Store))^*$.
2. The trace of sets is equivalent to $(I \rightarrow \wp(Store))^*$. This sequence of mapping of instructions to set of stores can be approximated to $I \rightarrow \wp(Store)$.
3. Finally, a Galois connection between $\wp(Store)$ and *Abstore* completes the analysis.

2.2.1 Examples of concrete and abstract store domains

Consider the store domain given by the powerset of integers $\langle \wp(\mathbb{Z}), \subseteq \rangle$ and assume we are interested in the sign of a given integer number. Figure 3 presents some possible abstractions of $\wp(\mathbb{Z})$ expressing properties on the sign of integers.

The abstraction and concretization functions are clearly shown (e.g., $\alpha(\{0, -1, -2\}) = 0-$, $\alpha(\{-1, 2\}) = \mathbb{Z}$, while $\gamma(0+) = \{z \geq 0\}$ and $\gamma(0-) = \{z < 0\}$). It is easy to see that $A+, A-$ and *Sign* are in Galois connection with $\wp(\mathbb{Z})$.

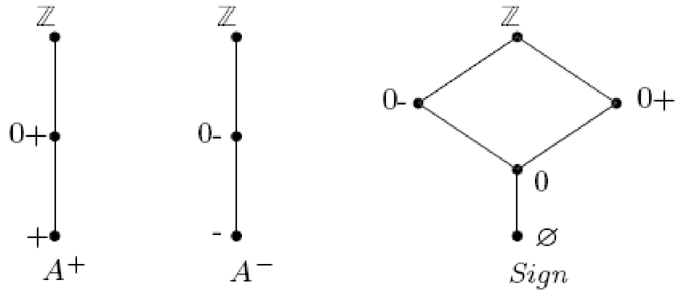


Figure 3: Abstractions of $\wp(\mathbb{Z})$.

A non-trivial well known abstraction for the powerset of integers is given by the abstract domain of intervals represented by $\langle Interval, \sqsubseteq_I \rangle$ (NIELSON; NIELSON; HANKIN, 1999). The elements of the *Interval* domain are defined by the following:

$$Interval \overset{\text{def}}{=} \{\bot\} \cup \{[l,h] \mid l \leq h, l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}\}$$

where the standard ordering on integers is extended to $\mathbb{Z} \cup \{+\infty, -\infty\}$ by setting $-\infty \leq +\infty$ and that for all $z \in Z : z \leq +\infty$ and $-\infty \leq z$. The idea is that the abstract element $[l,h]$ corresponds to the interval from l to h, including the end points if they are in $\mathbb{Z}$, while $\bot$ denotes the empty interval. Intuitively, an interval int_1 is smaller than an interval int_2, represented $int_1 \sqsubseteq_I int_2$, when int_1 is contained in int_2 . Formally we have:

- for all $int \in Interval : \bot \sqsubseteq_I int \sqsubseteq (-\infty, +\infty)$;
- for all $l_1, l_2 \in \mathbb{Z} \cup \{-\infty\}, h_1, h_2 \in \mathbb{Z} \cup \{+\infty\} : [l_1, h_1] \sqsubseteq_I [l_2, h_2] \Leftrightarrow l_2 \leq l_1 \wedge h_1 \leq h_2$;

Figure 4 represents the abstract domain of intervals. $(\wp(\mathbb{Z}), \alpha_I, \gamma_I, Interval)$ is a Galois insertion where the abstraction $\alpha_I : \wp(\mathbb{Z}) \rightarrow Interval$ and concretization $\gamma_I : Interval \rightarrow \wp(\mathbb{Z})$ maps are defined as follows:

$$\alpha_I(S) = \begin{cases} \bot & \text{if } S = \{\ \} \\ [l,h] & \text{if } min(S) = l \wedge max(S) = h \\ (-\infty, h] & \text{if } \nexists min(S) \wedge max(S) = h \\ [l, +\infty) & \text{if } min(S) = l \ \wedge \nexists max(S) \\ (-\infty, +\infty) & \text{if } \nexists min(S) \ \wedge \nexists max(s) \end{cases}$$

$$\gamma_I(int) = \begin{cases} \{\ \} & \text{if } int = \bot \\ \{z \in \mathbb{Z} \mid l \leq z \leq h\} & \text{if } int = [l,h] \\ \{z \in \mathbb{Z} \mid z \leq h\} & \text{if } int = (-\infty, h] \\ \{z \in \mathbb{Z} \mid z \geq l\} & \text{if } int = [l, +\infty) \\ \mathbb{Z} & \text{if } int = (-\infty, +\infty) \end{cases}$$

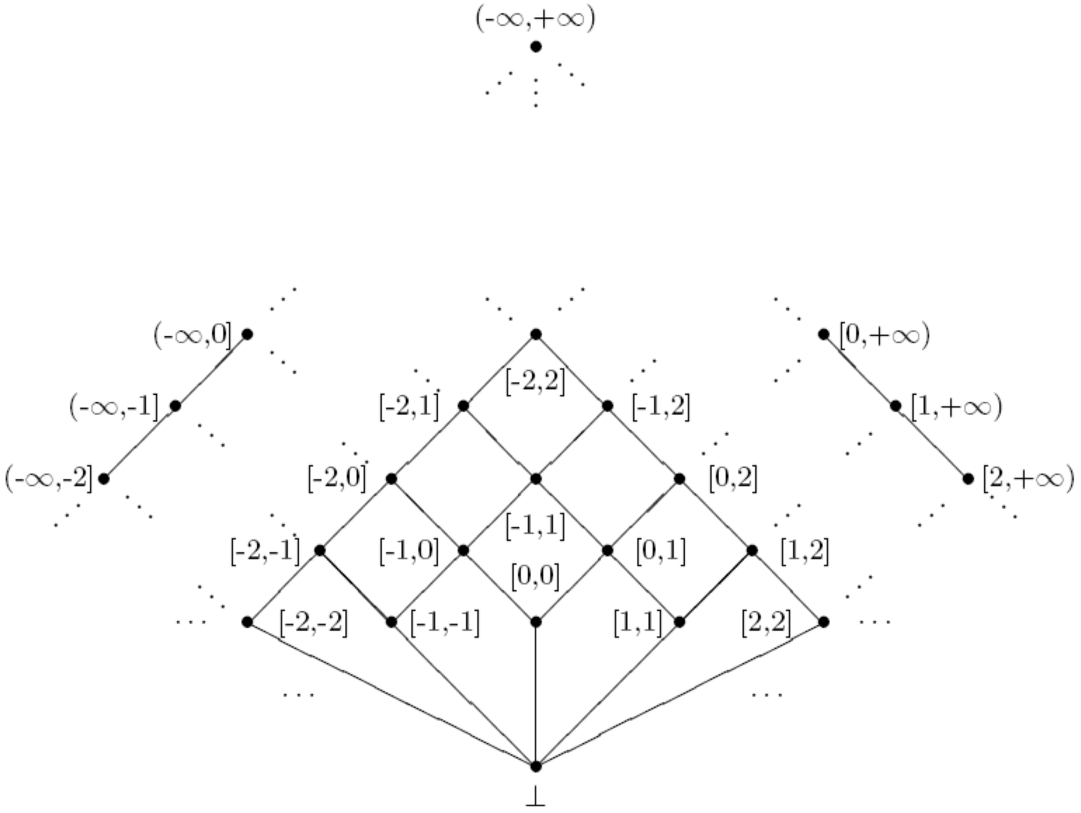


Figure 4: The Interval abstract domain.

where $l, h \in \mathbb{Z}$, and $S \in \wp(\mathbb{Z})$. For example, the set $\{2,\ 5,\ 8\}$ is abstracted in the interval $[2,\ 8]$, while the infinite set $\{z \in \mathbb{Z} \mid z \geq 10\}$ is abstracted in the interval $[10, +\infty)$. It is possible to prove that $\langle Interval, \sqsubseteq_I \rangle$ is a complete lattice with a top element given by $(-\infty, +\infty)$, a bottom element given by $\bot$, *glb* $\sqcap_I$ defined as $[l_1, h_1] \sqcap_I [l_2, h_2] = [max(\{l_1, l_2\}), min(\{h_1, h_2\})]$, and *lub* $\sqcup_I$ defined as $[l_1, h_1] \sqcup_I [l_2, h_2] = [min(\{l_1, l_2\}), max(\{h_1, h_2\})]$.

For example, $[2, 10] \sqcap_I [5, 20] = [5, 10]$ and $[2, 10] \sqcap_I (-\infty, 5] = [2, 5]$, while $[2, 10] \sqcap_I [20, 25] = \bot$. Thus, the *glb* of a set of intervals returns the larger interval contained in all elements of the sets. Also, $[2, 10] \sqcup_I [5, 20] = [2, 20]$ and $[2, 10] \sqcup_I (-\infty, 5] = (-\infty, 10]$, while $[2, 10] \sqcup_I [20, 25] = [2, 25]$. Hence, the *lub* of a set of intervals returns the smallest interval that contains all elements of the sets.

The abstract domain of intervals and the abstract domain of sign can be compared with respect to their degree of precision. In particular, *Interval* provides a more precise representation of the powerset of integers than *Sign* does, meaning *Interval* $\sqsubseteq$ *Sign*.

2.3 Disassembly

At the heart of static analysis for binaries lies disassembly. Disassembly is the process of reverse-engineering an executable to recover the set of assembly instructions stored within the executable image. Many disassemblers also perform additional functions, such as reconstructing the control-flow graph from the disassembled output. Disassembly, however, is not an exact science, and there are many different approaches, each with its own set of trade-offs.

A common and simple disassembly approach, known as linear sweep, requires the disassembler to start processing from the beginning (of the .text segment) and disassemble one instruction at a time until the end is reached (SCHWARZ; DEBRAY; ANDREWS, 2002), (VINCIGUERRA et al., 2003). This simplistic method fails when data is embedded within the code as shown below.

```
Main:
    ...
    JMP  FUNC
    DB   0E8h
    FUNC:
    ...
```

A linear sweep disassembler would simply disassemble the data as if it were code. If the data happens to match a real assembly instruction, then the disassembler would continue without a clue about the error. Otherwise, the disassembler may skip over the byte that is known not to be code and try the next byte. An interesting artifact from this is that the disassembler tends to eventually get “back on track.” That is, it will begin disassembling actual instructions correctly at some later point in time (LINN; DEBRAY, 2003). In the meantime, many instructions will be incorrectly disassembled, and the disassembler will not backtrack to correct the problem.

Another popular disassembly method is known as recursive traversal (LINN; DEBRAY, 2003), (SCHWARZ; DEBRAY; ANDREWS, 2002), (VINCIGUERRA et al., 2003). This method tries to overcome the weaknesses of linear sweep by following the control-flow of the program when disassembling, instead of disassembling in a straight line from start to finish. Thus, if the programmer inserts data in the middle of the instruction stream (as in the above example), the disassembler will jump over the data and will not be fooled. However, there are other situations that do cause problems. In order to know where to resume disassembly after a jump, the disassembler must be able to deduce the possible jump targets statically, which is not always an easy task. Indirect jumps, in particular, pose a challenge, and if the target of an indirect jump cannot be determined, there is the possibility of code not being disassembled (that is, the disassembler believes the code is data).

Another tricky situation arises when one of the jump targets is data. Take, for instance, the code below:

```
Main:
  CMP   eax, eax
  JE    FUNC+1
  JMP   FUNC
  ...
FUNC:
  DB    0E8h
  PUSH  0
  CALL  ExitProcess
```

Mentally tracing through the execution of the code reveals that the instruction 'JE $FUNC + 1$' (jump if equal) always transfers control to '$FUNC + 1$' because the two operands being compared (*eax* and *eax*) are always equal. Thus, the instruction 'JMP $FUNC$' is not reachable and will never be executed. This is a good thing too, since there is no code at '$FUNC$', only data. This style of trickery is known as an opaque predicate.

Other disassembly methods have been introduced and some variations of the previous techniques have been proposed. Worth noting is interactive disassembly, which requires a human to make key decisions to improve the disassembly (VINCIGUERRA et al., 2003). The disassembler may perform an initial disassembly, and the user may override some of the decisions made by the disassembler or may instruct the disassembler to disassemble areas of code that the disassembler misinterpreted for data. Also possible is running the suspect program in a debugger while allowing the user to guide the disassembly process. With this sort of dynamic analysis, a more precise disassembly can be achieved, but at the expense of more human interaction.

2.4 Code Obfuscation

Code obfuscation is a technique for altering the structure of a program's instruction set in order to make the meaning less apparent and thus harder for someone to reverse engineer (LINN; DEBRAY, 2003),(COLLBERG; THOMBORSON, 2002). Obfuscation has found a purpose in many legitimate applications where it is sometimes used to deter reverse-engineering by competitors who may be interested in learning proprietary formats (COLLBERG; THOMBORSON; LOW, 1997), (LINN; DEBRAY, 2003). Simple encryption of the code is not enough, since the code must be decrypted before it can be executed and would then become vulnerable to reverse engineering. Thus, obfuscation has proven to be a useful technique to hide important information within code.

On the other hand, like any technology, obfuscation can be used in a more malicious context. Specifically, malicious code writers frequently turn to code obfuscation to prevent analysis by researchers. Code obfuscation may be particularly effective at deterring static analysis, since dynamic analysis analyzes behavior and behavior is typically left unchanged by obfuscators. Taken from (LAKHOTIA; SINGH, 2003), the static analysis process can generally be broken down into five phases: disassembly, procedure abstraction, control-flow graph generation, dataflow analysis, and property verification. At the simplest level, they may attack the disassembler such that it generates incorrect assembly. On the other extreme, they may introduce spurious control flow paths, rendering static analysis useless by creating very large approximations. 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 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; SINGH, 2003).

Many obfuscation techniques use what is referred to as opaque predicates. An opaque predicate is a program variable whose value is known before run-time and is used in such a way that the obfuscator can predict the flow of control, but an analyst may not (or can not do so easily) (COLLBERG; THOMBORSON; LOW, 1997). The obfuscator uses the opaque predicate as an argument to a conditional branch and is then able to decide which path is chosen based on whether the predicate is true of false. A good opaque predicate is one that contains a value that is computationally expensive to determine. If an analyst cannot determine the value of the predicate in a conditional, then the only option is to assume both paths are possible. Among other things, opaque predicates provide a way to attack the control-flow graph generation phase of static analysis. By using these contrived branch statements, one can force an analyzer to add unnecessary edges to the control-flow graph, thus degrading the results.

Another useful tactic is to insert irrelevant code or data, frequently with the help of opaque predicates. Take the following piece of code as an example:

```
if x < 0 then
  y = true
else
  y = false
```

It is obvious that, after execution, y may have two possible values, either true or false. If the variable x, however, is an opaque predicate (it's value is known before run-time), then clearly y may have only one value after execution- true if x is less than zero, false otherwise. Such an attack is aimed at the dataflow analysis phase and can decrease the precision of the analysis results.

A variation of the above theme is to insert junk data, as shown in the below code. In this attack, the goal is to trick the disassembler into incorrectly disassembling data as if it were code. This attack is clearly aimed at the disassembly phase of static analysis.

```
Main:
  CMP   eax, eax
  JE    FUNC+1
  JMP   FUNC
  ...
FUNC:
  DB    0E8h
  PUSH  0
  CALL  ExitProcess
```

Another interesting obfuscation technique is to break the common assumptions surrounding branch instructions. In most compiler generated code, a *call* instruction is paired with a *ret* instruction, and the *ret* instruction transfers control to the instruction following the *call* instruction, but one can choose not to follow this convention. By doing so, it obscures the flow of execution in a program and is thus an attack on the control-flow generation phase. It can also, however, cause problems during the disassembly phase for certain methods of disassembly (recursive-traversal for instance). This obfuscation breaks down current context-sensitive interprocedural methods, which are based on these conventions. Four types of obfuscation related to call and return statements are identified by Lakhotia *et al.* (LAKHOTIA; KUMAR; VENABLE, 2005) below:

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* 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 *addr*, and the *ret* instruction causes execution to jump to *addr*. There are other ways to achieve the equivalent behavior. In Figure 5 an example of an obfuscated call is presented. In this example, the instruction in line L3 pushes the return address onto the stack, while the instruction in line L4 jumps to the function entry. Since no *call* statement is present, the programs call-graph will not be correct.

```
Main:
L1:   push  4
L2:   push  2
L3:   push  offset L5
L4:   jmp   Max
L5:   ret

Max:
L6:   mov   eax, [esp+4]
L7:   mov   ebx, [esp+8]
L8:   cmp   eax, ebx
L9:   jg    L11
L10:  mov   eax, ebx
L11:  ret   8
```

Figure 5: Example of obfuscation of a *call* instruction.

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 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. In Figure 6 an example of an obfuscated return is presented. In this example, the instruction in line L10 pops the return address off the stack, while the instruction in line L12 jumps to the address in the register *ebx*. Since no *ret* statement is present, the programs call-graph will not be correct. The instruction in line L11 fixes the stack pointer adding 8 bytes (4 bytes for each passed parameter).

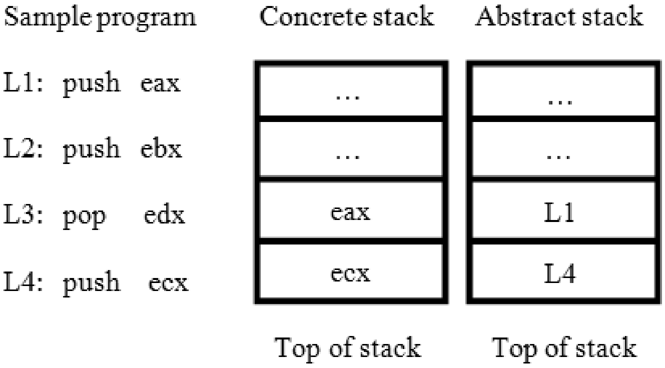
```
Main:
L1:   push  4
L2:   push  2
L3:   call  Max
L4:   ret

Max:
L5:   mov   eax, [esp+4]
L6:   mov   ebx, [esp+8]
L7:   cmp   eax, ebx
L8:   jg    L11
L9:   mov   eax, ebx
L10:  pop   ebx
L11:  add   esp, 8
L12:  jmp   ebx
```

Figure 6: Example of obfuscation of a *ret* instruction.

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.

2.5 Abstract Stack Graph

The concept of ASG 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. For example, considering Figure 7, each instruction in that program is marked with its address from $L1$ through $L4$. The contents of both concrete and abstract stacks, after execution of the instruction at $L4$, are shown in Figure 7. In particular, for the abstract stack, the addresses $L1$ and $L2$ are initially pushed onto the stack, then $L2$ is popped out by the *pop* instruction at $L3$, and $L4$ is pushed in afterwards.



```
L1:  push  eax
L2:  push  ebx
L3:  pop   edx
L4:  push  ecx
```

Figure 7: Concrete and abstract stacks.

```
E: //entry point
L0:   push   eax
L1:   sub    ecx, 1h
L2:   beqz   L8
L3:   push   ebx
L4:   push   ecx
L5:   dec    ecx
L6:   beqz   L3
L7:   jmp    L10
L8:   pop    ebx
L9:   push   esi
L10:  pop    edx
L11:  beq    L0
L12:  call   Abc
```

Figure 8: Sample program.

The example given in Figure 8 illustrates the construction of the abstract stacks. The control flow graph for that program appears in Figure 9. Each block in the control flow graph contains either a single *push*, *pop*, or *call* instruction or one of these instructions plus a control transfer instruction. Figure 10 shows some feasible abstract stacks at four program points. For example, the third abstract stack at program point 2 is the result of the following execution trace:

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

In another example, the abstract stack shown at program point 4 results from the trace:

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

While the execution trace

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

yields the abstract stack at program point 8.

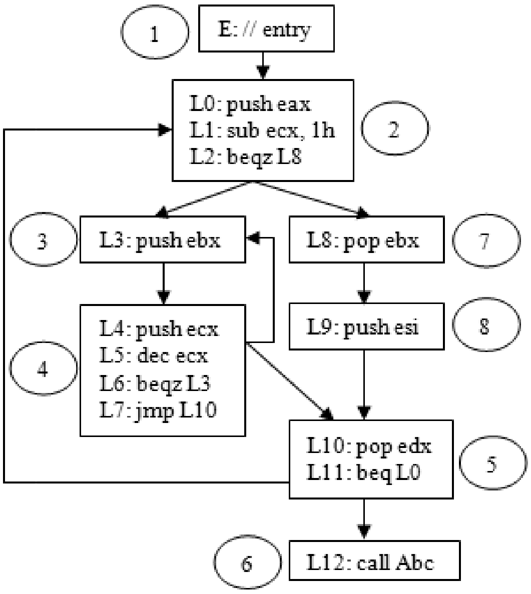


Figure 9: Control flow graph for sample program in Figure 8.

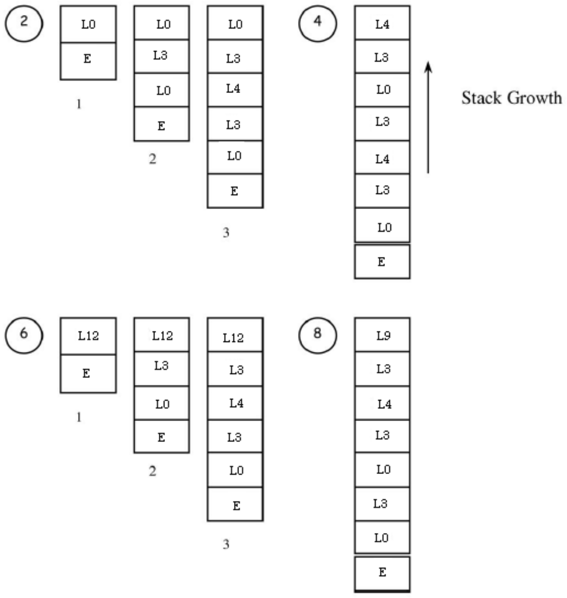


Figure 10: Possible abstract stacks at some program points.

Our interest is in finding all possible abstract stacks at each program point for all execution traces. Since multiple execution traces from the entry node to any program point may exist, there may be multiple abstract stacks at each program point. This is enumerated in the example by the multiple traces for program points 2 and 6 in Figure 10. In fact, program points 3 and 4 may have infinite number of abstract stacks because of loops. This is because there is a loop between program points 3 and 4, and the loop contains unbalanced *push*, *i.e.*, a *push* that is not matched with a *pop*.

An ASG is a concise representation of all, potentially infinite, abstract stacks at all points in the program. Figure 11 shows the abstract stack graph for the program in Figure 8. A path (sequence of nodes beginning from the abstract stack's top toward its bottom) in the graph represents a specific abstract stack. In that graph, rectangular boxes indicate nodes containing instruction addresses, while the edges represent potential traces that push values onto the stack.

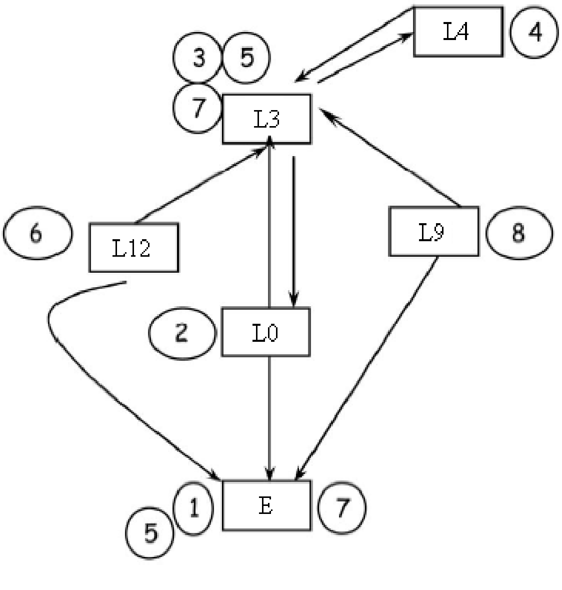


Figure 11: Abstract stack graph for sample program in Figure 8.

Formally, given a set of addresses represented by *Addr*, an abstract stack graph is a directed graph represented by the 3-tuple $< A, Ae, Aspr >$, where:

- $A \subseteq Addr$ is a set of nodes, and an address $a \in A$ implies the instruction at address a performs a push operation.
- $Ae \subseteq Addr \times Addr$ is a set of edges. An edge $< a, m > \in Ae$ denotes that there is possible execution trace in which the instruction at address a 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 $< x, a > \in Aspr$ means that program point x receives the abstract stack resulting from the value pushed by instruction a at the top.

The last relation is represented in the graph by annotating each node a with the address x in a circle, such that $< x, a > \in Ae$. This relation may be read as: a is the stack top at program point x or, alternatively, the stack top a is associated with the program point x.

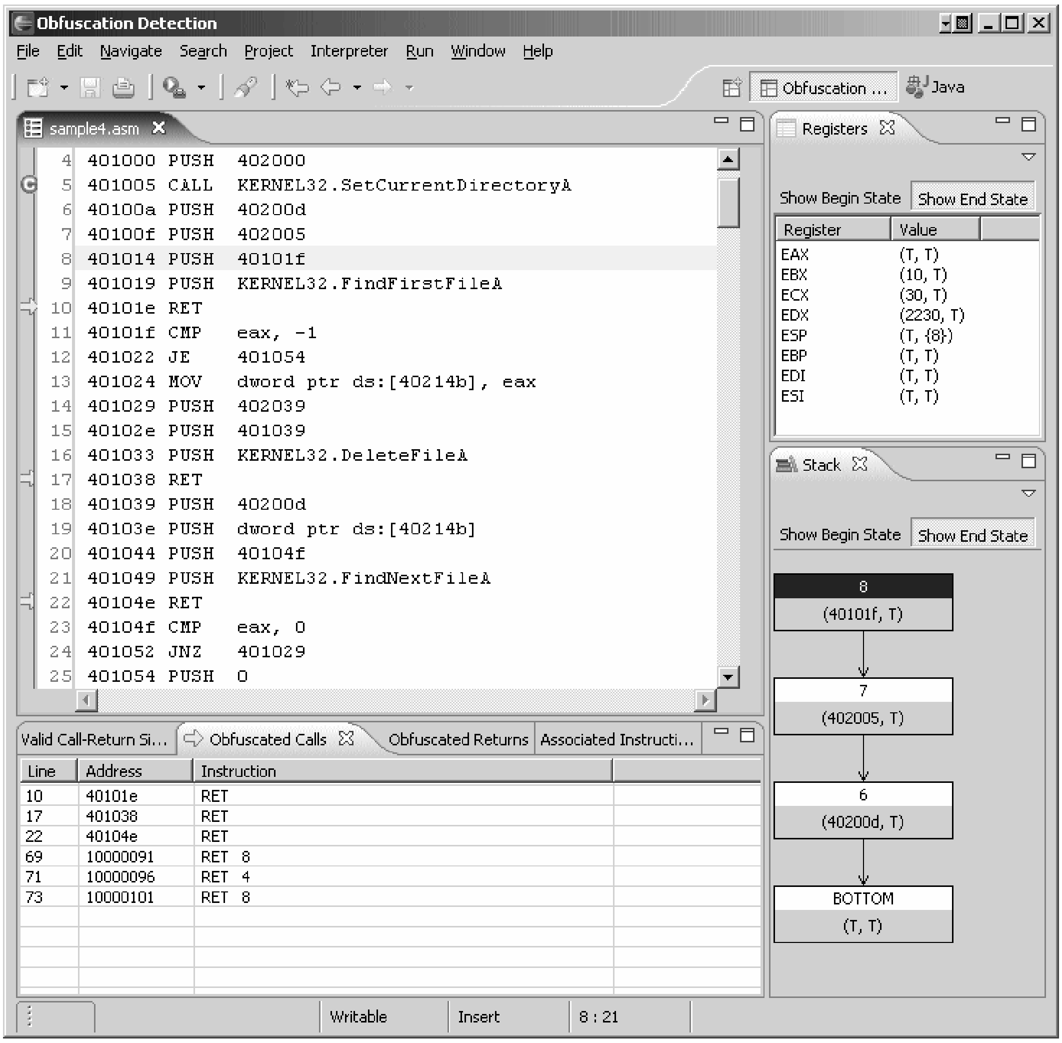
The analysis domain is defined by sets of instructions. A given domain **I** represents the abstract syntax domain for a given set of instructions. In such domain each instruction is annotated with its address in the program. Thus, $[m : call\ addr]$ is the abstract interpretation of the concrete instruction '*call addr*' at address m. In the same way, the domain ASG is the domain of abstract stack graphs. An element of ASG is a 3-tuple $< A, Ae, Asp >$, where A and Ae have the same meaning as in the definition of abstract stack graph. However, the set Asp is a projection of $Aspr$, where $Asp \subseteq Addr$ is the set of stack tops.

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 $L1|L2|L3|..|Lj$ such that $< Li \rightarrow Li + 1 > \in Ae$. A path p is mapped by a function Ψ to an abstract stack with the last-in element $L1$ and the first-in element Lj.

To be concise, in Figure 9 we use the number of each block in the CFG instead of the address of its instructions. 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 Figure 9, all points in a block have the same top of stack. For example, in Figure 11, $L3$ is an abstract node representing the address of the instruction *push ebx* and is associated with the set of program points $P = \{3, 5, 7\}$. All program points in P receive abstract stacks with top $L3$, *i.e.*, the abstract stack pointer $asp = L3$. Two possible abstract stacks when traversed from $asp = L3$ are $L3|L0|E$ and $L3|L4|L3|L0|E$.

2.6 DOC: Detector of Obfuscated Calls

The Detector of Obfuscated Calls (DOC), proposed by Venable *et al.* (VENABLE et al., 2005), (KUMAR; VENABLE, 2007), is a static analysis suite that detects obfuscations in executable, particularly procedure call and call-return obfuscations. It combines Lakhotia and Kumar's abstract stack graph (ASG) (LAKHOTIA; KUMAR, 2004) with Reps and Balakrishnan's value set analysis (VSA) (BALAKRISHNAN; REPS, 2004). It uses abstract interpretation to find instances in which explicit call or call-return instructions are not used. Figure 12 shows a screenshot of this tool.



```
 4 401000 PUSH   402000
 5 401005 CALL   KERNEL32.SetCurrentDirectoryA
 6 40100a PUSH   40200d
 7 40100f PUSH   402005
 8 401014 PUSH   40101f
 9 401019 PUSH   KERNEL32.FindFirstFileA
10 40101e RET
11 40101f CMP    eax, -1
12 401022 JE     401054
13 401024 MOV    dword ptr ds:[40214b], eax
14 401029 PUSH   402039
15 40102e PUSH   401039
16 401033 PUSH   KERNEL32.DeleteFileA
17 401038 RET
18 401039 PUSH   40200d
19 40103e PUSH   dword ptr ds:[40214b]
20 401044 PUSH   40104f
21 401049 PUSH   KERNEL32.FindNextFileA
22 40104e RET
23 40104f CMP    eax, 0
24 401052 JNZ    401029
25 401054 PUSH   0
```

Register	Value
EAX	(T, T)
EBX	(10, T)
ECX	(30, T)
EDX	(2230, T)
ESP	(T, {8})
EBP	(T, T)
EDI	(T, T)
ESI	(T, T)

8
(40101f, T)

7
(402005, T)

6
(40200d, T)

BOTTOM
(T, T)

Line	Address	Instruction
10	40101e	RET
17	401038	RET
22	40104e	RET
69	10000091	RET 8
71	10000096	RET 4
73	10000101	RET 8

Figure 12: Eclipse interface for DOC.

Embedded within every executable program are tell-tale signs of the program's intent located in a table of system calls that are needed by the program. This list of system calls can be used to determine a program's behavior prior to executing the program. Consider a program that makes use of the system calls. It is quite clear that data transfer over a network will take place if this program is executed. Malicious code writers, being wary of such transparency in their code, defend their programs' privacy by obfuscating the table of required system calls, thus obfuscating the program's true behavior. A common approach to this form of obfuscation is known as "call obfuscation."

In call obfuscation, the detection of used system calls is made difficult by replacing the existing *call* and *ret* instructions with a different, but semantically equivalent, set of instructions. DOC is capable of statically detecting such code obfuscation by interpreting the executable and building an abstract representation of the registers and stack, which allows us to detect pieces of code that violate standard calling conventions. The result is a more complete listing of system calls used by the program. Using this improved list of system calls, one is in a better position to accurately gauge a program's runtime behavior; a crucial first step in detecting malicious behavior.

DOC consists of four phases: disassembly, parsing, interpretation, and analysis. First, the binary must be disassembled into assembly instructions. Next, it is parsed by the program and translated into a suitable internal representation to be used for the next phase. Parsing is a relatively simple task requiring that only a grammar describing the input to be parsed be provided. This grammar can be swapped with other grammar descriptions, opening the possibility to handle other input formats such as disassembled code from other disassemblers. In the current implementation of DOC, the defined grammar is for the Ollydbg disassembler/debugger (OLLYDBG, 2009) and is thus susceptible to the same shortcomings as any recursive-traversal disassembler. DOC shows the feasibility of using abstract interpretation and abstract stack graph as a means to discover obfuscated function calls, a task that requires, but is separate from, disassembly.

The interpretation phase consists of computing abstract values where concrete values ordinarily reside (COUSOT; COUSOT, 1976). The abstract value may over-approximate the corresponding concrete value, but should never under-approximate it. After interpretation, it should be possible to observe the state at any given instruction and see what values some particular register or piece of memory may hold. The final phase is the analysis. The goal of the analysis is to find all call obfuscations hidden in the executable. The analysis phase has at its disposal the disassembled executable where each instruction is annotated with the state at that particular instruction. Using the ASG, the algorithm can uncover any obfuscations. Recall, the abstract stack graph is a data structure that represents the actual stack by storing abstract values, instead of actual ones. To assist in discovering obfuscations, each node in the ASG is modified to hold the address of the instruction that caused the node to be created. Constructing the graph in this way allows for easy detection of modifications to the return address located on the stack.

Two different types of obfuscations can be detected, obfuscations involving the replacement of a *call* instruction with some other combination of instructions, and obfuscations involving the replacement of the return address with a new address.

To detect obfuscations that involve simulating a *call* instruction with some combination of other instructions, the ASG at each *ret* instruction is examined. The instruction that created the node that is pointed to by register *esp* is retrieved. Note that this node is the top of the stack and should hold the return address at this point. Since the current instruction is a return instruction, the top of the stack should have been created by a corresponding *call* instruction (the *call* instruction pushes the return address onto the stack). If any other instruction is responsible for placing the return address onto the stack, then a call obfuscation has been detected. That is, some other tactic was used to simulate a call instead of using the *call* instruction directly.

The second type of obfuscation involves removing the return address from the stack and replacing it with some other value. This type of obfuscation is a useful attack against most disassemblers, since many of them typically assume that control will transfer back to the instruction that made the call, and thus will construct an incorrect control-flow graph.

To detect this type of obfuscation, we look for situations where the return address is either popped off the stack or some number is added to register *esp* which results in the return address being removed from the stack. At each *pop* instruction, if the instruction that created the node at the top of the stack (the node to be popped) is a *call* instruction, then it is evident that the return address is being removed from the stack. Thus, the *pop* instruction is flagged as an obfuscation. A similar approach is used for cases where the return address is removed by adding some number to *esp*.

One limitation of the current implementation involves efficiency. Each time an instruction is encountered and the state has changed, this instruction must be interpreted again using the new state. In a large program, there may easily be hundreds of different paths leading to an instruction, each path containing a new state. Thus, many instructions may have to be interpreted a large number of times, requiring a level of efficiency that is not met by the current implementation. Lack of context-sensitive interprocedural analysis complicates this situation by increasing the number of paths.

Another limitation involving efficiency is the lack of full memory support. The current implementation provides support for only the register and stack. Having full memory support will not only help improve the results of call obfuscation detection, but may also help make the project useful for other applications.

Another limitation worth noting is the lack of support for structured exception handling (SEH). SEH is a programming mechanism useful for detecting serious errors and transferring control to code designed to handle these errors. Malicious programmers sometimes use SEH as another way to control the flow of execution by intentionally causing an error to occur (such as a division by zero) and placing the malicious code at the location intended for code that will handle the error. The interpreter does not implement SEH and therefore is vulnerable to such attacks. This weakness can result in reachable code that is never processed by the interpreter.

3 Proposed algorithm

This chapter presents our proposed context-sensitive analysis of x86 obfuscated executables. In section 3.1, we illustrate that the Abstract Stack Graph (ASG) introduced by Lakhotia *et al.* (LAKHOTIA; KUMAR; VENABLE, 2005) can be used to adapt Sharir and Pnueli's (SHARIR; PNUELI, 1981) call-string approach in order to perform context-sensitive interprocedural analysis of programs with non-standard manipulation of stack, including obfuscation of calls. In section 3.2, we introduce a trace semantics in which context is made explicit. In section 3.3, generalizations of Sharir and Pnueli's (SHARIR; PNUELI, 1981) k-suffix method for abstracting calling-contexts and Emami *et al.*'s method of abstracting calling-contexts by reducing recursive cycles (EMAMI; GHIYA; HENDREN, 1994) are presented. In section 3.4, we present our algorithm for analysis of binaries. It adapts the concept of context-trace to binaries using stack-contexts and also summarizes the use of Balakrishnan and Reps' Value-Set Analysis (BALAKRISHNAN; REPS, 2004),(BALAKRISHNAN, 2007) to determine transfer of control in assembly programs. Finally, section 3.5 contains examples illustrating the context-sensitive analysis process.

3.1 Motivation and Intuition

Context-sensitivity is presented classically in the literature in terms of paths of an Interprocedural Control Flow Graph (ICFG), a graph that encodes the transfer of control component of semantics of instructions. An ICFG consists of CFGs for individual procedures. Edges between these CFGs represent interprocedural control flow, typically expressed by *call* edges and *return* edges. A path, starting from the entry node, in an individual CFG represents a valid sequence of flow of control. A flow-sensitive analysis propagates data over paths of a CFG. However, a path that starts from the entry of the program and traverses nodes in multiple CFGs may not always represent a valid flow of control. For example, propagating information to all the successors at a return instruction 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 create spurious pathways for information flow. For such a path to be valid, the *call* and *ret* edges in the path should be paired and should meet certain constraints. In the following, we describe the most general and simplest method of performing context sensitive interprocedural data flow analysis.

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 then propagated as the tagged information is used to compute other information. Finally, at the return edge, information is propagated back only to 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 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 (CG) 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 $(c_1c_2...c_n)$ such that call site c_1 belongs to the entry procedure, and there exists a path in the call-graph consisting of edges $c_1, c_2, ..., c_n$. 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 $(*c_1c_2...c_k)$, where the parameter k is the bound of the call string size and represents the set $\{cs_k \mid cs_k \in CS_k,\ cs = \pi c_1c_2...c_k$ and $|\pi| \geq 1\}$.

Since the prior definition of context-sensitivity is tied to semantics of procedure call and return statements of high-level languages, and therefore, *call* and *ret* instructions of assembly language, it is not directly applicable for context-sensitive analysis of binaries that are obfuscated. In the following, we show how an abstract stack graph (ASG) may be used in place of a call-graph (CG) for context-sensitive analysis of binaries that are obfuscated.

Lemma 3.1.1 *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.*

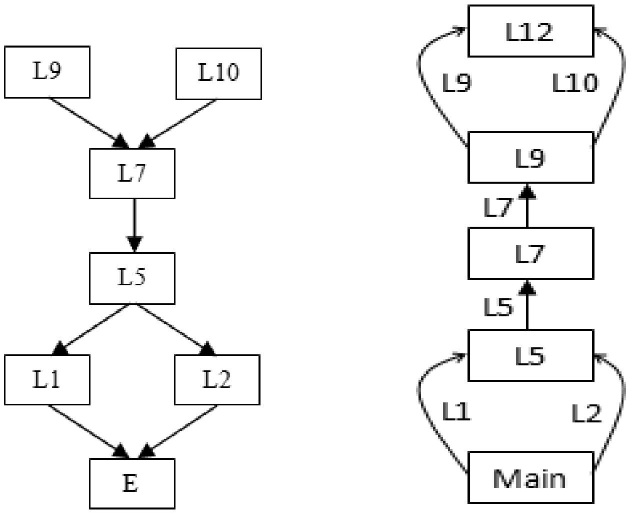
Proof The nodes of the ASG for such a program will consist of only the call sites. An edge in the ASG from a call-site L_j to a call-site L_i exists iff there is an execution path from L_i to L_j with no other *call* instruction along the path. Assume that L_i is a statement in procedure P_i, and L_j 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 context, a finite length path in an ASG. Formally, a stack context can be defined as a path in the ASG of program locations $(l_1 l_2 ... l_n)$ 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 $l_1 l_2 ... l_n$ such that l_n is the top of the stack. Analogous to Sharir and Pnueli's saturated call-string we can define a saturated stack context as a string whose encoded history of the program locations exceeds some limit k. It is represented as $(*l_1 l_2 ... l_k)$, where the parameter k is the bound of the stack-string size and represents the set $\{ss_k \mid ss_k \in SS_k,\ ss = \pi l_1 l_2 ... l_k \text{ and } |\pi| \geq 1\}$.

Figure 13 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 3.1.2 *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).*



(a) Abstract stack graph.

(b) Call-graph.

Figure 13: Abstract stack graph and call-graph for code of Figure 1(a).

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

The above discussion implies that the ASG can be used as a substitute for programs that do not obfuscate *call* and *ret* instructions. When performing interprocedural analysis, 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 14 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 13(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 14 and 13(a) suggests that paths in the ASG may be treated as a replacement for call-string, even for obfuscated programs. Instead

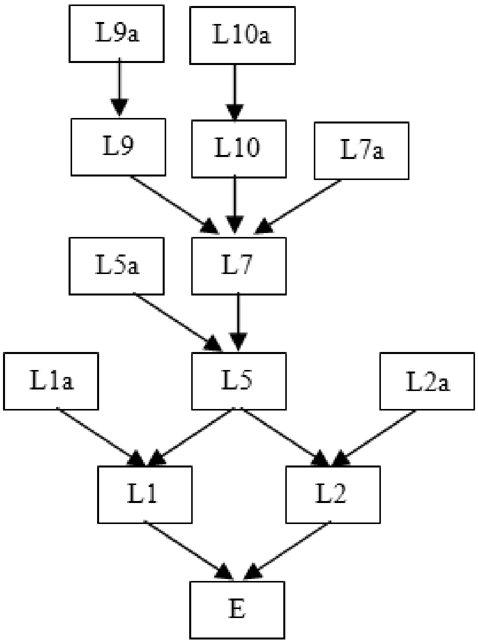


Figure 14: Abstract stack graph for the obfuscated code of Figure 1(c)

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; KUMAR; VENABLE, 2005),(VENABLE et al., 2005). When the computation of ASG may require performing other analysis, as is likely in obfuscated programs, the two analyses 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 a CG. Even for non-obfuscated code an ASG may have more nodes than 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 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 a larger k.

In the following section, we show using abstract interpretation framework how to derive contexts using trace semantics and how to adapt for use with stack context prior work on performing context-sensitive analysis using calling-contexts.

Before preceding to this chapter, the following notations and operators involving strings are introduced. Let X^* denote the Kleene closure of the set X, *i.e.*, the set of finite sequences over X. We use the symbol $\epsilon \in X^*$ to denote the sequence of length 0 and $(x\ i)$ to represent the i^{th} element of the sequence $x \in X^*$. The symbol '.': $X \times X^* \rightarrow X^*$ is the *cons operator*, which inserts an element at the head of a sequence. It is defined formally as: $a.x = y \Leftrightarrow (y\ 0) = a \wedge \forall i \geq 0 \ \ : \ (y\ i+1) = (x\ i)$. If $\langle X, \sqsubseteq_X\rangle$ is a lattice, then $\langle X^*, \sqsubseteq_{X^*}\rangle$ is a lattice where $\sqsubseteq_{X^*}$ is defined as follows:

$$\begin{aligned}
&\forall x_1, x_2 \in X; s, s_1, s_2 \in X^* \\
&\epsilon \sqsubseteq_{X^*} s \\
&x_1.s_1 \sqsubseteq_{X^*} x_2.s_2 \Leftrightarrow x_1 \sqsubseteq_X x_2 \wedge s_1 \sqsubseteq_{X^*} s_2
\end{aligned}$$

The order resulting from $\sqsubseteq_{X^*}$ is called *strong ordering*, for it defines a sequence to be smaller than another sequence iff all of its elements are smaller than the corresponding elements of the other sequence. We introduce some operators on sequences for syntactic convenience. We assume two polymorphic extensions of the cons operator ".". One to insert an element at the end of a sequence: $X^* \times X \rightarrow X^*$, and the other to concatenate two sequences: $X^* \times X^* \rightarrow X^*$. We also define the function *rest* operating on X^* as follows: $(rest\ a.x) = x$. When convenient, we also use the notation "$Y{\downarrow}X$" to denote the Xth element of the pair Y.

3.2 Context-trace Semantics

In this section we use the machinery of abstract interpretation to develop a generalized notion of context-sensitive analysis, where contexts are maintained in LIFO order. The concept is general in that it only requires the knowledge of the set of instructions that create contexts and those that delete contexts. This generalized concept of context-sensitivity does not depend on whether an instruction transfers control. The primary constraint required is that the most recently created context be destroyed first.

Let $\llparenthesis \subseteq I$ denote the set of instructions (of a language) that open contexts, and $\rrparenthesis \subseteq I$ denote the set of instructions that close contexts. A *context string* is the sequence of context opening instructions belonging to the $\llparenthesis^* \subseteq I^*$. The function π represents the effect of an individual state, an element of Σ, on the accumulated context string.

$$\begin{aligned}
&\pi : \Sigma \rightarrow \llparenthesis^{*} \rightarrow \llparenthesis^{*} \\
&\quad \pi\ s\ \nu\ \triangleq \begin{cases} i.\nu & \text{if } i \in \llparenthesis \\ (rest\ \nu) & \text{if } i \in \rrparenthesis \end{cases} \\
&\quad \text{where } i = s \downarrow 1.
\end{aligned}$$

If the instruction in the state given by $s \downarrow 1$ belongs to the set $\llparenthesis$, it is pushed on the current context string. If the instruction belongs to $\rrparenthesis$, it pops the topmost context from the context string. Otherwise, the context string is left unchanged.

Now given a trace σ we can map it to its current context $\nu = \Pi\ \sigma$, where Π is defined as follows:

$$\begin{aligned}
&\Pi : \Sigma^{*} \rightarrow \llparenthesis^{*} \\
&\quad \Pi\ \sigma\ \triangleq (\Pi'\ \sigma\ \epsilon) \\
&\Pi' : \Sigma^{*} \rightarrow \llparenthesis^{*} \rightarrow \llparenthesis^{*} \\
&\quad \Pi'\ \epsilon\ \nu\ \triangleq \nu \\
&\quad \Pi'\ s.\sigma\ \nu\ \triangleq (\Pi'\ \sigma\ (\pi\ s\ \nu))
\end{aligned}$$

The function Π maps a trace to its context string —the list of contexts that are open— by applying π repeatedly on successive elements of σ. Let ν_i represent the context string from the i^{th} application of π. The function Π (using Π') establishes the following relation $\nu_i = (\pi\ (\sigma\ i)\ \nu_{i-1})$, where $\nu_0 = \epsilon$, for $1 \leq i \leq |\sigma|$.

Let us assume, for example, context strings created by procedure calls, in which the opening and closing contexts are given by $c_i \in \llparenthesis$ and $r_i \in \rrparenthesis$, respectively. Figure 15 shows an example of an Interprocedural Control Flow Graph (ICFG) taken from Sharir and Pnueli (SHARIR; PNUELI, 1981) with the addition of call site c_5. An ICFG contains nodes used in an intraprocedural CFG and two extra nodes for every call site: a call node (c_i) and a return node (r_i). An ICFG for a program can be obtained by connecting the CFGs for individual procedures and adding edges among call, return, start (s_i) and end nodes (e_i) as follows:

- For every call site calling some procedure X, an edge is added from the corresponding call node to the start node of procedure X.
- For every procedure X, an edge is added from the end node of X to the return nodes associated with every call to procedure X.

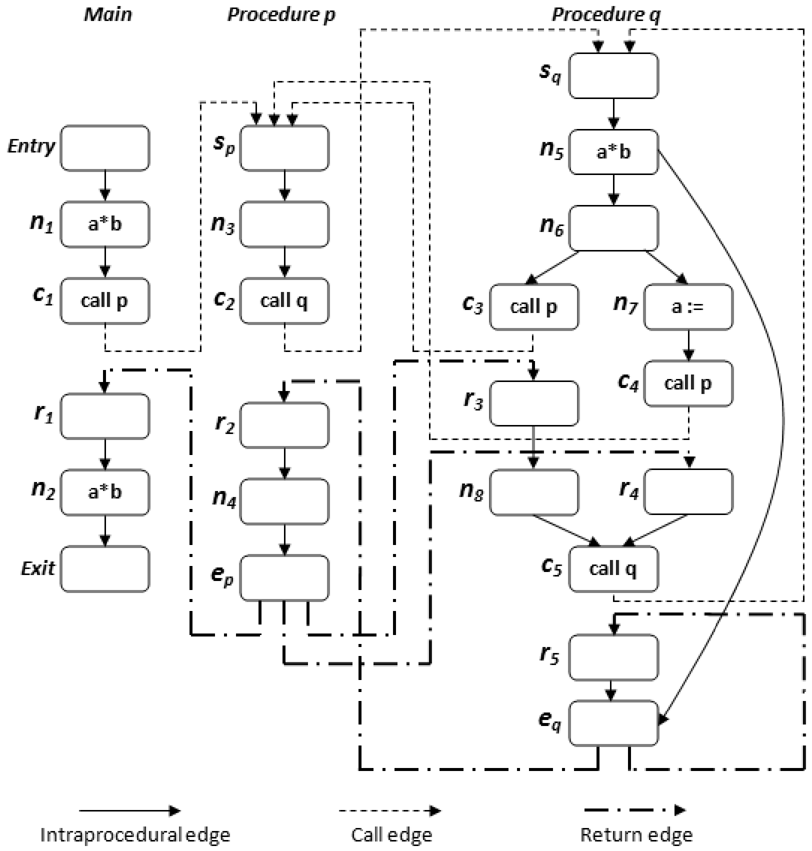


Figure 15: Example to demonstrate context string derivation.

Consider, for example, the sequence of instructions from the node n_1 to the node n_4 resulting from projecting out only the instructions from a trace:

$$n_1\ c_1\ s_p\ n_3\ c_2\ s_q\ n_5\ n_6\ c_3\ s_p\ n_3\ c_2\ s_q\ n_5\ e_q\ r_2\ n_4$$

Extracting open and close contexts in this trace, we obtain the sequence $c_1\ c_2\ c_3\ c_2\ r_2$. The context string associated with each prefix of this sequence is given as follows: $\{(c_1) \mapsto (c_1)$, $(c_1\ c_2) \mapsto (c_2\ c_1)$, $(c_1\ c_2\ c_3) \mapsto (c_3\ c_2\ c_1)$, $(c_1\ c_2\ c_3\ c_2) \mapsto (c_2\ c_3\ c_2\ c_1)$, $(c_1\ c_2\ c_3\ c_2\ r_2)$ $\mapsto (c_3\ c_2\ c_1)\}$. Table 3 shows other examples of extracted sequences and their respective context strings.

A *context-trace* is a pair of a context string and a trace $(\nu, \sigma) \in (\llparenthesis^* \times \Sigma^*)$. Not all elements of the set $(\llparenthesis^* \times \Sigma^*)$ are meaningful. We define a context-trace in which the context string represents the context associated with the trace as a Π-*valid* context-trace.

Definition 3.2.1 *A context-trace* $(\nu, \sigma) \in (\llparenthesis^* \times \Sigma^*)$ *is* Π-*valid iff* $\nu = \Pi\ \sigma$.

A Π-valid context-trace is equivalent to a valid-interprocedural path in the ICFG of a program when the sets $\llparenthesis$ and $\rrparenthesis$ represent the set of call and return instructions, respectively,

Table 3: Examples of sequences of open and close contexts for the program of Figure 15 and their respective context strings.

Trace	**Context**
c_1c_2	c_2c_1
$c_1c_2c_3c_2$	$c_2c_3c_2c_1$
$c_1c_2c_4c_2$	$c_2c_4c_2c_1$
$c_1c_2c_3c_2c_4c_2$	$c_2c_4c_2c_3c_2c_1$
$c_1c_2c_4c_2c_3c_2$	$c_2c_3c_2c_4c_2c_1$
$c_1c_2c_4c_2c_3c_2r_2$	$c_3c_2c_4c_2c_1$
$c_1c_2c_4c_2c_3c_2r_2r_3$	$c_2c_4c_2c_1$
$c_1c_2c_4c_2c_3c_2r_2r_3c_5$	$c_5c_2c_4c_2c_1$
$c_1c_2c_4c_2c_3c_2r_2r_3c_5c_3c_2r_2$	$c_3c_5c_2c_4c_2c_1$
$c_1c_2c_4c_2c_3c_2r_2r_3c_5c_4c_2r_2r_4c_5$	$c_5c_5c_2c_4c_2c_1$
$c_1c_2c_4c_2r_2r_4c_5r_5$	c_2c_1
$c_1c_2c_4c_2r_2r_4c_5r_5r_2r_1$	ϵ

of that program.

We denote the set of all finite partial Π-valid context-traces as $\wp((\!|^* \times \Sigma^*)_\Pi \equiv (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*)$. This forms the semantic domain for the context-trace semantics. The following lemma shows that this semantic domain is equivalent to $\wp(\Sigma^*)$, the semantic domain for the trace semantics.

Lemma 3.2.2 $(\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) \equiv \wp(\Sigma^*)$

Proof Let $f : (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) \longrightarrow \wp(\Sigma^*)$ and $g : \wp(\Sigma^*) \longrightarrow (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*)$ be functions defined as follows:

$$f : (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) \longrightarrow \wp(\Sigma^*)$$

$$f\ Y = \bigcup_{\nu\ \in\ dom\ Y} Y\ \nu$$

$$g : \wp(\Sigma^*) \longrightarrow (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*)$$

$$g\ X = \lambda\nu\,\text{.}\,\{\sigma \mid \sigma \in X, \nu = \Pi\ \sigma\}$$

where $Y : (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*)$ and $X \in \wp(\Sigma^*)$. We can say that $(\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) \equiv \wp(\Sigma^*)$ if $\forall X \in \wp(\Sigma^*) : f(g\ X) = X$ and $\forall Y \in (\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) : g(f\ Y) = Y$.

i) $\forall X \in \wp(\Sigma^*) : f(g\ X) = X$.

We have to show that $f(g\ X) \subset X$ and $X \subset f(g\ X)$. The proof is as follows.

$$f(g\ X) = \bigcup_{\nu\ \in\ dom(g\ X)} (g\ X)\nu = \bigcup_{\nu\ \in\ dom(g\ X)} \{\sigma \mid \sigma \in X, \nu = \Pi\ \sigma\} \subset X$$

It is clear that $X \subset f(g\ X)$ since that $\sigma \in X$ in the union of $f(g\ X)$.

ii) $\forall Y \in \llparenthesis^{*} \xrightarrow{\Pi} \wp(\Sigma^{*}) : g(f\ Y) = Y$.

We have to show that $g(f\ Y) \subset Y$ and that $Y \subset g(f\ Y)$. The proof is as follows.

$$g(f\ Y) = \{\sigma \mid \sigma \in (f\ Y), \nu = \Pi\ \sigma\} = \{\sigma \mid \sigma \in \bigcup_{\nu\ \in\ dom\ Y} (Y\ \nu), \nu = \Pi\ \sigma\} \subset (Y\ \nu)$$

Now, for $Y \subset g(f\ Y)$, we assume $\exists \nu \in Y$ such that $\nu \notin \{\sigma \mid \sigma \in \bigcup_{\nu\ \in\ dom\ Y}(Y\ \nu), \nu = \Pi\ \sigma\}$, *i.e.*, $\nu \neq \Pi\ \sigma$ which contradicts g definition. ▮

Since the domains of context-trace semantics and trace semantics are equivalent, it follows that the trace semantics can be mapped to its equivalent context-trace semantics. This then gives us the framework needed to develop context-sensitive analyses, where context is made explicit. Most importantly it gives the framework to derive a context-sensitive counterpart of context insensitive analysis.

Assume that an analysis $I \longrightarrow Abstore$ derived from the trace semantics $\wp((I \times Store))^{*}$ is context-insensitive. Its context sensitive counterpart may be derived using the following chain of Galois connections:

$$\begin{aligned}
\llparenthesis^{*} \xrightarrow{\Pi} \wp((I \times Store)^{*}) &\sqsubseteq \llparenthesis^{*} \xrightarrow{\Pi} (\wp(I \times Store))^{*} \\
&\equiv \llparenthesis^{*} \xrightarrow{\Pi} (I \longrightarrow \wp(Store))^{*} \\
&\sqsubseteq \llparenthesis^{Abs} \xrightarrow{\Pi} I \longrightarrow Abstore
\end{aligned}$$

where $\llparenthesis^{Abs}$ is an abstraction of the concrete context $\llparenthesis^{*}$. In the following section we describe two context abstractions generalized from analogous abstractions used for calling-contexts.

3.3 Context Abstractions

Due to recursion and mutual recursion, the set of all finite length calling-contexts in a program may be infinite. Even when a program does not have recursion, the number of calling-contexts it has can be exponentially large (LHOTÁK; HENDREN, 2006). So while a full call-string analysis may yield the most precise results, it may not be practical to compute it. To make the analysis scaleable for large programs, it is common to reduce the space of calling-contexts by using certain abstractions.

The literature contains two significant classes of abstractions for calling-contexts. The first one, introduced by Sharir and Pnueli (SHARIR; PNUELI, 1981), abstracts a call string by mapping it to its k-length suffix. The second abstraction, introduced by Emami *et al.* (EMAMI; GHIYA; HENDREN, 1994), effectively abstracts a call string by reducing recursive paths in it by a single node. We say 'effectively' because the method is not stated as an abstraction over call-strings but can be mapped to such an abstraction. There are a few later works whose calling-context abstractions may also be mapped to this second abstraction (WILSON; LAM, 1995),(WHALEY; LAM, 2004),(ZHU; CALMAN, 2004).

What is true of calling-contexts will also be true for any other instantiation of our generalized notion of context. Hence, it is beneficial to develop generalized context abstractions for use in any context-sensitive analysis. Since the abstractions for calling contexts have been defined in terms of paths over ICFG, the original definitions cannot be directly mapped to generalized contexts that are defined independently of control flow.

In the following subsections, we derive the two abstractions using the machinery of abstract interpretation. We call the generalization of Sharir and Pnueli's k-suffix approach as k-context abstraction and Emami *et al.*'s reduction of recursive loops as ℓ-context abstraction. While Sharir and Pnueli used k length *suffixes*, our abstraction uses k length *prefixes* because in our stack the most recent element is inserted at the head of the sequence. Mapping from our method to Emami *et al.*'s is not that straightforward. Emami *et al.* define a context as a node in an "invocation graph." Our ℓ-context strings correspond to paths in Emami *et al.*'s invocation graph.

It is apparent that there is not a significant algorithmic challenge in generalizing the abstractions from calling-contexts to generalized contexts. However, the real issue in developing the abstraction is in how one would prove that an analysis using that abstraction will be sound. When used for abstracting calling-context, such arguments are made by reasoning over paths of an ICFG. Since the generalized context does not have the benefit of an ICFG, albeit by design, the arguments about soundness must be developed.

Thus, the most significant component of the generalization we perform is the derivation of Galois connections, and for these are necessary to prove the soundness of any analysis derived from these context abstractions.

3.3.1 k-Context

Let $\llparenthesis^k$ represent the set of sequences of opening contexts of length $\leq k$ and $k+1$ length sequences created by appending $\top = \bigsqcup \llparenthesis$ to k-length sequences of opening contexts. An element of $\llparenthesis^k$ is called a *k-context*. We can establish a map $\alpha_k : \llparenthesis^* \rightarrow \llparenthesis^k$ as:

$$\alpha_k \ \nu \triangleq \begin{cases} \nu & \text{if } |\nu| \leq k \\ \nu_k.\top & \text{otherwise, where } \nu = \nu_k.\nu' \text{ for some } \nu' \end{cases}$$

In other words, when ν is longer than k, α_k maps it to $\nu_k.\top$, where ν_k is the k-length prefix of ν. A sequence of length $\leq k$ is mapped to itself. It is shown in the following that $\llparenthesis^k$ is an abstraction of $\llparenthesis^*$.

Lemma 3.3.1 *α_k is surjective and additive.*

Proof The surjectivity property follows from the application of α_k, since $\forall \nu_k \in \llparenthesis^k$ is formed from a k-length prefix of a $\nu \in \llparenthesis^*$. The additivity property is true if: $f(a \sqcup b) = f(a) \sqcup f(b)$, where $f(x)$ is the application of α_k over a context string ν, and

i) $a \sqcup b$ results from the consecutive application of the strong ordering operator over every element in the context strings a and b.

ii) $f(a \sqcup b)$ produces a k-context string with the prefix k elements from $a \sqcup b$, preserving their order.

iii) $f(a)$ and $f(b)$ produces two k-context strings with the prefix k elements from a and b, respectively, preserving their order.

iv) $f(a) \sqcup f(b)$ produces a context string resulting from the consecutive application of the strong ordering operator over every element in the k-contexts $f(a)$ and $f(b)$.

Since strong ordering operator applied over two elements of a given context string results either in $\top$ if the elements are different or the actual element if it is the same in both context strings, and α_k preserves the order of the context, from (ii) and (iii), it can be proved that α_k is additive. ∎

Thus, $\llparenthesis^*$ and $\llparenthesis^k$ form a Galois insertion with the abstraction map α_k. Context-sensitive analyses may be derived by defining appropriate context abstraction $\llparenthesis^k \sqsubseteq \llparenthesis^{Abs}$.

In Table 4, the “Context” column provides some examples of contexts. Their corresponding k-context abstractions, with $k = 2$, are shown in the “2-Context” column.

Table 4: Examples of contexts and abstract contexts.

Context	**2-Context**	**ℓ-Context**
c_2c_1	c_2c_1	c_2c_1
$c_2c_3c_2c_1$	$c_2c_3\top$	$c_2^+c_1$
$c_2c_4c_2c_1$	$c_2c_4\top$	$c_2^+c_1$
$c_2c_4c_2c_3c_2c_1$	$c_2c_4\top$	$c_2^+c_1$
$c_2c_3c_2c_4c_2c_1$	$c_2c_3\top$	$c_2^+c_1$
$c_3c_2c_4c_2c_1$	$c_3c_2\top$	$c_3c_2^+c_1$
$c_2c_4c_2c_1$	$c_2c_4\top$	$c_2^+c_1$
$c_5c_2c_4c_2c_1$	$c_5c_2\top$	$c_5c_2^+c_1$
$c_3c_5c_2c_4c_2c_1$	$c_3c_5\top$	$c_3c_5c_2^+c_1$
$c_5c_5c_2c_4c_2c_1$	$c_5c_5\top$	$c_5^+c_2^+c_1$
c_2c_1	c_2c_1	c_2c_1
ϵ	ϵ	ϵ

3.3.2 ℓ-Context

Let $\mathbb{B}$ represent the set $\{1, +\}$, where $1 \sqsubseteq +$. The set $\llparenthesis^\ell \subseteq (\llparenthesis \times \mathbb{B})^*$ is defined as:

Definition 3.3.2 *$\llparenthesis^\ell$ is the smallest set contained in $(\llparenthesis \times \mathbb{B})^*$ satisfying:*

1. $\epsilon \in \llparenthesis^\ell$

2. $\forall \nu_\ell \in \llparenthesis^\ell \; ; c \in \llparenthesis$
$\forall x \in \mathbb{B} \;:\; (c, x) \notin \nu_\ell \Rightarrow (c, 1).\nu_\ell \in \llparenthesis^\ell \wedge (c, +).\nu_\ell \in \llparenthesis^\ell$

Assume $\llparenthesis = \{a, b, c\}$, the notation x denotes $(x, 1)$, and x^+ denotes $(x, +)$. The following strings are some examples of sequences in $\llparenthesis^\ell$: ϵ, a, ab, a^+, ab^+, $a^+b^+c^+$. Some examples of sequences in $(\llparenthesis \times \mathbb{B})^*$, but not in $\llparenthesis^\ell$, are: aa, $abba$, a^+a^+, aba^+b^+.

The following lemma gives the bound on the size of strings in $\llparenthesis^\ell$.

Lemma 3.3.3 $\forall \nu_\ell \in \llparenthesis^\ell \;:\; |\nu_\ell| \leq |\llparenthesis|$.

Proof For any element $c \in \llparenthesis^\ell$, either c or c^+ may be in ν_ℓ, and each element can occur only once. ∎

The element a^+ represents the set of all contexts that start at the opening context a followed by a sequence of contexts and then terminating on the opening context a. Table 4 provides examples of contexts and their corresponding ℓ-contexts. Consider the context “$c_3c_2c_4c_2c_1$,” which when read right-to-left gives the order in which the contexts were pushed. It is abstracted to $c_3c_2^+c_1$. The term c_2^+ represents the set of all non-zero length sequences starting with c_2 and ending with c_2, and thus represents all cyclic context strings from c_2 to c_2. The term $c_3c_2^+c_1$ thus represents the set of contexts consisting of the opening context c_3, pushed on a sequence of openings contexts starting with c_2 and ending with c_2 and pushed on the opening context c_1.

To develop the abstraction function from $\llparenthesis^*$ to $\llparenthesis^\ell$ we first develop the abstract syntax tree (AST) domain $\llparenthesis_T$ that is isomorphic to $\llparenthesis^*$. The abstraction map is then defined on $\llparenthesis_T$. The following rule defines the syntactic structure of $\llparenthesis_T$ in terms of $\llparenthesis_T^*$.

$$\llparenthesis_T = \perp \cup \llparenthesis_T \times \llparenthesis_T^* \times \llparenthesis$$

An element of $\llparenthesis_T$ may either be $\perp$ or a 3-tuple consisting of (ν_T, σ_T, c) where $\nu_T \in \llparenthesis_T$, $\sigma_T \in \llparenthesis_T^*$, and $c \in \llparenthesis$. In addition, we also require that the elements of $\llparenthesis_T$ further satisfy the semantic constraint that (t, σ_T, c) is in $\llparenthesis_T$ iff c does not occur again in the subtrees t and σ_T, which is formally defined as follows:

$$\forall t \in \llparenthesis_T; \sigma_T \in \llparenthesis_T^*; c \in \llparenthesis \;:\; (t, \sigma_T, c) \in \llparenthesis_T \Leftrightarrow c \notin_T t \wedge c \notin_{T^*} \sigma_T$$

where the two relations $\in_T \subseteq \llparenthesis \times \llparenthesis_T$ and $\in_{T^*} \subseteq \llparenthesis \times \llparenthesis_T^*$ are defined as follows:

$$\begin{aligned}
&\forall c, d \in \llparenthesis; \sigma_T \in \llparenthesis_T^*; t \in \llparenthesis_T\\
&c \;\in_T\; (t, \sigma_T, d) \Leftrightarrow c \in_T t \vee c \in_{T^*} \sigma_T \vee d = c\\
&c \;\in_{T^*}\; t.\sigma_T \Leftrightarrow c \in_T t \vee c \in_{T^*} \sigma_T
\end{aligned}$$

The function ϕ maps elements from $\llparenthesis^*$ to $\llparenthesis_T$. This map amounts to parsing.

$$\begin{aligned}
&\phi : \llparenthesis^* \rightarrow \llparenthesis_T\\
&\quad \phi\ \epsilon \;\triangleq\; \perp\\
&\quad \phi\ \sigma.c \;\triangleq\; ((\phi\ s_1), (map\ \phi\ [s_2, s_3, \ldots, s_n]), c)
\end{aligned} \tag{3.1}$$

where $\sigma = s_1.c.s_2.c.\ldots.c.s_n$ for some $s_1, s_2, \ldots, s_n \in \llparenthesis^*$ such that $\forall\ 1 \leq i \leq n \;:\; c \notin s_i$. The function splits a context string, using its first context c, into a sequence of maximal substrings $s_1, \ldots, s_n$ such that each of the s_i does not contain c. The triple (s_1, $[s_2 \ldots s_n]$, c) is used to create the recursive structure, with the function *map* lifting ϕ to apply it point-wise on all elements of a sequence. This construction ensures that the semantic constraint for $\llparenthesis_T$ is preserved. The map from a sequence $\sigma.c \in \llparenthesis^*$ to the triple (s_1, $[s_2 \ldots s_n]$, c) is bijective. Thus, the domains $\llparenthesis^*$ and $\llparenthesis_T$ are isomorphic. Table 5 provides

Table 5: Examples of mapping contexts and T-contexts.

Context	**T-Context**
c_2c_1	$((\bot,\epsilon,c_2),\epsilon,c_1)$
$c_2c_3c_2c_1$	$((\bot,[(\bot,\epsilon,c_3)],c_2),\epsilon,c_1)$
$c_2c_4c_2c_1$	$((\bot,[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_2c_4c_2c_3c_2c_1$	$((\bot,[(\bot,\epsilon,c_4),(\bot,\epsilon,c_3)],c_2),\epsilon,c_1)$
$c_2c_3c_2c_4c_2c_1$	$((\bot,[(\bot,\epsilon,c_3),(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_3c_2c_4c_2c_1$	$(((\bot,\epsilon,c_3),[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_2c_4c_2c_1$	$((\bot,[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_5c_2c_4c_2c_1$	$(((\bot,\epsilon,c_5),[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_3c_5c_2c_4c_2c_1$	$((((\bot,\epsilon,c_3),\epsilon,c_5),[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
$c_5c_5c_2c_4c_2c_1$	$(((\bot,[\bot],c_5),[(\bot,\epsilon,c_4)],c_2),\epsilon,c_1)$
c_2c_1	$((\bot,\epsilon,c_2),\epsilon,c_1)$
ϵ	$\bot$

examples of contexts and their corresponding "T-contexts", i.e., the corresponding terms in $\mathbb{C}_T$.

We now define an abstraction map $\alpha_\ell : \mathbb{C}_T \to \mathbb{C}^\ell$ as follows:

$$\begin{aligned} &\forall t \in \mathbb{C}_T; s \in \mathbb{C}_T^*; c \in \mathbb{C} \\ &(\alpha_\ell \ \bot) \triangleq \epsilon \\ &(\alpha_\ell \ (t,s,c)) \triangleq (\alpha_\ell \ t).c^{(\alpha_\mathbb{B} \ |s|)} \end{aligned} \tag{3.2}$$

where $\alpha_\mathbb{B}$ is defined as:

$$(\alpha_\mathbb{B} \ n) \triangleq \begin{cases} 1 & n = 0 \\ + & n > 0 \end{cases}$$

It follows from the definition that $\alpha_\mathbb{B}$ is surjective and additive. Hence, $\alpha_\mathbb{B}$ is an abstraction from $\mathbb{N}$ to $\mathbb{B}$, and $\mathbb{N}$ and $\mathbb{B}$ form a Galois insertion. To demonstrate that α_ℓ is additive we introduce the relation $\sqsubseteq_T$ on $\mathbb{C}_T$ as follows:

$$\begin{aligned} &\forall t, t_1, t_2 \in \mathbb{C}_T; \forall s_1, s_2 \in \mathbb{C}_T^*; \forall c_1, c_2 \in \mathbb{C} \\ &\bot \sqsubseteq_T t, \\ &(t_1, s_1, c_1) \sqsubseteq_T (t_2, s_2, c_2) \Leftrightarrow t_1 \sqsubseteq_T t_2 \wedge s_1 \sqsubseteq_T^* s_2 \wedge c_1 \sqsubseteq c_2 \end{aligned}$$

It can be shown that $\sqsubseteq_T$ is reflexive, anti-symmetric, and transitive, thus defining a partial order on $\mathbb{C}_T$.

Lemma 3.3.4 *α_ℓ is surjective and additive.*

Proof The surjectivity property follows from the application of the definition of α_ℓ. The additivity property follows from structural induction using the recursive functions defined for mappings ϕ (eq. 3.1) and α_ℓ (eq. 3.2).

Once again, context-sensitive analyses may be derived by defining $\llparenthesis$ and $\rrparenthesis$, the set of opening and closing contexts, respectively, and then defining appropriate context abstraction $\llparenthesis^\ell \sqsubseteq \llparenthesis^{Abs}$.

3.4 Analysis of Obfuscated Assembly Programs

We now turn our attention to context-sensitive analysis of assembly programs in which the *call* and *ret* instructions may be obfsucated. The semantics of the classic *call* and *ret* instructions consist of two parts: manipulation of return address on the stack and transfer of program control. To obfuscate a procedure call (or return from a call) the two parts of the semantics of the instructions may be separated and performed using other instructions. Further, all instructions participating in simulating a call or a return may not be contiguous in the code; they may be distributed and/or intermixed with other instructions. On the other hand, *call* (*ret*) instructions may be employed for purposes other than making (returning from) a procedure call. For instance, a *call* instruction may be used to transfer control, but the return address may be discarded. Such obfuscation relating to procedure calls are enumerated in (LAKHOTIA; KUMAR, 2004).

Procedure call and return obfuscations thwart analysis of assembly program by attacking an important step needed for interprocedural analysis: identification of procedures and creation of a call-graph (LAKHOTIA; SINGH, 2003). Most assembly languages do not provide any mechanism for encapsulating procedures. Thus, disassemblers use *call* and *ret* instructions to determine procedure boundaries and to create the call-graph (IDAPRO, 2009). When these instructions are obfuscated, the procedures identified and the call-graph created may be questionable and any subsequent interprocedural analyses circumspect.

3.4.1 Programming language

To present our analysis of assembly programs where *call* and *ret* instructions are obfuscated, we first introduce a simple assembly language that does not contain these instructions. Instead, the language provides primitives that manipulate the stack pointer and the instruction pointer, both of which are registers in IA32 architecture. Thus, our

Syntactic Categories:

$b \in \mathbf{B}$	(boolean expressions)
$e, e' \in \mathbf{E}$	(integer expressions)
$i \in \mathbf{I}$	(instructions)
$l, l' \in \mathbf{L} \subseteq \mathbb{Z}$	(labels)
$z \in \mathbb{Z}$	(integers)
$p \in \mathbf{P}$	(programs)
$r \in \mathbf{R}$	(references)

Syntax:

$$\begin{aligned}
e &::= l \mid z \mid r \mid *\, r \mid e_1 \; op \; e_2 \qquad (op \in \{+,\ -,\ *,\ /, ...\}) \\
b &::= \; true \mid false \mid e_1 < e_2 \mid \neg b \mid b1 \;\&\&\; b2 \\
i &::= l : \; esp \; = \; esp + e \; \centerdot \; eip = e' \mid \\
&\qquad l : \; esp \; = e \; \centerdot \; eip = e' \mid \\
&\qquad l : \; *esp \; = e \; \centerdot \; eip = e' \mid \\
&\qquad l : \; r \; = e \; \centerdot \; eip = e' \mid \\
&\qquad l : \; *r \; = e \; \centerdot \; eip = e' \mid \\
&\qquad l : \; if \; (b) \; eip = e; \; eip = l' \\
p &::= seq(i)
\end{aligned}$$

Figure 16: An x86-like assembly language.

language captures the essential properties needed to present our algorithm for performing context-sensitive analysis of obfuscated assembly programs.

Figure 16 presents the syntax of the language we use to model our analysis. A program p in this language consists of a sequence of instructions, *viz.* $seq(i)$. Instructions can be either conditional or unconditional. A conditional instruction at a label l has the form "$l : \; if \; (b) \; eip = e; \; eip = l'$", where b is a boolean expression and e is an integer expression, which evaluates to the label of the instruction to execute when b evaluates to $true$; and l' is the label of the instruction to execute when b evaluates to $false$. An unconditional instruction at a label l has the form of "$l : \; assign \; \centerdot \; eip = e$", where $assign$ may assign the result of evaluating an expression to a reference (a register or memory location), or a memory location pointed to by a reference. The component "$eip = e'$" of an unconditional instruction assigns to the instruction pointer eip the label of the command to be executed next. The notation $x \centerdot y$ represents that both arguments x and y are not necessarily executed in sequence.

Our language assumes a unique symbol esp representing the stack pointer, which may be a register or a memory location. As noted by the rules in Figure 16 for instruction i, the operations on (or through) the stack pointer are distinguishable from other operations. The analysis presented assumes that the stack grows towards lower memory addresses,

Semantic domains:

$\delta \in \Delta = R + L \rightarrow \mathbb{Z}$	(store environment)
$s \in \Sigma = I \times \Delta$	(program states)
$z \in \mathbb{Z}$	(integers)
$\mathcal{B} = \{true,\ false\}$	(truth values)

Semantic functions:

$F_{esp} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta$

$\quad F_{esp}\ z\ \delta = [esp \mapsto ((\delta\ esp) + z)]\delta$

$F_{reset} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta$

$\quad F_{reset}\ z\ \delta = \ [esp \mapsto z]\delta$

$F_{*esp} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta$

$\quad F_{*esp}\ z\ \delta = [l' \mapsto z]\delta,$ where $l' = \delta\ esp$

$F_{assign} : R \rightarrow \mathbb{Z} \rightarrow \Delta \rightarrow \Delta$

$\quad F_{assign}\ r\ z\ \delta = [r \mapsto z]\delta$

$F_{*assign} : R \rightarrow \mathbb{Z} \rightarrow \Delta \rightarrow \Delta$

$\quad F_{*assign}\ r\ z\ \delta = [l' \mapsto z]\delta,$ where $l' = \delta\ r$

$F_{expr} : E \rightarrow \Delta \rightarrow \mathbb{Z}$

$\quad F_{expr}[\![l]\!]\delta = l$

$\quad F_{expr}[\![z]\!]\delta = z$

$\quad F_{expr}[\![r]\!]\delta = \delta\ r$

$\quad F_{expr}[\![*r]\!]\delta = \delta\ l$, where $l = \delta\ r$

$\quad F_{expr}[\![e_1\ op\ e_2]\!]\delta = F_{expr}[\![e_1]\!]\delta\ op\ F_{expr}[\![e_2]\!]\delta$

$F_{bool} : B \rightarrow \Delta \rightarrow \mathcal{B}$

$\quad F_{bool}[\![true]\!]\delta = true$

$\quad F_{bool}[\![false]\!]\delta = false$

$\quad F_{bool}[\![e_1 < e_2]\!]\delta = F_{expr}[\![e_1]\!]\delta < F_{expr}[\![e_2]\!]\delta$

$\quad F_{bool}[\![\neg b]\!]\delta = \neg F_{bool}[\![b]\!]\delta$

$\quad F_{bool}[\![b_1\ \&\&\ b_2]\!]\delta = F_{bool}[\![b_1]\!]\delta \wedge F_{bool}[\![b_2]\!]\delta$

Figure 17: Semantic domains and functions for our semantics.

but it can be changed trivially to accommodate the opposite convention.

Though our language does not explicitly model *call*, *ret*, *push*, or *pop* instructions, equivalent behavior may be performed using primitives of our language. For example, a "*call l*" instruction may be mapped to the following sequence of instructions in our language:

$$l_0 : esp = esp - 1\ \centerdot\ eip = l_1$$

$$l_1 : *esp = l_2\ \centerdot\ eip = l$$

where l_2 is the address of the instruction after the call instruction. It is not necessary that these two instructions appear contiguously in code. A "*ret*" instruction may be mapped to the following instruction in our language:

$$l_0 : esp = esp + 1\ \centerdot\ eip = *esp$$

Figures 17 and 18 present the semantics for our model language of Figure 16. Figure 17 describes the semantic domains and semantic functions, while Figure 18 presents the transition relation for our semantics.

A program state is represented by a pair $(i, \delta) \in I \times \Delta$, where i is the next instruction to be executed in the store environment δ. Thus, $\Sigma = I \times \Delta$ denotes the set of all possible program states. A store environment $\delta \in \Delta$ is represented by the mapping of disjoint union of references and labels to integers, denoted by $R+L \rightarrow \mathbb{Z}$. The semantic functions are F_{esp}, F_{reset}, F_{*esp}, F_{assign}, $F_{*assign}$, F_{expr} and F_{bool}.

The stack pointer operator F_{esp} changes the current stack pointer esp by either increasing or decreasing it. It takes as input an integer z and a store environment δ, and returns an updated store environment, where the stack pointer is modified (increased or decreased) by z. The reset operator F_{reset} resets the stack pointer, assigning an integer to the stack pointer esp. It takes as input an integer z and a store environment δ, and returns an updated store environment, where z is assigned to the stack pointer. The stack pointer assignment operator F_{*esp} changes the top value of the stack pointer esp. It takes as input an integer z and a store environment δ, and returns an updated store environment, where the value z is assigned to the top value of the stack pointer.

The assignment operator F_{assign} assigns a value to a reference. It takes as input a reference r, an integer z and a store environment δ, and returns an updated store environment that holds z assigned to r. The pointer assignment operator $F_{*assign}$ assigns a value to a location. It takes as input a reference r, an integer z and a store environment δ, and returns an updated store environment that holds z at the specified location pointed by r. The expression evaluation F_{expr} evaluates an integer expression to an integer. It takes as input an integer expression e and a store environment δ, and returns an integer, which is the result of evaluation of e in the store environment δ. The boolean evaluation F_{bool} performs a boolean evaluation through a combination of logical operators and boolean comparisons.

Transition relation:

$$
\begin{aligned}
&\mathcal{I} : \Sigma \rightarrow \wp(\Sigma)\\
&\quad \mathcal{I}([\![l:\ esp\ =\ esp + e\ \centerdot\ eip = e']\!],\ \delta) = \{((F_{expr}\ e'\ \delta), F_{esp}\ (F_{expr}\ e\ \delta)\ \delta)\}\\
&\quad \mathcal{I}([\![l:\ esp\ = e\ \centerdot\ eip = e']\!],\ \delta) = \{((F_{expr}\ e'\ \delta), F_{reset}\ (F_{expr}\ e\ \delta)\ \delta)\}\\
&\quad \mathcal{I}([\![l:\ *esp\ = e\ \centerdot\ eip = e']\!],\ \delta) = \{((F_{expr}\ e'\ \delta), F_{*esp}\ (F_{expr}\ e\ \delta)\ \delta)\}\\
&\quad \mathcal{I}([\![l:\ r\ = e\ \centerdot\ eip = e']\!],\ \delta) = \{((F_{expr}\ e'\ \delta), F_{assign}\ r\ (F_{expr}\ e\ \delta)\ \delta)\}\\
&\quad \mathcal{I}([\![l:\ *r\ = e\ \centerdot\ eip = e']\!],\ \delta) = \{((F_{expr}\ e'\ \delta), (F_{*assign}\ r\ (F_{expr}\ e\ \delta)\ \delta)\}\\
&\quad \mathcal{I}([\![l:\ if\ (b)\ eip = e;\ eip = l']\!],\ \delta) = \begin{cases} \{((F_{expr}\ e\ \delta), \delta)\} & \text{if } true\ = (F_{bool}\ b\ \delta)\\ \{(l',\ \delta)\} & \text{if } false\ = (F_{bool}\ b\ \delta)\end{cases}
\end{aligned}
$$

Figure 18: Transition relation for our semantics.

The transition relation between program states is defined as $\mathcal{I} : \Sigma \rightarrow \wp(\Sigma)$, *i.e.*, the transition relation represents the behavior of an instruction i when executed in a certain store environment δ. Given a program state $s \in \Sigma$, the semantic function $(\mathcal{I}\ s)$ gives the set of possible successor states of s.

The transition relation, written for the set of all possible states, may be specialized for the states of a specific program as follows. Let $\Sigma^p = p \times \Delta^p$ be the set of states of a program p, then the transition relation $\mathcal{I}^p : \Sigma^p \rightarrow \wp(\Sigma^p)$ on program p is: $(\mathcal{I}^p\ i\ \delta) = \{(i', \delta') \mid (i', \delta') \in (\mathcal{I}\ i\ \delta), i' \in p,\ \mathit{and}\ \delta, \delta' \in \Delta^p\}$.

The concrete trace semantics for a program p is given by the least fixed point of the following function:

$$\mathcal{F}^p\ T = \Sigma^p_i \cup \{\sigma.s.s' \mid \sigma.s \in T \wedge s' \in \mathcal{I}^p\ s\}$$

where T is a set of finite partial traces; σ is a sequence of program states $s_0 ... s_n$ of length $|\sigma| > 0$ such that $\forall i \in [1, n) : s_i \in (\mathcal{I}^p\ s_{i-1})$ and $s_0 \in \Sigma^p_i$, the set of initial states. Following section 3.2, the concrete context-trace semantics can be obtained by the least fixed point of $\mathcal{F}_c : \llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*) \longrightarrow \llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*)$ which is mapped from $\mathcal{F} : \wp(\Sigma^*) \longrightarrow \wp(\Sigma^*)$.

3.4.2 Stack-context

We now define the sets $\llparenthesis'_{asm}$ and $\rrparenthesis'_{asm}$, which are the sets of instructions that open and close contexts, respectively, based on operations on the stack pointer. An instruction opens a context, i.e., belongs in $\llparenthesis'_{asm}$, if it decrements the stack pointer.

$$\llparenthesis'_{asm} \triangleq \{i | \exists n \in \mathbb{N}, \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) - n\}$$

Analogously, an instruction closes a context, i.e., belongs in $\rrparenthesis'_{asm}$, if it increments the stack pointer.

$$\rrparenthesis'_{asm} \triangleq \{\ i \mid \exists n \in \mathbb{N},\ \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) + n\}$$

Consider the class of programs in which (a) instructions that modify the stack pointer always increment or decrement it by a statically known constant and (b) that constant is the same for all instructions. This class includes programs that use only *call*, *ret*, *push*, and *pop* instructions to modify the stack pointer. Programs generated by conventional compilers typically fall in this class. For this class of programs the analysis domain $\llparenthesis'^{\ell}_{asm} \rightarrow \mathit{Abstore}$ or $\llparenthesis'^{k}_{asm} \rightarrow \mathit{Abstore}$ may be used to derive a context-sensitive analysis.

Now consider the programs that meet constraint (a), but not (b). That is, programs in which the increment/decrement applied to a stack pointer can be statically determined, but not all instructions use the same constant. Since the size of space allocated/deallocated on the stack is not the same, a closing context statement may not remove the entire context on the top of the stack. The analysis can be trivially extended for this class of programs by statically introducing pseudo instructions such that all stack pointer operations use the same constant.

Obfuscated programs, however, may not meet either of the constraints. They may contain instructions that modify the stack pointer by direct assignment of values, such as using the instruction $l : esp = e \centerdot eip = e$, or contain instructions that increment or decrement the stack pointer by an expression whose value cannot be determined statically. In the absence of any further information about the possible values of the expression, say from using the *Abstore*, to derive a safe analysis a worst case assumption would need to be made.

To analyze the most general class of assembly programs, we need to develop a concrete context-trace semantics that allows for non-fixed size contexts. The set of opening contexts for such semantics may be represented by the domain: $\llparenthesis_{asm} \subseteq I \times \mathbb{N}$, meaning that a context is a pair of a statement and stack units. The set of closing contexts is represented by the domain $\rrparenthesis_{asm} \subseteq I \times \mathbb{N}$. The domains are described as follows:

$$\begin{aligned}
\llparenthesis_{asm} &\triangleq \{(i,n) \mid \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) - n\} \\
\rrparenthesis_{asm} &\triangleq \{(i,n) \mid \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) + n\}
\end{aligned}$$

A context string is a sequence belonging to $\llparenthesis^*_{asm}$. A function $\Pi_{asm} : \Sigma^* \rightarrow \llparenthesis^*_{asm}$ may now be defined as a function that maps a trace to its context string. This function accounts for creation and destruction of varying size contexts.

$$\begin{aligned}
\Pi_{asm} :&\Sigma^*_{asm} \rightarrow \llparenthesis^*_{asm} \\
&\Pi_{asm}\ \sigma\ \triangleq (\Pi'\ \sigma\ \epsilon) \\
\Pi'_{asm} :&\Sigma^*_{asm} \rightarrow \llparenthesis^*_{asm} \rightarrow \llparenthesis^*_{asm} \\
&\Pi'_{asm}\ \epsilon\ \nu\ \triangleq \nu \\
&\Pi'_{asm}\ s_1.\epsilon\ \nu\ \triangleq \nu \\
&\Pi'_{asm}\ s_1.s_2.\sigma\ \nu\ \triangleq (\Pi'_{asm}\ s_2.\sigma\ (\pi\ (i_1,\ n)\ \nu)) \\
&where\ n = (\delta_2\ esp) - (\delta_1\ esp), s_1 = (i_1, \delta_1), \delta_2 = s_2 \downarrow 2
\end{aligned}$$

$$\begin{aligned}
\pi_{asm} :&(I \times \mathbb{N}) \rightarrow \llparenthesis^*_{asm} \rightarrow \llparenthesis^*_{asm}\\
&\pi_{asm}\ (i,0)\ \nu \triangleq \nu\\
&\pi_{asm}\ (i,n)\ \nu \triangleq (i,-n).\nu,\ \text{where } n < 0\\
&\pi_{asm}\ (i,n)\ (j,m).\nu \triangleq \begin{cases} (j, m-n).\nu & \text{if } m > n\\ \pi_{asm}\ (i, n-m)\ \nu & \text{otherwise}\end{cases}\\
&\mathit{where}\ n > 0
\end{aligned}$$

The function π_{asm} above is a counter-part of the function π described in section 3.2. It performs the *push* and *pop* operations on context string depending on the value of n. A negative value of n implies a *push* operation when the stack grows towards lower memory addresses. Correspondingly, a positive value of n implies a *pop* operation.

The domains $\llparenthesis_{asm}$ and $\rrparenthesis_{asm}$ represent the concrete domains. However, since $\mathbb{N}$ forms an infinite lattice, so do these two domains. Besides, the value of n representing the size of the context, though available in concrete analysis, may not be statically computable. Hence, we need an abstraction of these domains. We use the formulation used in constant-propagation to abstract $\mathbb{N}$. Let N represent the flat lattice consisting of the set of numbers in N and the special values $\top$ and $\bot$. The lattice is flat in that $\forall n, n_1, n_2 \in \mathbb{N}\ :\ \bot \sqsubseteq_N n$, $n \sqsubseteq_N \top$, and if $n_1 \neq n_2$ then $n_1 \sqcup n_2 = \top$. We can now define the abstract context domains $\hat{\llparenthesis}_{asm} = I \times N$ and $\hat{\rrparenthesis}_{asm} = I \times N$. The ℓ-context abstraction of $\hat{\llparenthesis}^*_{asm}$, denoted by $\hat{\llparenthesis}^\ell_{asm}$, will be used in the context-sensitive analysis of assembly programs.

Let us compare the difference between the stack context and calling-context of a non-obfuscated program. It is apparent that the set $\hat{\llparenthesis}_{asm}$ specialized for instructions in a program p may be larger when using stack context than for calling-context. This is because when using stack context, a context is created not only for *call* instructions, but also for *push* instructions. What is the implication of these extra nodes on the computational complexity of the analysis? The complexity depends on two factors. The total number of context strings created for a program and the number of context strings reaching a statement. The total number of context strings (which includes all partial strings) will increase as will the length of the strings. However, the number of context strings reaching an instruction will be the same for both methods. This is because when using stack context the *push* instructions will simply increase the length of the context strings passing through them, but not increase the number of the context strings. Thus, for an appropriate representation of context strings, such as using BDD (WHALEY; LAM, 2004),(ZHU; CALMAN, 2004), the complexity of the algorithm will remain unchanged.

3.4.3 Modeling transfer of control

To complete the analysis of programs in the model assembly language we still need to develop abstraction for modeling the transfer of control. In the concrete semantics, the register *eip* represents the instruction pointer. Upon execution of each instruction the *eip* is updated with the label (a numerical value) of the next instruction to be executed. The value of the label may be computed from an expression involving values of registers and memory locations. Thus, to model transfer of control we need an abstraction of the values computed by an expression.

We use Balakrishnan and Reps' Value-Set Analysis (VSA) (BALAKRISHNAN; REPS, 2004),(BALAKRISHNAN, 2007) to recover information about the contents of memory locations and registers manipulated by an assembly program. VSA uses the domain $RIC = \mathbb{N} \times \mathbb{Z} \times \mathbb{Z}$ to abstract $\wp(\mathbb{Z})$. A Reduced Interval Congruence (RIC) is a hybrid domain that merges the notion of interval with that of congruence. Since an interval captures the notion of upper and lower bound (COUSOT; COUSOT, 1976) and a congruence captures the notion of stride information, one can use RIC's to combine both worlds. An RIC is a formal, well-defined, and well-structured way of representing a finite set of integers that are equally apart. For example, say we need to over-approximate the set of integers $\{1, 3, 5, 9\}$. An interval over-approximation of this set would be $[1, 9]$ which contains the integers $1, 2, 3, 4, 5, 6, 7, 8$, and 9; a congruence representation would note that $1, 3, 5$, and 9 are odd numbers and over-approximate $\{1, 3, 5, 9\}$ with the set of all odd numbers $1, 3, 5, 7, ...$ Both of these approximations are too conservative to achieve a tight approximation of such a small set. The set of odd numbers is infinite and the interval does not capture the stride information and hence loses some precision. In the above example, the RIC $2[1, 9]$, which represents the set of integer values $\{1, 3, 5, 7, 9\}$, clearly is a tighter over-approximation of our set. Formally written, a value $s[lb, ub] \in RIC$, where $s \in \mathbb{N}$ and lb, $ub \in \mathbb{Z}$ are mapped to $\wp(\mathbb{Z})$ by the following concretization map:

$$\gamma(s[lb, ub]) = \{z | lb \leq z \leq ub, z \equiv lb \pmod{s}\}.$$

Thus, $\gamma(2[1, 9]) = \{1, 3, 5, 7, 9\}$.

Since memory addresses are numerical values, the domain RIC provides a safe approximation of the set of numerical values as well as addresses held by a register or a memory location. Whether the values represented by an element $s[lb, ub]$ are memory addresses or numerical values follows from how the information is used in an instruction. When the value is assigned to *eip* it is treated as a memory address, in particular, a label of an instruction. Similarly, the value represents a memory address when used in an indirect memory operand, such as when computing the expression $* \, r$.

3.4.4 Semantic domain and algorithm

We now discuss the derivation of semantic domain and algorithm for the context sensitive version of Venable's *et al.*'s algorithm. Venable *et al.*'s algorithm (VENABLE et al., 2005) is a static analyzer that can track stack manipulations where the stack pointer may be saved and restored in memory or registers. It combines Lakhotia and Kumar's abstract stack graph (ASG) (LAKHOTIA; KUMAR, 2004) with Reps and Balakrishnan's value-set analysis (VSA) (BALAKRISHNAN; REPS, 2004).

The ASG domain was introduced by (LAKHOTIA; KUMAR, 2004) as an abstraction of the set $\wp(\llparenthesis^{*}_{asm})$. Each element of $\wp(\llparenthesis^{*}_{asm})$ represents a set of (partial) stack strings. To create a finite representation of a possibly infinite set of such strings, Lakhotia *et al.* abstracted $\wp(\llparenthesis^{*}_{asm})$ using the domain $ASG = \wp(\hat{\llparenthesis}_{asm}) \times \wp(\hat{\llparenthesis}_{asm} \times \hat{\llparenthesis}_{asm})$. The relevant abstraction/concretization maps to show that $\wp(\llparenthesis^{*}_{asm}) \sqsubseteq \wp(\hat{\llparenthesis}_{asm}) \times \wp(\hat{\llparenthesis}_{asm} \times \hat{\llparenthesis}_{asm})$ may be derived from the following insight. The first component of the ASG domain, $\wp(\hat{\llparenthesis}_{asm})$, represents the set of stack tops. Starting from a node in the set of stack tops, a path in the graph representing the second component, $\wp(\hat{\llparenthesis}_{asm} \times \hat{\llparenthesis}_{asm})$, gives a partial stack string.

Venable *et al.* combined ASG and VSA methods to derive the context-insensitive analysis $I \to R + L \to ASG \times RIC$. We derive a context-sensitive equivalent of this analysis using the domain $\hat{\llparenthesis}^{\ell}_{asm} \to I \to R + L \to RIC$. This domain does not include a mapping from an instruction to its ASG because the abstraction of the stack is implicitly available in a context-sensitive analysis. The analyzer may be derived using the chain of Galois connections. To ensure termination of the analyzer, we use the widening operator for RIC domain as given by (BALAKRISHNAN, 2007) to accelerate fixed point computation.

The pseudo-code for the top-level procedure and its fluxogram are shown in Figures 19 and 20, respectively. Procedure *ContextSensitive-ℓ* interprets a disassembled input program and returns the same program with state information '$\delta^{\#} \in \Delta^{\#} = R + L \to RIC$' attached to each instruction. This procedure initializes all values of memory location $R + L$ and the work-list. Procedure $\mathcal{I}^{\#}$ interprets the given instruction 'i' using the given state '$\delta^{\#}$', while procedures *push-ℓ* and *pop-ℓ* manipulate the context strings '$\nu_{\ell} \in \llparenthesis^{\ell}_{asm}$' during the context-sensitive analysis. In the case of the *push* instruction, we have 'i **in** $\llparenthesis_{asm}$' and we have 'i **in** $\rrparenthesis_{asm}$' in the case of *pop* instruction. The value '*next*' represents the transfer of control during the interpretation, which is acquired from register *eip*. The analysis ends when the work-list is null.

```
decl worklist: Set of $(\llparenthesis^{\ell}_{asm} \times I \times L \times \Delta^{\#})$
decl next: Set of $L$

proc ContextSensitive-$\ell$()
    $\delta^{\#'}$ := Initial values of memory locations
    $l' := \delta^{\#'}(eip)$ // next address to be interpreted
    $i' := getInstruction(l')$
    $worklist := (\epsilon, i', l', \delta^{\#'})$
    while $(worklist <> \emptyset)$ do
        Select and remove $(\nu_\ell, i, l, \delta^{\#})$ from worklist
        $\delta^{\#'} := \mathcal{I}^{\#}(i, \delta^{\#})$
        $next := \delta^{\#'}(eip)$
        for $l$ in $next$ do
            if $i$ in $\llparenthesis_{asm}$ then
                push-$\ell(\nu_\ell, i, l, \delta^{\#'})$ // Function to manipulate push in the stack
            else if $i$ in $\rrparenthesis_{asm}$ then
                pop-$\ell(\nu_\ell, i, l, \delta^{\#'})$ // Function to manipulate pop in the stack
            else
                $i := getInstruction(l)$
                put $(\nu_\ell, i, l, \delta^{\#'})$ in worklist
            endif
        end for
    end while
end proc
```

Figure 19: Pseudo-code for the top-level procedure of our algorithm.

A version of this algorithm for k-context analysis can be achieved by replacing the calls for *push-ℓ* and *pop-ℓ* by calls to *push-k* and *pop-k*. These functions are modified versions of the former ones where a k length is applied to the string.

The pseudo-code for the $\mathcal{I}^{\#}$ procedure is shown in Figure 21, and the semantic functions are shown in Figure 22. Each instruction described in Figure 16 is evaluated separately. For matching purposes, we use function notation to describe the interpretation of each instruction in our abstracted semantics. This notation can be related to the notation used to describe our concrete semantics. The main differences for the abstract and concrete semantics are described next. In the abstract semantics, the functions denoted by a superscript $^{\#}$ may result in more than one address, while in the concrete semantics we just have one address. Moreover, the abstract semantic functions contain an extra label in its signature, which represents an abstraction of the real address needed to represent context strings and the abstract stack graph. Another difference is related to conditional instructions, in which the abstract semantics result in a union of the paths since our

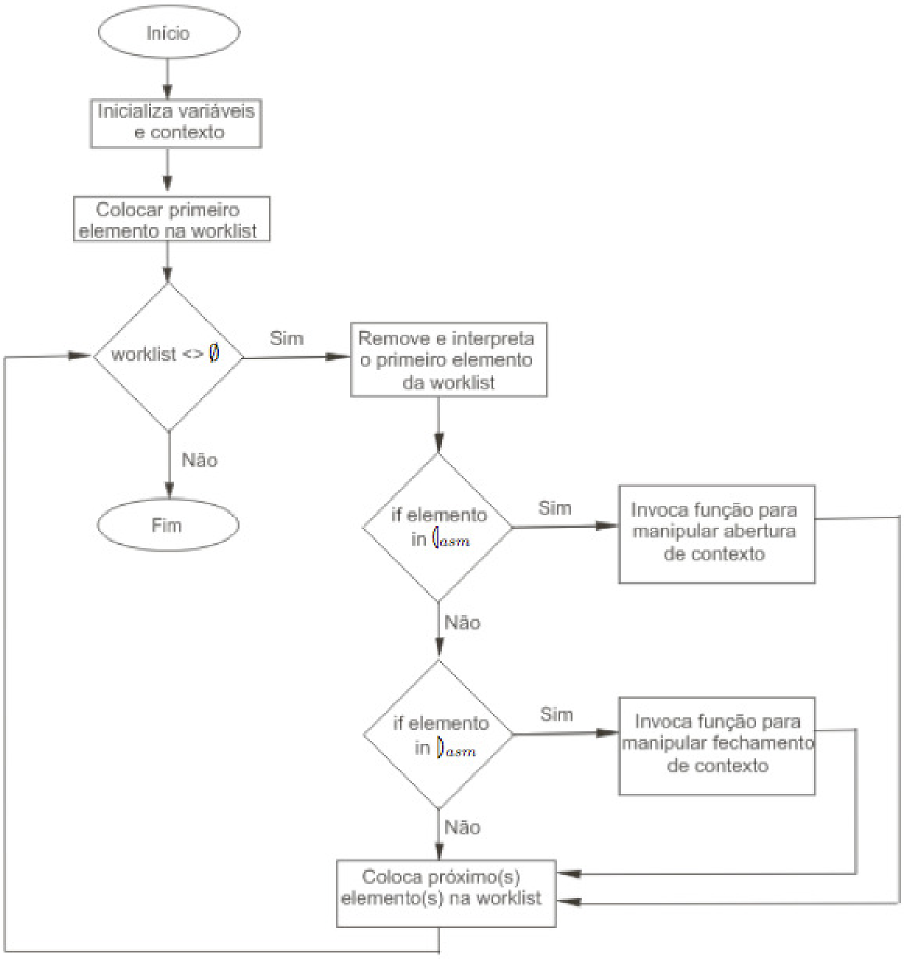

Início
Inicializa variáveis e contexto
Colocar primeiro elemento na worklist
worklist <> ∅
Sim
Não
Fim
Remove e interpreta o primeiro elemento da worklist
if elemento in $($_{asm}$
Sim
Invoca função para manipular abertura de contexto
Não
if elemento in $)$_{asm}$
Sim
Invoca função para manipular fechamento de contexto
Não
Coloca próximo(s) elemento(s) na worklist

Figure 20: Fluxogram for the top-level procedure of our algorithm.

interpreter does not consider carry flags.

The pseudo-code for the *push-ℓ* procedure is shown in Figure 23. This procedure represents a *push* operation on the context string. The address of the *push* instruction is prepended in the context string if the address is not in the context string or the context string is abstracted by the cycle of the *push* instruction address.

Figure 24 shows the pseudo-code for the *pop-ℓ* procedure, and the pseudo-code for the *recursivePaths* procedure. The *pop-ℓ* procedure represents a *pop* operation on the context string. In the case of no recursion, the first address is popped from the context string. In the recursive case, the *recursivePaths* procedure traverses the ASG to find all possible context strings which may result in the current context string. Basically, this procedure finds all possible recursive cycle paths in the ASG starting with the opening context l' on the top of the context string to all successors of l'.

```
proc $\mathcal{I}^{\#}(i, \delta^{\#})$
    switch $i$
        case $[\![l : \; esp \; = \; esp + e \centerdot eip = e']\!]$
            $(F^{\#}_{expr} \; e' \; \delta^{\#}) \times \{F^{\#}_{esp} \; l \; (F^{\#}_{expr} \; e \; \delta^{\#}) \; \delta^{\#}\}$
        case $[\![l : \; esp \; = e \centerdot eip = e']\!]$
            $(F^{\#}_{expr} \; e' \; \delta^{\#}) \times \{F^{\#}_{reset} \; l \; (F^{\#}_{expr} \; e \; \delta^{\#}) \; \delta^{\#}\}$
        case $[\![l : \; *esp \; = e \centerdot eip = e']\!]$
            $(F^{\#}_{expr} \; e' \; \delta^{\#}) \times \{F^{\#}_{*esp} \; l \; (F^{\#}_{expr} \; e \; \delta^{\#}) \; \delta^{\#}\}$
        case $[\![l : \; r \; = e \centerdot eip = e']\!]$
            $(F^{\#}_{expr} \; e' \; \delta^{\#}) \times \{F^{\#}_{assign} \; l \; r \; (F^{\#}_{expr} \; e \; \delta^{\#}) \; \delta^{\#}\}$
        case $[\![l : \; *r \; = e \centerdot eip = e']\!]$
            $(F^{\#}_{expr} \; e' \; \delta^{\#}) \times \{F^{\#}_{*assign} \; l \; r \; (F^{\#}_{expr} \; e \; \delta^{\#}) \; \delta^{\#}\}$
        case $[\![l : \; if \; (b) \; eip = e; \; eip = l']\!]$
            $((F^{\#}_{expr} \; e \; \delta^{\#}) \times \delta^{\#}) \cup \{(l', \; \delta^{\#})\}$
    end switch
end proc
```

Figure 21: $\mathcal{I}^{\#}$ procedure of our algorithm.

3.4.5 Soundness

The concrete context-trace semantics is given by the least fixed point of the function $\mathcal{F}_c : \llparenthesis^{*}_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*) \longrightarrow \llparenthesis^{*}_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*)$, where $\Sigma = I \times R + L \rightarrow \mathbb{Z}$. The context-trace semantics of the context-sensitive analyzer is given by the least fixed point of the function $\mathcal{F}^{\#} : (\hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC) \longrightarrow (\hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC)$.

Lemma 3.4.1 $\llparenthesis^{*}_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*) \sqsubseteq \hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC.$

Proof It follows from Lemma 3.3.4 that $\llparenthesis^{*}_{asm} \sqsubseteq \hat{\llparenthesis}^{\ell}_{asm}$, and it follows from Balakrishnan and Reps' (BALAKRISHNAN, 2007) that $\wp(\mathbb{Z}) \sqsubseteq RIC$. Then, it follows by monotone and total function space combination techniques of Galois connections that

$$\begin{aligned}
\llparenthesis^{*}_{asm} \xrightarrow{\Pi_{asm}} \wp((\Sigma)^*) &\sqsubseteq \llparenthesis^{*}_{asm} \rightarrow (\wp(\Sigma))^* \\
&\equiv \llparenthesis^{*}_{asm} \rightarrow (I \rightarrow R + L \rightarrow \wp(\mathbb{Z}))^* \\
&\sqsubseteq \llparenthesis^{*}_{asm} \rightarrow I \rightarrow R + L \rightarrow \wp(\mathbb{Z}) \\
&\sqsubseteq \hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC. \quad \blacksquare
\end{aligned}$$

It follows from lemma 3.4.1 and the fixed point transfer theorem that $\mathcal{F}^{\#}$ is a sound approximation of $\mathcal{F}_c$. However, $\mathcal{F}^{\#}$ may not be complete *w.r.t* $\mathcal{F}_c$.

$$\delta^{\#} \in \Delta^{\#} = R + L \rightarrow RIC$$
$$ric \in RIC$$

$$\begin{array}{l}
F^{\#}_{esp} : L \rightarrow RIC \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad F^{\#}_{esp}\ l\ ric\ \delta^{\#} = \begin{array}[t]{ll} \text{if } ric = \{x\} \wedge x \leq 0 & Addnode\ l\ x\ \delta^{\#} \\ \text{else if } ric = \{x\} \wedge x > 0 & Succnode\ x\ \delta^{\#} \\ \text{else} & F^{\#}_{reset}\ l\ ric\ \delta^{\#} \end{array} \\
F^{\#}_{*esp} : L \rightarrow RIC \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad F^{\#}_{*esp}\ l\ ric\ \delta^{\#} = [l' \mapsto ric]\delta^{\#},\ where\ l' \in \delta^{\#}\ esp \\
F^{\#}_{assign} : L \rightarrow R \rightarrow RIC \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad F^{\#}_{assign}\ l\ r\ ric\ \delta^{\#} = [r \mapsto ric]\delta^{\#} \\
F^{\#}_{*assign} : L \rightarrow R \rightarrow RIC \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad F^{\#}_{*assign}\ l\ r\ ric\ \delta^{\#} = [l' \mapsto ric]\delta^{\#},\ where\ l' \in \delta^{\#}\ r \\
F^{\#}_{expr} : E \rightarrow \Delta^{\#} \rightarrow RIC \\
\quad F^{\#}_{expr}[\![l]\!]\delta^{\#} = l[1,1] \\
\quad F^{\#}_{expr}[\![z]\!]\delta^{\#} = z[1,1] \\
\quad F^{\#}_{expr}[\![r]\!]\delta^{\#} = \delta^{\#}\ r \\
\quad F^{\#}_{expr}[\![*r]\!]\delta^{\#} = \sqcup\{\delta^{\#}\ l \mid l \in \delta^{\#}\ r\} \\
\quad F^{\#}_{expr}[\![e_1\ op\ e_2]\!]\delta^{\#} = F^{\#}_{expr}[\![e_1]\!]\delta^{\#}\ op\ F^{\#}_{expr}[\![e_2]\!]\delta^{\#} \\
F^{\#}_{bool} : B \rightarrow \Delta^{\#} \rightarrow \wp(\mathcal{B}) \\
\quad F^{\#}_{bool}[\![true]\!]\delta^{\#} = true \\
\quad F^{\#}_{bool}[\![false]\!]\delta^{\#} = false \\
\quad F^{\#}_{bool}[\![e_1 < e_2]\!]\delta^{\#} = F^{\#}_{expr}[\![e_1]\!]\delta^{\#} < F^{\#}_{expr}[\![e_2]\!]\delta^{\#} \\
\quad F^{\#}_{bool}[\![\neg b]\!]\delta^{\#} = \neg F^{\#}_{bool}[\![b]\!]\delta^{\#} \\
\quad F^{\#}_{bool}[\![b_1\ \&\&\ b_2]\!]\delta^{\#} = F^{\#}_{bool}[\![b_1]\!]\delta^{\#} \wedge F^{\#}_{bool}[\![b_2]\!]\delta^{\#} \\
F^{\#}_{reset} : L \rightarrow RIC \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad F^{\#}_{reset}\ l\ ric\ \delta^{\#} =\ [esp \mapsto \bot]\delta^{\#}
\end{array}$$

$$\begin{array}{l}
Addnode : L \rightarrow \mathbb{Z} \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad Addnode\ \ l\ z\ \delta^{\#} = \begin{array}[t]{l} \text{if (z} < 0\text{) } Addnode\ (\ l\ (z+1)\ [esp \mapsto l^{z}]\delta^{\#}), \text{where } l^{z} \notin \delta^{\#} \\ \text{else}\quad \delta^{\#} \end{array} \\
Succnode : \mathbb{Z} \rightarrow \Delta^{\#} \rightarrow \Delta^{\#} \\
\quad Succnode\ \ z\ \delta^{\#} = \begin{array}[t]{l} \text{if (z} > 0\text{) } Succnode\ ((z-1)\ [esp \mapsto \sqcup\ succ(\delta^{\#}\ esp)]\delta^{\#}) \\ \text{else}\quad \delta^{\#} \end{array}
\end{array}$$

Figure 22: Abstracted semantic functions.

```
proc push-ℓ($\nu_\ell$ : $(\!|^\ell_{asm}$, $i$ : $I$, $l$ : $L$, $\delta^\#$ : $\Delta^\#$)
    let $l' = (i, x); i' = getInstruction(l)$ // x can be + or 1
    in  if $l'$ not in $\nu_\ell$ then
        // not recursive
            put $((i,1).\nu_\ell, i', l, \delta^\#)$ in worklist
        else
        // recursive case
            let $\nu_\ell = \nu'_\ell.l'.\nu''_\ell$
                put $((i,+).\nu''_\ell, i', l, \delta^\#)$ in worklist

        endif
end proc
```

Figure 23: *push-ℓ* procedure of our algorithm.

3.5 Examples

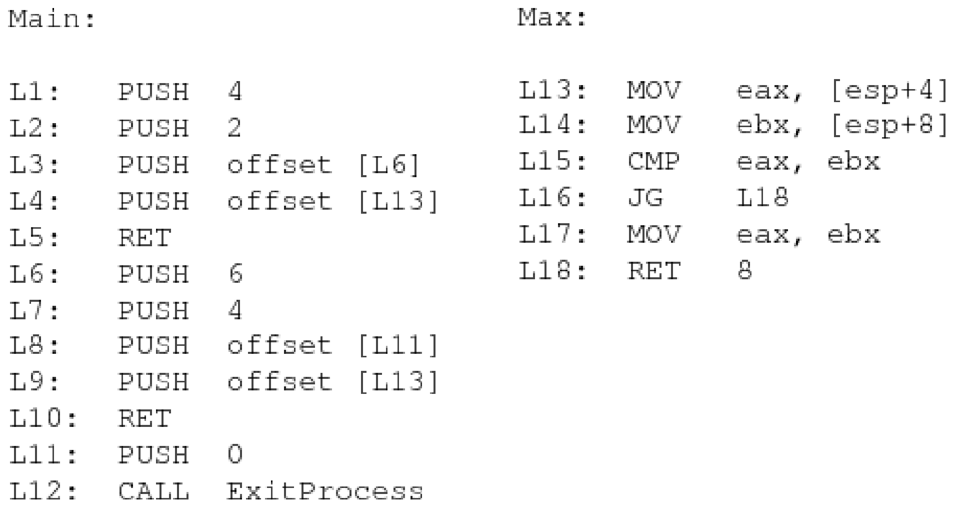
This section explains the context-sensitive analysis process of obfuscated code using stack-contexts. Figure 25 contains an assembly obfuscated program with two stack-contexts. The program consists of two functions: *Main* and *Max*. The function *Max* takes as input two numbers and returns as output the larger of the two numbers. The function *Main* pushes the two arguments onto the stack before calling *Max*, but instead of calling *Max* directly, it uses two *push* instructions and a *ret* instruction to achieve the same behavior. This obfuscation technique can effectively hide the boundary between the two procedures and result in a less accurate ICFG. Analysis methods relying on the interprocedural control flow graph may, in effect, produce less accurate results as well.

Our analysis is independent of the ICFG and is described next. After careful inspection, one may observe that in order to perform context-sensitive analysis of the code in Figure 25, we have to match the node $L18$ (end node of procedure *Max*) with nodes $L6$ and $L11$ (return nodes). Our analysis can correctly perform these matches. Table 6 provides the results for our context-sensitive analysis for this obfuscated code. It provides the in and out for each node. For simplicity, the results only contain valued memory locations. For example, in the line 'L1 In',$< \epsilon \mid eip = l_1 >$ represents that the empty context ϵ contains $eip = l_1$ (entry point) and that all other memory locations are indetermined. The following (incomplete) steps illustrate the context-sensitive interprocedural analysis process:

- Upon entry, we have 'L1 In' $< \epsilon \mid eip = l_1 >$. Instruction at L1 pushes a value onto the stack, consequently changing the stack-context to l_1. The result is represented

```
decl $visited$: $Set\ of\ L$
proc $pop\text{-}\ell(\nu_\ell : (\!|^{\ell}_{asm}, i : I, l : L, \delta^\# : \Delta^\#)$
    let $l' = (i', x) = top(\nu_\ell)$,
        $\nu'_\ell = rest(\nu_\ell)$,
        $i'' = getInstruction(l)$
    in  if $(x = 1)$ then
                $put\ (\nu'_\ell, i'', l, \delta^\#)$ in $worklist$
        else // recursive node on top
            for $(l', succ)$ in $ASG$ do
                $visited := \emptyset$
                $paths := recursivePaths(succ, l')$
                for $p$ in $paths$ do
                    $put\ (p.(i', 1).\nu'_\ell, i'', l, \delta^\#)$ in $worklist$
                    $put\ (p.(i', +).\nu'_\ell, i'', l, \delta^\#)$ in $worklist$
                end for
            end for
        endif
end proc

proc $recursivePaths(start : L, end : L) : Set\ of\ paths$
    $paths := \emptyset$
    if $start$ not in $visited$ then
        $put\ start$ in $visited$
        if $(start <> end)$
            for $(start, start')$ in $ASG$ do
                $paths := recursivePaths(start', end)$
                for p in paths do
                    $put(start.p)$ in $paths$
                end for
            end for
        end if
    end if
    return paths
end proc
```

Figure 24: *pop-ℓ* procedure of our algorithm.



```
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 25: Obfuscated call using *push/ret* instructions.

by ‘L1 Out $< l_1 \mid l_1 = 4, esp = l_1, eip = l_2 >$’. Instructions at L2, L3 and L4 perform in a manner similar to $L1$, consequently at ‘L4 Out’ we have $< l_4 l_3 l_2 l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, l_4 = l_{13}, esp = l_4, eip = l_5 >$.

- The *ret* instruction at L5 pops the address L13 from $esp = l_4$, changes the context to $l_3 l_2 l_1$, and transfers the control of execution to L13. Instructions from L13 to L17 have no effect on the context. Instruction L16 is a conditional jump and does not change the values. During evaluation, each possible target will be processed, and each resulting state is joined once the two execution paths meet. That is represented by the set $eip = \{l_{17}, l_{18}\}$ at line ‘L16 Out’. Instruction at L17 copies the value from *ebx* to *eax*. Instruction L18 pops the address L6 from $esp = l_6$, changes the context to ϵ, and transfers the control of execution to L6. However, since L18 can be reached from L16 and L17, the results of evaluating the two paths must be joined before processing L18. At line ‘L16 Out’, $eax = 2$, whereas at line ‘L17 Out’, $eax = 4$. The union of the two is the set {2, 4}, or the RIC 2[1,2].

- Instruction at L6 pushes a value onto the stack and changes the context to l_6. Instructions at L7, L8 and L9 perform in a manner similar to L6. Thus, at ‘L9 Out’ we have $< l_9 l_8 l_7 l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, l_9 = l_{13}, esp = l_9, eax = 2[1,2], ebx = 4, eip = l_{10} >$. The *ret* instruction at L10 pops the address L13 from $esp = l_9$, changes the context to $l_8 l_7 l_6$, and transfers the control of execution to L13. At this point, the analysis contains two contexts: $l_3 l_2 l_1$ and $l_8 l_7 l_6$. The evaluation from L13 to L18 now proceeds in the context $l_8 l_7 l_6$. Instruction at L18 pops and transfers the control of execution to $L11$ (return address off the top of the stack in the stack-context $l_8 l_7 l_6$). It also changes the stack-context to ϵ. Evaluation continues at L11,

Table 6: Stack contexts and associated values for interprocedural analysis of obfuscated binaries.

Nodes		Context and associated values
L1	In	$< \epsilon \mid eip = l_1 >$
L1	Out	$< l_1 \mid l_1 = 4, esp = l_1, eip = l_2 >$
L2	In	$< l_1 \mid l_1 = 4, esp = l_1, eip = l_2 >$
L2	Out	$< l_2l_1 \mid l_1 = 4, l_2 = 2, esp = l_2, eip = l_3 >$
L3	In	$< l_2l_1 \mid l_1 = 4, l_2 = 2, esp = l_2, eip = l_3 >$
L3	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eip = l_4 >$
L4	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eip = l_4 >$
L4	Out	$< l_4l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, l_4 = l_{13}, esp = l_4, eip = l_5 >$
L5	In	$< l_4l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, l_4 = l_{13}, esp = l_4, eip = l_5 >$
L5	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eip = l_{13} >$
L6	In	$< \epsilon \mid eax = 2[1,2], ebx = 4, eip = l_6 >$
L6	Out	$< l_6 \mid l_6 = 6, esp = l_6, eax = 2[1,2], ebx = 4, eip = l_7 >$
L7	In	$< l_6 \mid l_6 = 6, esp = l_6, eax = 2[1,2], ebx = 4, eip = l_7 >$
L7	Out	$< l_7l_6 \mid l_6 = 6, l_7 = 4, esp = l_7, eax = 2[1,2], ebx = 4, eip = l_8 >$
L8	In	$< l_7l_6 \mid l_6 = 6, l_7 = 4, esp = l_7, eax = 2[1,2], ebx = 4, eip = l_8 >$
L8	Out	$< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 2[1,2], ebx = 4, eip = l_9 >$
L9	In	$< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 2[1,2], ebx = 4, eip = l_9 >$
L9	Out	$< l_9l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, l_9 = l_{13}, esp = l_9, eax = 2[1,2], ebx = 4, eip = l_{10} >$
L10	In	$< l_9l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, l_9 = l_{13}, esp = l_9, eax = 2[1,2], ebx = 4, eip = l_{10} >$
L10	Out	$< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 2[1,2], ebx = 4, eip = l_{13} >$
L11	In	$< \epsilon \mid eax = 2[2,3], ebx = 6, eip = l_{11} >$
L11	Out	$< l_{11} \mid l_{11} = 0, esp = l_{11}, eax = 2[2,3], ebx = 6, eip = l_{12} >$
L12	In	$< l_{11} \mid l_{11} = 0, esp = l_{11}, eax = 2[2,3], ebx = 6, eip = l_{12} >$
L12	Out	$< l_{12}l_{11} \mid l_{11} = 0, l_{12} = ExitProc, esp = l_{12}, eax = 2[2,3], ebx = 6, eip = ExitProc >$
L13	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eip = l_{13} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 2[1,2], ebx = 4, eip = l_{13} >$
L13	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, eip = l_{14} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 4, eip = l_{14} >$
L14	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, eip = l_{14} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 4, eip = l_{14} >$
L14	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = l_{15} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = l_{15} >$
L15	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = l_{15} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = l_{15} >$
L15	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = l_{16} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = l_{16} >$
L16	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = l_{16} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = l_{16} >$
L16	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = \{l_{17}, l_{18}\} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = \{l_{17}, l_{18}\} >$
L17	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2, ebx = 4, eip = l_{17} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 4, ebx = 6, eip = l_{17} >$
L17	Out	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 4, ebx = 4, eip = l_{18} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 6, ebx = 6, eip = l_{18} >$
L18	In	$< l_3l_2l_1 \mid l_1 = 4, l_2 = 2, l_3 = l_6, esp = l_3, eax = 2[1,2], ebx = 4, eip = l_{18} >$ $< l_8l_7l_6 \mid l_6 = 6, l_7 = 4, l_8 = l_{11}, esp = l_8, eax = 2[2,3], ebx = 6, eip = l_{18} >$
L18	Out	$< \epsilon \mid eax = 2[1,2], ebx = 4, eip = l_6 >$ $< \epsilon \mid eax = 2[2,3], ebx = 6, eip = l_{11} >$

which proceeds to the end of the program.

This example shows how our proposed algorithm provides context-sensitivity in the analysis of obfuscated code, in which pieces of code are analyzed separately for different data flow values at different stack-contexts.

Figure 26 shows the same code, but using the *push/jmp* obfuscation. The function *Main*, instead of calling *Max* directly, pushes the return address onto the stack and jumps to *Max*. Analysis methods that rely on the correctness of an ICFG will surely fail when analyzing such code.

```
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 26: Obfuscated call using *push/jmp* instructions.

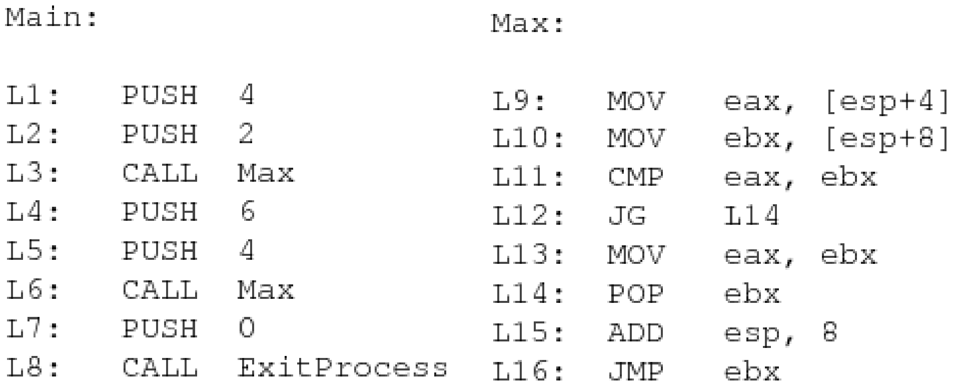
Upon entry, the stack-context is ϵ. Instruction at L1 pushes a value onto the stack, consequently changing the stack-context to l_1. Instructions at L2 and L3 perform in a manner similar to L1. After interpretation of instruction at L3, the stack-context is $l_3 - l_2 - l_1$. Instruction at L4 transfers the control to L11 (entry point of function *Max*), and the stack-context is left unchanged.

The next instruction evaluated is the target of the jump, or L11 in this case. Instructions from the nodes L11 to L15 have no effect on the stack-context. Instruction L16 implicitly pops and transfers the control of execution to L5, which is the return address off the top of the stack in the stack-context $l_3 - l_2 - l_1$. Instruction at L16 also changes the stack-context $l_3 - l_2 - l_1$ to ϵ, *i.e.*, it pops the return address and adds 8 bytes in the register *esp*.

Instruction at L5 pushes a value onto the stack, consequently changing the stack-context to l_5. Instructions at L6 and L7 perform in a manner similar to L5. After interpretation of instruction at L7, the stack-context is $l_7 - l_6 - l_5$. Similarly to the instruction at L4, instruction at L8 transfers the control to the function *Max*, and the context is left unchanged. As before, instructions from the nodes L11 to L15 have no effect on the stack-context. Instruction at L16 implicitly pops and transfers the control of execution to L9, which is the return address off the top of the stack in the stack-context $l_7 - l_6 - l_5$. Instruction at L16 also changes the stack-context $l_7 - l_6 - l_5$ to ϵ, *i.e.*, it pops the return address and adds 8 bytes in the register *esp*. Evaluation continues at L9, which proceeds to the end of the program.

In Figure 27, 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 (nodes L14 to L16). The stack pointer is adjusted using the *add* instruction at node L15.

Following the previous examples, at instruction L14, our analysis has two stack-contexts: $l_3 - l_2 - l_1$ and $l_6 - l_5 - l_4$. The *pop* instruction at L14 pops the address L4 in the context $l_3 - l_2 - l_1$ and L7 in the context $l_6 - l_5 - l_4$, consequently, changing the contexts to $l_2 - l_1$ and $l_5 - l_4$, respectively. The *add* instruction at L15 adds eight bytes to *esp*, then alters the context strings to ϵ. Instruction at L16 jumps indirectly to the address L4 or L7 depending on the address popped.



```
Main:                          Max:

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

Figure 27: Obfuscated return using *pop/jmp* instructions.

3.6 Discussion

We have presented a method for performing context-sensitive analysis for binaries in which calling-contexts cannot be discerned as in obfuscated binaries. Our method of context-sensitive analysis does not rely on finding procedure boundaries and determining procedure calls. Instead, it defines a context based on the state of the stack. We adapt prior work on context-sensitive interprocedural analysis using call-strings to use with the stack context. The notion of call-strings has in the past been described in terms of valid paths of an ICFG (SHARIR; PNUELI, 1981). We generalize the concept using abstract interpretation and define contexts using trace semantics.

The method presented enables context-sensitive analysis of obfuscated programs that cannot be analyzed by current methods, and hence improves upon the state-of-the-art. Yet, its performance on non-obfuscated programs is equally important. Treating any push on the stack to create a new context increases the number of nodes in a context-graph, a graph that contains the call-graph. Since the number of call-strings in a program grows exponentially, it is reasonable to ask whether adding more nodes in the call-graph will make matters worse. The addition of nodes for *push* instruction in the graph does not change the number of paths between any two call sites. Thus, although the context strings may be longer, the number of context strings at an instruction does not change with the addition of the new nodes. Alternatively, if the context-graph is constructed *a priori*, sequences of single-entry, single-exit nodes may be collapsed into single nodes (like blocks of a CFG).

In the next chapter, we present the empirical evaluation of the context sensitive and insensitive versions of Venable's algorithm.

4 Empirical evaluation

In this chapter, we present the results and insights of an empirical evaluation of context-sensitive analysis of obfuscated programs. Our evaluation is divided into two phases: 1) comparison of the two context abstractions ℓ- and k-context and 2) study of improvement in analysis of obfuscated code resulting from using context-sensitive version of Venable *et al.*'s analysis (VENABLE et al., 2005) against its context-insensitive version.

To perform the evaluation we implemented four versions of Venable *et al.*'s (VENABLE et al., 2005) method, combining the two context abstractions (ℓ and k) with the two methods for determining contexts (calling-contexts and stack-contexts). The implementation, done on the Java Eclipse workbench, analyzes disassembled binary code. Eclipse is an extensible development environment with a rich set of tools to aid in development (ECLIPSE, 2009). 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. Both analyses were implemented to share most of the code and differ only on the context operations. Thus, there is a high likelihood that the differences in precision and performance observed may be attributed to the analyses, and not as a side-effect of the implementation decisions. All evaluations were performed on an Intel Core2 Duo 2Ghz/3GB Dell Workstation.

Our empirical evaluation shows that as expected a context-sensitive analysis produces more precise results than its context-insensitive counterpart. Quite unexpectedly our evaluation also shows that for certain call structures the context-sensitive analysis is more efficient.

4.1 Comparison of ℓ- and k- Context Analyses

We now present a comparison of the performance of Emami *et al.*'s ℓ-context abstraction of call strings (EMAMI; GHIYA; HENDREN, 1994) against Sharir and Pnueli's k-context abstraction of call strings (SHARIR; PNUELI, 1981). We use the classic method of terminating the analysis when the context reaches k bound, rather than the recent improvements (KHEDKER; KARKARE, 2008) where the call-string construction is terminated using the data flow values.

The evaluation is performed using the following classical programs: *binarySeach*, *bubbleSort*, *factorial*, *fibonacci*, *gcd* and *hanoi*. Besides being classic, the programs also have different recursive structures, an important component for evaluating interprocedural analyses. The program *bubbleSort* has no procedure (other than *main*), hence it is not recursive. Programs *factorial* and *gcd* contain single recursion, and programs *fibonacci* and *hanoi* contain multiple recursion. These programs, all originally written in *C*, were compiled, linked, and then disassembled for analysis. The code corresponding to the *C* functions were then extracted and used for our analysis.

Table 7 presents our measurements from analyzing each of the programs using (a) k-context abstraction ($\llparenthesis^{k}_{call}$) and (b) ℓ-context abstraction ($\llparenthesis^{\ell}_{call}$), both using call-strings. The metrics we use have been borrowed from (KHEDKER; KARKARE, 2008) and are explained at the bottom of the table. The k presented in Table 7 gives the largest value for which the k-context analysis does not run out of space, except for *bubbleSort*, *factorial* and *gcd*. The data shows that the ℓ-context abstraction produces fewer call strings per node (*#CSPN*) and consequently takes less time. The improvement in time can be associated with the recursive structure of the program. For *bubbleSort*, a program with no recursive calls, interprocedural analysis using ℓ-context takes approximately the same time. For *factorial* and *gcd*, programs with single recursion, the ℓ-context based analysis is over 10 times faster than k-context analysis (for the largest k where the analysis terminates). However, for programs with multiple recursion ℓ-context based analysis is 1000 times faster.

The improvement in the approximation for these analyses is harder to quantify. There are instances in which the ℓ-context analysis may produce a value set $[1, \infty]$, while the k-context analysis would produce $[-\infty, \infty]$. We implement the widening operator, rather than the narrowing operator. Therefore, any comparison may not be definitive. However, it can be observed that for all programs the ℓ-context analysis is no worse than the k-context analysis. As expected the k-context analysis resulted in flow of values across non-valid interprocedural paths, after the k bound was reached. This, however, was not the case for the ℓ-context abstraction, which abstracts only the cycles in recursive paths.

Table 7: Empirical measurements on (a) k-context-abstraction and (b) ℓ-context-abstraction.

			k-context-abstraction						ℓ-context-abstraction				
Program	#Inst	#C	k	#CS	MaxL	#CSPN	#InterInst	Time (ms)	#CS	MaxL	#CSPN	#InterInst	Time (ms)
bubbleSort	66	1	1	3	1	1	114	97	3	1	1	114	94
factorial	26	2	21	44	21	43	586	484	5	2	3	72	47
gcd	33	2	21	44	21	43	674	515	5	2	3	81	47
binarySearch	77	3	6	191	6	190	819,366	1.2×10^6	15	3	13	2,072	1,045
fibonacci	34	3	6	191	6	190	165,539	2.5×10^5	11	3	9	353	390
hanoi	39	3	7	383	7	382	583,698	1.3×10^6	11	3	9	325	156

#Inst is the number of instructions
#C is the number of call sites
#CS is the total of call strings
#CSPN is the maximum number of call strings reaching any node
#InterInst denotes the total number of interpreted instructions
MaxL denotes the maximum length of any call strings.

4.2 Improvement in Analysis of Obfuscated Code

In the absence of any accepted gold standard or benchmark for evaluating obfuscated programs, we crafted our own procedure. We performed the analysis using two sets of programs. Programs in the first set were hand-crafted with a certain known obfuscated calling structure. By hand-crafting the programs we were able to control the call-structure and study how the performance changed with changes in the structure. While the extreme case of the call-structures we created are unlikely to occur in real programs, they are nonetheless revealing on how the performance varies with growth of context. The hand-crafted programs were assembled using the Turbo Assembler 5.0. The second set contains W32.Evol.a, a metamorphic virus that employs call obfuscation. This virus has been thorougly analyzed in our lab, and hence we are in a position to evaluate the results of our analysis. While we have thousands of malicious programs in our repository, we have not used them for our analysis because of lack of knowledge of their details and hence our inability to evaluate the results of their analysis.

To make our comparison quantitative we use two metrics: time and size of the sets. Time is measured as the CPU time (in milliseconds) to complete an analysis. The size of the sets is measured in terms of the cumulative size of the value sets for all instructions. The size of the value set at an instruction i for context-insensitive analysis is denoted by $S_{in}(i)$, and that for context-sensitive analysis is denoted by $S_{sen}(i)$. These are computed as follows:

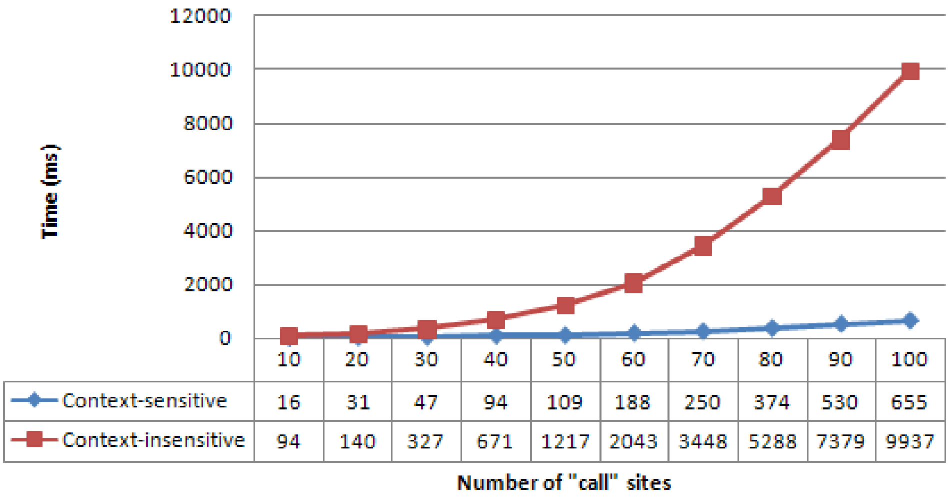
$$S_{in}(i) = \sum_{r} |\mathcal{I}^{\#}_{in}\ i\ r| + \sum_{l} |\mathcal{I}^{\#}_{in}\ i\ l|$$

$$S_{sen}(i) = \sum_{\nu_{asm} \in \hat{\llparenthesis}^{\ell}_{asm}} \Big(\sum_{r} |\mathcal{I}^{\#}_{sen}\ \nu_{asm}\ i\ r| + \sum_{l} |\mathcal{I}^{\#}_{sen}\ \nu_{asm}\ i\ l|\Big)$$

where the function $\mathcal{I}^{\#}_{in} : I \rightarrow R + L \rightarrow ASG + RIC$ and $\mathcal{I}^{\#}_{sen} : \hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC$ are the context-insensitive and context-sensitive abstractions of $\mathcal{I}^{\#} : I \rightarrow R + L \rightarrow \wp(\mathbb{Z})$, respectively.

Each program in the hand-crafted set contains a single procedure that adds two parameters and returns the value. The programs differ in the number of calls to this procedure. We constructed 10 programs, where the program number n has n "calls" to the same procedure. Each "call" passes different pairs of numbers and is implemented using a combination of two *push* instructions and a *ret* instruction. Although all stack contexts in these programs are bounded by four (the number of *push* instructions), this class of programs helps us evaluate the effect of the number of contexts.

Figure 28 plots the time for analyzing the 10 programs. The results show that for this limited class of programs the computational cost of context-sensitive analysis grows linearly with the number of contexts (for the same procedure). In contrast, the cost for context-insensitive analysis grows quadratically. This is expected because Venable *et al.*'s context-insensitive analysis essentially performs intraprocedural analysis on the program. Since the program is obfuscated, its calling structure is unavailable. The analysis, thus, returns the results of a call to every "call"-site, leading to $O(n^2)$ paths for returning values.



	10	20	30	40	50	60	70	80	90	100
◆ Context-sensitive	16	31	47	94	109	188	250	374	530	655
■ Context-insensitive	94	140	327	671	1217	2043	3448	5288	7379	9937

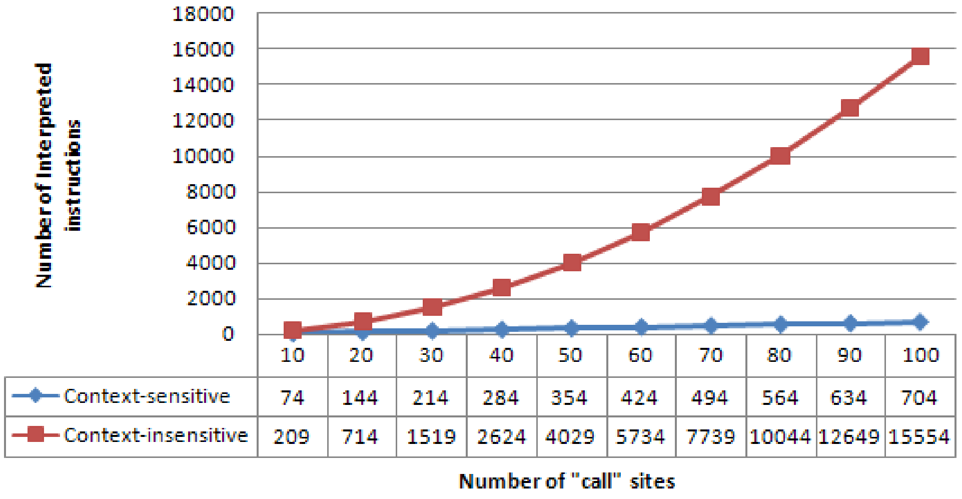
Figure 28: Time evaluation of the set of hand-crafted, obfuscated programs.

Figure 29 shows the difference in the number of interpreted instructions for both analysis, in which we can observe that the number of interpreted instructions in the context-sensitive analysis grows linearly with the growth of contexts; however, it grows quadratically in the context-insensitive analysis. The quadratic growth can be explained due to the analysis being performed on a much larger number of invalid paths.

Figure 30 shows the size of the value sets for all stores in the whole program in the context-sensitive analysis (S_{sen}) and context-insensitive analysis (S_{in}). We can observe that S_{sen} grows linearly with the growth of contexts; however, S_{in} grows quadratically. The quadratic growth can be explained due to the analysis being performed on a much larger number of invalid paths.

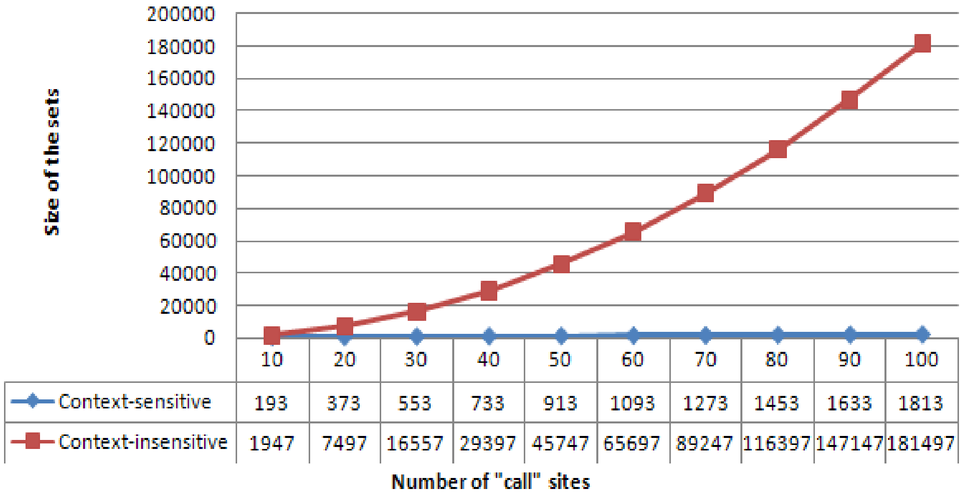
To quantify the improvement resulting from analyzing W32.Evol.a using our context-sensitive analysis over Venable *et al.*'s context-insensitive analysis we compute the difference in the size of the value sets resulting from the two analyses for each instruction. Since the sizes resulting from context-insensitive analysis are always higher, we compute the difference as $S_{in}(i) - S_{sen}(i)$, for instruction i.

Figure 31 presents a histogram that shows the number of instructions where the



	10	20	30	40	50	60	70	80	90	100
◆ Context-sensitive	74	144	214	284	354	424	494	564	634	704
■ Context-insensitive	209	714	1519	2624	4029	5734	7739	10044	12649	15554

Figure 29: Comparison of number of interpreted instructions between context-sensitive and context-insensitive analyses.



	10	20	30	40	50	60	70	80	90	100
◆ Context-sensitive	193	373	553	733	913	1093	1273	1453	1633	1813
■ Context-insensitive	1947	7497	16557	29397	45747	65697	89247	116397	147147	181497

Figure 30: Evaluation of the size of the value sets between context-sensitive and context-insensitive analyses.

context-insensitive analysis gives larger sets (for various intervals of differences). The data shows an improvement in precision with approximately 25% (25 of 98) of the interpreted instructions of W32.Evol.a virus producing answers with better precision. The time for analyzing W32.Evol.a virus was 300 ms and 1100 ms for context-sensitive and context-insensitive analysis, respectively. Thus, our context-sensitive analysis is more efficient and more precise than Venable *et al.*'s context-insensitive analysis.

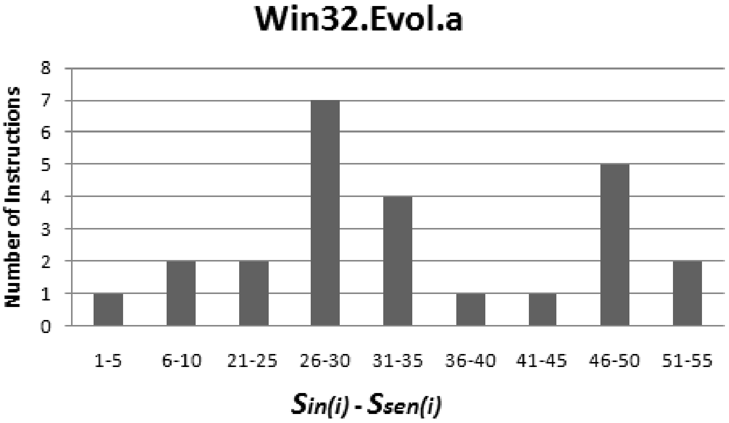


Figure 31: Histogram of approximations for Win32.Evol.a.

4.3 Discussion

We implement a context-sensitive variant of Venable *et al.*'s analysis that combines the VSA and ASG (abstract stack graph) domains (VENABLE et al., 2005). Empirical evaluation shows that context-sensitive analysis using ℓ-context leads to several order of magnitude improvement in the running time, as well as improvement in precision. The results also show that context-sensitive analysis of obfuscated code produces more precise and effective results than its context-insensitive counterpart. In the next chapter, we summarize the major contributions of this dissertation and briefly describe future works that we would like to explore.

5 *Conclusions and further work*

This chapter synthesizes the research outcomes of this dissertation and provides some directions for further work.

5.1 Research Outcomes

The following contributions have been made in this dissertation.

A model of contexts based on the stack was proposed. This model is intended to be more general than existing ones, in particular the call string approach of Sharir and Pnueli (SHARIR; PNUELI, 1981). The notion of stack-context is a generalization of calling-context, where instead of distinguishing contexts using *call* instructions, it uses stack configuration. Context-sensitive interprocedural analyses when guided by a call-graph (the basis to extract calling-contexts) are limited to only 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 analyses on obfuscated, optimized, or hand-written code. As a result, malware forensic tools based on such analyses can easily be thwarted. Such binaries are often crafted to break existing methods of analysis. For instance, the IDA Pro disassembler identifies the procedures in a binary by analyzing its *call* instructions. Any analysis based on such a disassembler would fail if the binary does not use the *call* instruction to make a procedure call. Obfuscations that defeat analyzers are commonly used by authors of malware. They are also used by authors of legitimite programs to protect their intellectual property. The use of stack-contexts enables performing context-sensitive analyses of binaries when the calling-contexts cannot be distinguished, such as due to obfuscations.

Our method of context-sensitive analysis does not rely on finding procedure boundaries and determining procedure calls. Instead, it defines a context based on the state of the stack. Thus, any operation that pushes data on the stack creates a context. Conversely, any operation that removes data from the stack removes a context. As a consequence, a context in our method does not imply transfer of control (as in the case of procedure call and return). The problem of determining transfer of control, also an important problem for obfuscated binaries, is solved separately by using Balakrishnan and Reps' Value-Set Analysis (VSA) (BALAKRISHNAN; REPS, 2004).

We demonstrate how an abstract stack graph (ASG) may be used as a replacement of the call-graph (CG) 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 CG and ASG for the same program is isomorphic when the program follows the standard compilation model. 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 adaptation is simple enough to directly impact interprocedural analysis algorithms based on call-graph (BALAKRISHNAN, 2007),(MATTHEW et al., 2005).

A model of abstracting context strings was introduced. This model adapts for use with stack context prior work on performing context-sensitive analysis using calling-contexts. The notion of call-strings has in the past been described in terms of valid paths of an ICFG (SHARIR; PNUELI, 1981). We generalize the concept using abstract interpretation and define contexts using trace semantics. Unlike Sharir and Pnueli's formulation this generalization does not require transfer of control, an intrinsic part of semantics of procedure call and return. Similarly, a ℓ-context abstraction is derived that generalizes for use with stack-context Emami *et al.*'s strategy of abstracting calling-contexts by reducing cycles due to recursion (EMAMI; GHIYA; HENDREN, 1994) (which we term as ℓ-context), thus leading the way to the use of BDDs for making the analysis scalable (ZHU, 2005),(WHALEY; LAM, 2004).

Improvement on the DOC algorithm. DOC (Detector of Obfuscated Calls) was proposed by Venable *et al.* (VENABLE et al., 2005). It is a static analysis suite that detects obfuscations in executables, particularly procedure call and call-return obfuscations. It uses abstract interpretation to find instances in which explicit call or call-return instructions are not used. However, DOC implementation has two limitations. First, it does not perform interprocedural analysis (it treats the program as one big function). As a result, it performs what amounts to be intraprocedural analysis on the entire program. The resulting analysis is expensive and leads to very significant over approximation, thus confining its application to small programs.

We improved DOC algorithm by adding context-sensitivity using stack-contexts. The resulting analysis is shown to be sound. To perform the evaluation, we implemented four versions of Venable's algorithm using the combination of the two context abstractions (ℓ and k) with the two methods for determining contexts (calling-contexts and stack-contexts). Empirical results show that the context-sensitive algorithms produced better (smaller) approximations in less time than the context-insensitive algorithm. This was counter-intuitive because the expectation due to maintaining data for each context supposedly consumes more time. The loss due to increased data appears to be overcome by the decrease in the number of instructions interpreted. The context-sensitive algorithm performs fewer approximation operations, implying that it results in answers with better precision.

The results regarding implications of obfuscated code in the AV detectors were published in the Integrated Seminar of Software and Hardware (SEMISH 2007) (BOCCARDO; MANACERO JÚNIOR.; FALAVINHA JÚNIOR., 2007). The use of the Abstract Stack Graph (ASG) to adapt the call-graph (CG) to analyze obfuscated binaries was published in the Brazilian Symposium on Information and Computer System Security (SBseg 2009) (BOCCARDO et al., 2009). The proposed formalism using abstract interpretation to generalize prior work on performing context-sensitive analysis using calling-contexts and to perform the analysis of obfuscated assembly programs was submitted to the Partial Evaluation and Program Manipulation (PEPM 2010).

5.2 Directions for Further Work

Directions for further work are:

Completeness of the abstract interpreters. As future work we may consider the possibility of discussing the completeness, in the abstract interpretation sense, of the abstractions of this dissertation (or their completeness refinement). It might be interesting to verify if these abstractions are complete (or if they are complete for some properties of the semantics, for example for the system call behavior), *i.e.*, if there is no loss of precision accumulated in the abstract computation. For example, abstracting the result of the concrete-trace semantics leads to the same result obtained by the derived context-sensitive analyzer. Completeness could give some formal insight on the precision of the abstract analysis.

Improvements on DOC algorithm. The following improvements can be done in the DOC algorithm in order to make it more efficient and useful for other applications.

- Memory Support - The current implementation provides support for only the register and stack. Extending DOC to have full memory support will not only help improve the results of call obfuscation detection, but may also help to make the project useful for other applications.
- Built-in Disassembler - The first phase of the DOC algorithm consists of disassembling the executable. However, this process is manual, and it needs to be made in a disassembler. The output of the disassembler is then loaded into the DOC to perform its analysis. Constructing a built-in disassembler in the DOC improves its user interface. It is also possible to use context-sensitive value-set analysis to improve disassembly capabilities.
- Unpacking capability - The use of packers has become more popular among malware writers. A packed piece of malware has a better chance of remaining undetected for a long time, as well as spreading faster due to its smaller size. Packing an existing piece of malware is also by far the easiest way to create a ‘new’ one. Of the new incoming samples we see more than 50% are produced simply by repacking existing malware (STEPAN, 2006) using different packers. Therefore, adding unpacking capability to the DOC will be useful for malware forensics of packed malwares.

References

AHO, A. V. et al. **Compilers**: principles, techniques, and tools. 2.ed. Upper Saddle River: Addison Wesley, 2006.

AMME, W. et al. Data dependence analysis of assembly code. **International Journal of Parallel Programming**, Dordrecht, v. 28, n. 5, p. 431–467, 2000.

BACKES, W. **Programmanalyse des XRTL Zwischencodes (In German.)**. 2004. Thesis (PhD) — Universitaet de Saarlandes, 2004.

BALAKRISHNAN, G. **WYSINWYX**: What You See Is Not What You eXecute. 2007. Thesis (PhD) — Computer Science Department., University of Wisconsin, Madison, WI, 2007.

BALAKRISHNAN, G.; REPS, T. Analyzing memory accesses in x86 executables. In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION (CC), 2004, Barcelona. **Proceedings of the...** Barcelona, Spain: Springer-Verlag, 2004. p. 5–23.

BALAKRISHNAN, G.; REPS, T. W. Divine: discovering variables in executables. In: VERIFICATION, MODEL CHECKING, AND ABSTRACT INTERPRETATION (VMCAI), 8, 2007, Nice. **Proceedings of the...** Nice: Springer Berlin / Heidelberg, 2007. p. 1–28.

BALL, T.; MILLSTEIN, T.; RAJAMANI, S. K. Polymorphic predicate abstraction. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, New York, NY, USA, v. 27, n. 2, p. 314–343, 2005. ISSN 0164-0925.

BERGERON, J. et al. Detection of malicious code in COTS software: a short survey. In: INTERNATIONAL SOFTWARE ASSURANCE CERTIFICATION CONFERENCE (ISACC), 1, 1999, Washington. **Proceedings of the...** Washington, DC: IEEE Press., 1999.

BERGERON, J. et al. Static detection of malicious code in executable programs. In: INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING FOR INFORMATION SECURITY (SREIS), 2001, Indianapolis. **Electronic proceedings of the...** Indianapolis: Springer Verlag, 2001. p. 1–8.

BERGERON, J. et al. Static analysis of binary code to isolate malicious behaviors. In: WORKSHOP ON ENABLING TECHNOLOGIES ON INFRASTRUCTURE FOR COLLABORATIVE ENTERPRISES (WETICE), 8, 1999, Washington. **Proceedings of the...** Washington: IEEE Computer Society, 1999. p. 184–189.

BOCCARDO, D. R. et al. Adapting call-string approach for x86 obfuscated binaries. In: BRAZILIAN SYMPOSIUM ON INFORMATION AND COMPUTER SYSTEM SECURITY (SBSeg), 9, 2009, Campinas. **Proceedings of the...** Campinas: SBC, 2009.

BOCCARDO, D. R.; MANACERO JÚNIOR., A.; FALAVINHA JÚNIOR., J. N. Implicações da ofuscação de código no desenvolvimento de detectores de código malicioso. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE (SEMISH), 34, 2007, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 2007. p. 2277–2291.

CHRISTODORESCU, M.; JHA, S. Static analysis of executables to detect malicious patterns. In: USENIX SECURITY SYMPOSIUM, 12, 2003, Washington. **Proceedings of the...** Washington: USENIX Association, 2003. p. 169–186.

CIFUENTES, C.; FRABOULET, A. **Interprocedural data flow recovery of high-level language code from assembly**. Queensland. Tech Report. University of Queensland, 1997.

CIFUENTES, C.; FRABOULET, A. Intraprocedural static slicing of binary executables. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), 13, 1997, Bari. **Proceedings of the...** Bari: IEEE Computer Society, 1997. p. 188–195.

CIFUENTES, C.; SIMON, D.; FRABOULET, A. Assembly to high-level language translation. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), 14, 1998, Bethesda. **Proceedings of the...** Bethesda: IEEE Press, 1998. p. 228–237.

COLLBERG, C.; THOMBORSON, C.; LOW, D. **A taxonomy of obfuscating transformations**. Auckland. Tech Report. University of Auckland, 1997.

COLLBERG, C. S.; THOMBORSON, C. Watermarking, tamper-proofing, and obfuscation - tools for software protection. **IEEE Transactions on Software Engineering**, New York, v. 28, n. 8, p. 735–746, 2002.

COUSOT, P. Abstract interpretation. **ACM Computing Surveys**, New York, v. 28, n. 2, p. 324–328, 1996.

COUSOT, P.; COUSOT, R. Static determination of dynamic properties of programs. In: INTERNATIONAL SYMPOSIUM ON PROGRAMMING, 2, 1976, Paris. **Proceedings of the...** Paris: [s.n.], 1976. p. 106–130.

COUSOT, P.; COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 4, 1977, Los Angeles. **Proceedings of the...** Los Angeles: ACM, 1977. p. 238–252.

COUSOT, P.; COUSOT, R. Systematic design of program analysis frameworks by abstract interpretation. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 6, 1979, San Antonio. **Proceedings of the...** San Antonio: ACM, 1979. p. 269–282.

COUSOT, P.; COUSOT, R. Modular static program analysis. In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION (CC), 11, 2002, Grenoble. **Proceedings of the...** Grenoble: Springer-Verlag, 2002. p. 159–178.

COUSOT, P.; COUSOT, R. Basic concepts of abstract interpretation. In: IFIP CONGRESS TOPICAL SESSIONS, 2004, Toulouse. **Proceedings of the...** Toulouse: Kluwer Academic Publishers, 2004. p. 359–366.

DALLA PREDA, M. et al. A semantics-based approach to malware detection. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 34, 2007, Nice, France. **Proceedings of the...** New York: ACM, 2007. p. 377–388.

DAVEY, B. A.; PRIESTLEY, H. A. **Introduction to lattices and order**. Cambridge: Cambridge Press, 1990.

DEBRAY, S. K.; MUTH, R.; WEIPPERT, M. Alias analysis of executable code. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 25, 1998, San Diego. **Proceedings of the...** San Diego: ACM, 1998. p. 12–24.

ECLIPSE. Eclipse Foundation. Data Rescue, Liege, Belgium, 2009. Available from Internet: <**http://www.eclipse.org**. Last accessed July 2009>.

EMAMI, M.; GHIYA, R.; HENDREN, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ***Sigplan ACM***, New York, v. 29, n. 6, p. 242–256, 1994.

GIERZ, G. et al. **A compendium on continuous lattices**. Berlin: Springer-Verlag, 1980.

GOODWIN, D. W. Interprocedural dataflow analysis in an executable optimizer. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), 1997, Las Vegas. **Proceedings of the...** New York: ACM, 1997. p. 122–133.

GRÄTZER, G. **General lattice theory**. Basel: Birkhäuser Verlag, 1978.

GULWANI, S.; TIWARI, A. Computing procedure summaries for interprocedural analysis. In: EUROPEAN SYMPOSIUM ON PROGRAMMING (ESOP), 16, 2007, Braga. **Proceedings of the...** Braga: Springer, 2007. v. 4421, p. 253–267.

IDAPRO. Ida Pro - Disassembler. 2009. Data Rescue, Liege, Belgium. Available from Internet: <**http://www.datarescue.com**. Last accessed July 2009>.

KARKARE, B.; KHEDKER, U. P. An improved bound for call strings based interprocedural analysis of bit vector frameworks. **ACM Transactions on Programming Language Systems**, New York, v. 29, n. 6, 2007.

KHEDKER, U.; KARKARE, B. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. **Lecture Notes in Computer Science**, Germany, v. 4959, p. 213–228, 2008.

KINDER, J.; VEITH, H.; ZULEGER, F. An abstract interpretation-based framework for control flow reconstruction from binaries. In: VERIFICATION, MODEL CHECKING, AND ABSTRACT INTERPRETATION (VMCAI), 10, 2009, Savannah. **Proceedings of the...** Savannah: Springer-Verlag, 2009. p. 214–228.

KUMAR, E. U.; VENABLE, M. DOC - Detector of Obfuscated Calls. 2007. **http://sourceforge.net/projects/obfuscation**. Last accessed July 2009.

LAKHOTIA, A.; KUMAR, E. U. Abstack stack graph to detect obfuscated calls in binaries. In: INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 4, 2004, Chicago. **Proceedings of the...** Chicago: IEEE Computer Society, 2004. p. 17–26.

LAKHOTIA, A.; KUMAR, E. U.; VENABLE, M. A method for detecting obfuscated calls in malicious binaries. **IEEE Transactions on Software Engineering**, Piscataway, v. 31, n. 11, p. 955–968, 2005.

LAKHOTIA, A.; SINGH, P. K. **Challenges in getting 'formal' with viruses**. Virus Bulletin, Virus Bulletin Ltd., p. 15–19, September 2003.

LAL, A.; REPS, T. Improving pushdown system model checking. In: COMPUTER-AIDED VERIFICATION, 18, 2006, Seattle. **Proceedings of the...** Seattle: Springer-Verlag, 2006.

LARUS, J. R.; SCHNARR, E. Eel: Machine-independent executable editing. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), 1995, La Jolla. **Proceedings of the...** La Jolla: ACM, 1995. p. 291–300.

LHOTÁK, O.; HENDREN, L. Context-sensitive points-to analysis: Is it worth it? In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION (CC), 15, 2006, Vienna. **Proceedings of the...** Vienna: Springer, 2006. v. 3923, p. 47–64.

LINN, C.; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: COMPUTER AND COMMUNICATIONS SECURITY (CCS), 10, 2003, Washington. **Proceedings of the...** Washington: ACM, 2003. p. 290–299.

MATTHEW, B. G. et al. Practical and accurate low-level pointer analysis. In: SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO), 3, 2005, San Jose. **Proceedings of the...** San Jose: IEEE Computer Society, 2005. p. 291–302.

MÜLLER-OLM, M.; SEIDL, H. Precise interprocedural analysis through linear algebra. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 31, 2004, Venice. **Proceedings of the...** Venice: ACM, 2004. p. 330–341.

MYCROFT, A. Type-based decompilation. In: EUROPEAN SYMPOSIUM ON PROGRAMMING (ESOP), 8, 1999. Amsterdam. **Proceedings of the...** Amsterdam: Springer-Verlag, 1999. p. 208–223.

NIELSON, F.; NIELSON, H. R.; HANKIN, C. **Principles of program analysis**. Secaucus: Springer-Verlag, 1999.

OLLYDBG. OllyDbg debugger. Oleh Yuschuk, 2009. Available from Internet: <**http://www.ollydbg.de/**. Last accessed July 2009>.

REPS, T.; BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION (CC), 2008, Budapest. **Proceedings of the...** Budapest: Springer Berlin / Heidelberg, 2008. p. 16–35.

REPS, T.; BALAKRISHNAN, G.; LIM, J. Intermediate-representation recovery from low-level code. In: WORKSHOP ON PARTIAL EVALUATION AND PROGRAM MANIPULATION (PEPM), 2006, Charleston. **Proceedings of the...** Charleston: ACM, 2006. p. 100–111.

REPS, T.; HORWITZ, S.; SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 22, 1995, San Francisco. **Proceedings of the...** San Francisco: ACM, 1995. p. 49–61.

REPS, T. et al. Weighted pushdown systems and their application to interprocedural dataflow analysis. **Science of Computer Programming**, Amsterdam, v. 58, n. 1-2, p. 206–263, 2005.

SAGIV, M.; REPS, T.; HORWITZ, S. Precise interprocedural dataflow analysis with applications to constant propagation. In: INTERNATIONAL JOINT CONFERENCE CAAP/FASE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, 6, 1995, London. **Proceedings of the...** London: Springer-Verlag, 1995. p. 651–665.

SCHWARZ, B.; DEBRAY, S.; ANDREWS, G. Disassembly of executable code revisited. In: WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), 9, 2002, Richmond. **Proceedings of the...** Richmond: IEEE Computer Society, 2002.

SCHWARZ, B.; DEBRAY, S. K.; ANDREWS, G. R. PLTO: A link-time optimizer for the Intel IA-32 architecture. In: WORKSHOP ON BINARY TRANSLATION (WBT), 2001, Barcelona. **Proceedings of the...** Barcelona: [s.n.], 2001.

SHARIR, M.; PNUELI, A. **Two approaches to interprocedural data flow analysis**. Program Flow Analysis: theory and applications. Englewood Cliffs: Prentice-Hall, 1981.

SRIVASTAVA, A.; WALL, D. A practical system for intermodule code optimization at linktime. **Journal of Programming Languages**, New York, v. 1, n. 1, p. 1–18, 1993.

STEPAN, A. Improving proactive detection of packed malware. In: VIRUS BULLETIN CONFERENCE, 16, 2006, Montreal. **Proceedings of the...** London: Virus Bulletin Ltd., 2006.

SYMANTEC. Understanding Heuristics. 1997. Available from Internet: <**http://www.symantec.com/avcenter/reference/heuristc.pdf**. Last accessed July 2009>.

SZÖR, P.; FERRIE, P. Hunting for metamorphic. In: VIRUS BULLETIN CONFERENCE, 11, 2001, Prague. **Proceedings of the...** Prague: Virus Bulletin, 2001. p. 123–144.

TARSKI, A. A lattice-theoretical fixpoint theorem and its applications. **Pacific Journal of Mathematics**, Berkeley, v. 5, n. 2, p. 285–309, 1955.

THOMPSON, K. Reflections on trusting trust. **Communications of the ACM**, New York, v. 27, n. 8, p. 761–763, 1984.

VENABLE, M. et al. Analyzing memory accesses in obfuscated x86 executables. In: DETECTION OF INTRUSIONS AND MALWARE & VULNERABILITY ASSESSMENT (DIMVA), 2005, Vienna. **Proceedings of the...** Vienna: Springer Berlin / Heidelberg, 2005. p. 1–18.

VENKITARAMAN, R.; GUPTA, G. Static program analysis of embedded executable assembly code. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS (CASES), 2004, Washington. **Proceedings of the...** New York: ACM, 2004. p. 157–166.

VINCIGUERRA, L. et al. An experimentation framework for evaluating disassembly and decompilation tools for c++ and java. In: WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), 10, 2003, Victoria. **Proceedings of the...** Washington: IEEE Computer Society, 2003. p. 14.

WALENSTEIN, A. et al. Normalizing metamorphic malware using term-rewriting. In: INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 6, 2006, Philadelphia. **Proceedings of the...** Philadelphia: IEEE Computer Society, 2006.

WHALEY, J.; LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), 2004, Washington. **Proceedings of the...** New York: ACM, 2004. p. 131–144.

WILSON, R. P.; LAM, M. S. Efficient context-sensitive pointer analysis for c programs. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), 1995, La Jolla. **Proceedings of the...** New York: ACM, 1995. p. 1–12.

WROBLEWSKI, G. General method of program code obfuscation. In: SOFTWARE ENGINEERING RESEARCH AND PRACTICE (SERP), 2002, Las Vegas. **Proceedings of the...** Las Vegas: [s.n.], 2002.

XIE, Y.; AIKEN, A. Scalable error detection using boolean satisfiability. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL), 32, 2005, Long Beach. **Proceedings of the...** Long Beach: ACM, 2005. p. 351–363.

ZHU, J. Towards scalable flow and context sensitive pointer analysis. In: DESIGN AUTOMATION CONFERENCE (DAC), 42, 2005, Anaheim. **Proceedings of the...** New York: ACM, 2005. p. 831–836.

ZHU, J.; CALMAN, S. Symbolic pointer analysis revisited. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), 2004, Washington. **Proceedings of the...** New York: ACM, 2004. p. 145–157.