Context-Sensitive Analysis of Obfuscated x86 Executables [*]

Arun Lakhotia[†] Davidson R. Boccardo[‡] Anshuman Singh[†] Aleardo Manacero Jr.[‡]

[†]Center for Advanced Computer Studies
University of Louisiana at Lafayette, LA, USA
{arun, axs6222}@louisiana.edu

[‡]Electrical Engineering Dept.
Paulista State University (UNESP), Brazil
drb3065@louisiana.edu, aleardo@ibilce.unesp.br

Abstract

A method for context-sensitive analysis of binaries that may have obfuscated procedure call and return operations is presented. Such binaries may use operators to directly manipulate stack instead of using native *call* and *ret* instructions to achieve equivalent behavior. Since definition of context-sensitivity and algorithms for context-sensitive analysis have 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 of changes in 'stack'-context. An abstract interpretation based framework is developed to reason about stack-contexts and to derive analogues of call-strings based methods for the context-sensitive analysis using stack-context. 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.

Categories and Subject Descriptors K.6.5 [*Security and Protection*]: Invasive software (e.g., viruses, worms, Trojan horses); D.2.0 [*Software Engineering–General*]: Protection mechanisms; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Languages, Security, Theory, Verification

Keywords Analysis of binaries, Context-sensitive analysis, Obfuscation, Deobfuscation

1. Introduction

Recent years have seen an increase in research activity in the area of binary analysis [1, 2, 8, 10, 13, 16, 22, 24, 28]. For third-party programs where the source code is not available to the analyst, analysis for malicious (hidden) behavior can reliably be performed only on binaries. Even when the code is available, analyzing the binary is the only true way to detect hidden capabilities, as demonstrated by Thompson in his Turing Award Lecture [27]. Lest Thompson's paper be considered theoretical, a variation of his ideas has been put into practice by the malware W32.Induc.A [20].

Current methods of analyzing binaries are modeled on methods for analysis of source code. A program is decomposed into a collection of procedures, and the program is analyzed using classical interprocedural analysis [25]. 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 as 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.' However, 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.

We consider a binary to use *call obfuscation* if it does not follow any particular convention for the layout of the procedure code in memory or if it does not use *call* or *ret* instructions to make procedure calls. 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 [3, 19]. However, the code may also be obfuscated to hide malicious intent [2, 15]. Most malwares today use a variety of obfuscations to deter its disassembly, its reverse engineering, or its analysis.

Figure 1 presents an example to help visualize the problem space. Consider the sample program of Figure 1(a). The program is simplified only to highlight its call and return structure. Figure 1(c) shows an obfuscated version of this program. It is generated using a naïve obfuscation: replace 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.

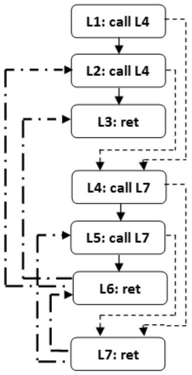
Figure 1(b) shows the control flow graph (CFG) of the sample program of Figure 1(a) created by assuming that the target of a *call* instruction represents the entry point of a procedure and a *ret* instruction returns from call to the closest preceding entry point.

[*] This research was supported in part by funds from the Louisiana Governorś Information Technology Initiative, Air Force Office of Scientific Research grant AF9550-09-1-0715, and Coordination for the Improvement of Higher Education - (CAPES - Brazil)

PEPM'10 January 18-19, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-727-1/10/01. . . $10.00

```
L1:  call L4
L2:  call L4
L3:  ret
L4:  call L7
L5:  call L7
L6:  ret
L7:  ret
```

(a) Sample code.



(b) CFG.

```
L1:  push L2
L1a: push L4
L2b: ret
L2:  push L3
L2a: push L4
L2b: ret
L3:  ret
L4:  push L5
L4a: push L7
L4b: ret
L5:  push L6
L5a: push L7
L5b: ret
L6:  ret
L7:  ret
```

(c) Obfuscated version.

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

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 [25] via a mismatched return edge.

Since the program of Figure 1(c) does not have any *call* instruction, it does not provide any clues for finding procedure entry points. Current technologies may infer that this program has only one procedure consisting of the entire code [11]. Furthermore, most works on analysis of binaries will assume that each *ret* instruction returns to the caller of this single procedure, thus generating an incorrect CFG. As a result, any resulting analysis based on this CFG will also be incorrect.

The obfuscation shown in Figure 1(c) is naïve and presented to demonstrate the concept. More obfuscations, although still trivial, may be performed by shuffling the two *push* instructions among other code. More complex obfuscations may be achieved by not using *push* and *ret* instructions; instead one may use move, increment, and decrement operations directly on the stack pointer to perform equivalent behavior [16].

This paper presents a method for performing context-sensitive analyses of binaries that obfuscate procedure calls and returns, such as the program in Figure 1(c). Unlike current methods for performing context-sensitive interprocedural analysis of binaries, our method does not require the use of explicit *call* and *ret* instructions. Our method depends only on the knowledge of how the stack pointer and the instruction pointer are represented, the direction of stack growth, and the static identification of operators manipulating the stack pointer. Our method requires only that the register or memory location used to represent a stack pointer must be known statically prior to the analysis. 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. Our analysis assumes the knowledge of this convention.

The main contributions of this paper 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 that uses call obfuscation.
- It presents a systematic development of generic context-sensitive analysis using Galois connection based abstractions of a traditional trace-based semantics. The context-abstractions it derives are generic in that they dependent only on the LIFO nature of creation and deletion of contexts. These abstractions enable derivation of stack-based context-sensitive analysis, which, unlike calling-context based analysis of prior work, do not depend upon transfer of control semantics.
- It systematically derives generic versions of Sharir and Pnueli's k-suffix call-strings abstractions [25] and Emami *et al.*'s strategy of abstracting calling-contexts (referred to in this paper as ℓ-contexts [9]. Prior work on these abstractions was dependent upon the control flow semantics of *call* and *ret* instructions.
- It proposes a general methodology for deriving sound context-sensitive analysis from context-insensitive one. As an application, a context-sensitive version of Venable *et al.*'s algorithm [28] is derived. 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 more precise results.

Section 2 discusses the related works. Section 3 provides background in domain theory and abstract interpretation. Section 4 introduces context-trace semantics, a trace semantics in which context is made explicit. Section 5 presents generalization of Sharir and Pnueli's [25] and Emami *et al.*'s context abstractions. Section 6 presents our algorithm derives context-sensitive version of Venable *et al.*'s algorithm. Section 7 presents empirical evaluation of the method presented and is followed by our concluding remarks.

2. Related Works

Prior research related to this paper may be broadly grouped into the following categories: interprocedural analysis (in general), interprocedural analysis of binary programs, and analysis of malicious/obfuscated programs.

Precise and efficient context-sensitive interprocedural data-flow analysis of high-level languages has been an active area of research. The general strategies fall within the two approaches proposed by Sharir and Pnueli [25], namely the call-string approach or the procedure summaries approach. A good summary of these works may be found in [12].

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 the procedures and their boundaries, thus violating a pre-requisite. Reps *et al.*'s weighted pushdown system based interprocedural analysis, which also computes function summaries [23], 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 [17, 23]. However, a pushdown system implicitly depends upon the the transfer of control semantics of call and return instructions, and thus may not be generalizable to programs with arbirtrary stack operations.

Might and Shivers [21] framestrings for λ-based languages is similar to our stack-string abstraction in that the context is defined in terms of push and pop stack operations. Their work also includes modeling environments, with the intent of enabling certain inlining optimizations. Use of their environment theory in our context would be a valuable direction for future work.

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. We generalize context abstraction such that it does not depend on the semantics of procedure invocation. As is done for context-sensitive points-to analysis, the call-graph used for the call-string approach may be computed on the fly [9, 31–33]. Determining transfer of control based on contents of memory or register is analogous to computing the points-to relation for higher-level 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 [26] 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 [1]. Goodwin uses the procedure summary approach to interprocedural analysis to aid link-time optimization [10]. Balakrishnan [1] 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 [11]. 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) [1].

Like us, Kinder *et al.* have developed abstract interpretation framework for contructing control flow of binaries containing indirect transfer of control [13]. Their analysis, unlike ours, is context-insensitive since it does not model contexts of any kind. It uses constant propagation to determine values in memory and register, and thus is likely to generate greater over-approximations than those resulting from our use of VSA [1].

There has been significant work in obfuscation of programs with the intent to thwart static analysis [3, 19]. The obfuscation techniques are targeted at defeating specific phases in the analysis of a binary [15]. On the other hand a metamorphic malware, a malware that transforms its own code as it propagates, may use procedure call obfuscations simply to help its own transformation algorithm, as is the case with the Win32.Evol virus, a metamorphic virus [30]. It has the side-effect of defeating any interprocedural analysis that depends on a traditional compiler model [15].

There has also been efforts in the use of other semantics based methods for detecting malware [2, 7]. Term-rewriting has been proposed to normalize variants of a metamorphic malware [29]. None of these works specifically addresses analysis of obfuscated programs that do not conform to the standard compilation model. The foundations of the approach presented in this paper come from previous work of our research group in analyzing programs with obfuscated calls [16, 28].

3. Preliminaries

3.1 Domain Theory

A binary relation $\sqsubseteq_C$: $C \times C$ is a *partial order* upon a set C iff $\sqsubseteq_C$ is reflexive, transitive and antisymmetric. For a set C partially ordered by $\sqsubseteq_C$ and a subset X of C, $\bigsqcup X$ denotes the element of C (if it exists) that satisfies the following conditions: (i) $\forall x \in X, x \sqsubseteq_C \bigsqcup X$; and (ii) $\forall y \in C, \forall x \in X, x \sqsubseteq_C y \Rightarrow \bigsqcup X \sqsubseteq_C y$. The element $\bigsqcup X$ is called the least upper bound (*lub*) of X. Its dual, the greatest lower bound (*glb*) of X, is denoted by $\bigsqcap X$. When operating on a set of two elements, the operations are represented by the binary operators $\sqcup$ and $\sqcap$, respectively. A partially ordered set $\langle C, \sqsubseteq_C \rangle$ is a *lattice* iff $\forall x, y \in C$, both $x \sqcup y$ and $x \sqcap y$ exist. A partially ordered set $\langle C, \sqsubseteq_C \rangle$ is a *complete lattice* iff for all subsets X of C, both $\bigsqcup X$ and $\bigsqcap X$ exist. For any set X, $\langle \wp(X)$, $\subseteq \rangle$ is a lattice under the usual subset ordering $\subseteq$.

Let X^* denote the Kleene closure of the set X, *i.e.*, the set of finite sequences over X. Let $\epsilon \in X^*$ denote the sequence of length 0. Let $(x\ i)$ denote the i^{th} element of the sequence $x \in X^*$. Let '.': $X \times X^* \rightarrow X^*$ be the *cons operator*, that inserts an element at the head of a sequence, 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:

$$\forall x_1, x_2 \in X; s, s_1, s_2 \in X^*$$

$$\epsilon \sqsubseteq_{X^*} s$$

$$x_1.s_1 \sqsubseteq_{X^*} x_2.s_2 \Leftrightarrow x_1 \sqsubseteq_X x_2 \wedge s_1 \sqsubseteq_{X^*} s_2$$

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 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.

Two complete lattices, C and A, form a *Galois connection* iff there exists an adjunction between C and A, *i.e.*, $\exists \alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ such that $\forall a \in A, c \in C : \alpha\ c \sqsubseteq_A a \Leftrightarrow c \sqsubseteq_C \gamma\ a$. A Galois connection is denoted by (C, α, γ, A) where $\alpha(\gamma)$ 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 uniquely determines 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).

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.

3.2 Abstract interpretation

Abstract interpretation is a unified framework for designing approximate semantics of programs [5, 6]. It allows the systematic derivation of data flow analyses and provides methods to prove their correctness and termination.

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 by their precision.

Following [4], a program may be formalized as a graph or 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 fixpoint 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 fixpoint (*lfp*) of $\mathcal{F}$ is as follows:

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

Let $(\wp(\Sigma^*), \alpha, \gamma, Abs)$ be a Galois connection, then $\mathcal{F}^\# : Abs \rightarrow Abs$ is sound *w.r.t* $\mathcal{F}$ when $\mathcal{F}^\#$ satisfies the above mentioned requirements (in Sec. 3.1). It can be shown that $\mathcal{F}^\#$ reaches a fixpoint by the well known *fixpoint transfer* theorem [6]. The precision and cost of the approximated fixpoint is related with the choice of the *widening* operator [5].

The derivation of the 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 $(\wp(I \times Store))^*$.
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 by $I \rightarrow \wp(Store)$.
3. Finally, a Galois connection between $\wp(Store)$ and $Abstore$ completes the analysis.

4. Context-trace semantics

Context-sensitivity is presented in the literature classically in terms of paths of an ICFG, a graph that encodes the transfer of control component of semantics of instructions. An ICFG consists of CFGs for individual procedures, and edges between these CFGs represent interprocedural control flow, typically represented 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 such a path to be valid the *call* and the *ret* edges in the path should be paired, meeting certain constraints, the details of which may be found elsewhere in the literature [12].

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 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.

$$\pi : \Sigma \rightarrow \llparenthesis^* \rightarrow \llparenthesis^*$$

$$\pi\ s\ \nu \triangleq \begin{cases} i.\nu & \text{if } i \in \llparenthesis \\ (rest\ \nu) & \text{if } i \in \rrparenthesis \end{cases}$$

where $i = s \downarrow 1$. 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:

$$\Pi : \Sigma^* \rightarrow \llparenthesis^*$$

$$\Pi\ \sigma \triangleq (\Pi'\ \sigma\ \epsilon)$$

$$\Pi' : \Sigma^* \rightarrow \llparenthesis^* \rightarrow \llparenthesis^*$$

$$\Pi'\ \epsilon\ \nu \triangleq \nu$$

$$\Pi'\ s.\sigma\ \nu \triangleq (\Pi'\ \sigma\ (\pi\ s\ \nu))$$

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 $1 \leq i \leq |\sigma|$.

Consider, for example, the sequence of instructions $a\ x\ b\ c\ y\ a\ b\ a$ resulting from projecting out only the instructions from a trace. Let $a, b, c \in \llparenthesis$ and $x, y \in \rrparenthesis$. The context string associated with each prefix of the projected trace sequence is given as follows:

$\{(a) \mapsto (a), (a\ x) \mapsto \epsilon, (a\ x\ b) \mapsto (b), (a\ x\ b\ c) \mapsto (c\ b), (a\ x\ b\ c\ y) \mapsto (b), (a\ x\ b\ c\ y\ a) \mapsto (a\ b), (a\ x\ b\ c\ y\ a\ b) \mapsto (b\ a\ b), (a\ x\ b\ c\ y\ a\ b\ a) \mapsto (a\ b\ a\ b)\}$.

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 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, of that program.

We denote the set of all finite partial Π-valid context traces as $\wp((\llparenthesis^* \times \Sigma^*)_\Pi \equiv \llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*)$. 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 4.1. $\llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*) \equiv \wp(\Sigma^*)$

Proof Follows from the definition of Π-valid context trace. ∎

This then gives us the framework needed to develop context-sensitive analyses, where context is made explicit. In particular, it gives the framework to derive context-sensitive counterpart of context insensitive analysis.

Assume that an analysis $I \rightarrow Abstore$ 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:

$$\llparenthesis^* \xrightarrow{\Pi} \wp((I \times Store)^*)$$

$$\sqsubseteq \llparenthesis^* \xrightarrow{\Pi} (\wp(I \times Store))^*$$

$$\equiv \llparenthesis^* \xrightarrow{\Pi} (I \rightarrow \wp(Store))^*$$

$$\sqsubseteq \llparenthesis^{Abs} \xrightarrow{\Pi} I \rightarrow Abstore$$

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.

5. Context abstractions

Due to 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 [18]. So while a full call-string analysis may yield the most precise results, it may not be practical to compute it. To make an analysis scalable 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 [25], abstracts a call string by mapping it to its k-length suffix. The second abstraction, introduced by Emami *et al.* [9], 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 [31, 33].

What is true of calling-contexts will also be true for any other instantiation of our generalized notion of context. Hence, it is fruitful 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 independent 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 on 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 no 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, for these are necessary to prove the soundness of any analysis derived from these context abstractions.

5.1 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:

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

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 follows from the lemma below that $\llparenthesis^k$ is an abstraction of $\llparenthesis^*$.

LEMMA 5.1. *α_k is surjective and additive.*

Proof Since a k-context is formed from a k-length prefix, additivity may be shown by using strong ordering on elements of $\llparenthesis^*$. ▮

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 1, the "Context" column provides some examples of contexts. Their corresponding k-context abstractions, with $k = 3$, are shown in the "3-Context" column.

Context	**3-Context**	**ℓ-Context**
abc	abc	abc
$aaaaa$	$aaa\top$	a^+
$abca$	$abc\top$	a^+
$abcaaaaaabc$	$abc\top$	abc^+
$abccba$	$abc\top$	a^+
$aaabbbaaabbbcbbb$	$aaa\top$	a^+b^+

Table 1. Examples of contexts and abstract contexts

5.2 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 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 5.2. $\forall \nu_\ell \in \llparenthesis^\ell : |\nu_\ell| \leq |\llparenthesis|$.

Proof For any element $c \in \llparenthesis$, either c or c^+ may be in ν_ℓ, and each element can occur at most 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 terminates on the opening context a. Table 1 provides examples of contexts and their corresponding ℓ-contexts. Consider the context "$abcaaaaaabc$," which when read right-to-left gives the order in which the contexts were pushed. It is abstracted to abc^+. The term c^+ represents the set of all non-zero length sequences starting with c and ending with c, and thus represents all cyclic context strings from c to c. The term abc^+ thus represents the set of contexts consisting of the opening context a, pushed on the opening context b, pushed on a sequence of openings contexts starting with c and ending with c.

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.

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

Context	**T-Context**
a	$(\bot, \epsilon, a)$
abc	$(((\bot, \epsilon, a), \epsilon, b), \epsilon, c)$
$aaaaa$	$(\bot, [\bot, \bot, \bot, \bot], a)$
$abca$	$(\bot, [((\bot, \epsilon, b), \epsilon, c)], a)$
$abcabc$	$(((\bot, \epsilon, a), \epsilon, b), [((\bot, \epsilon, a), \epsilon, b)], c)$
$cabcaaabab$	$(((\bot, \epsilon, c), \epsilon, a), [((\bot, \epsilon, c), [\bot, \bot], a), (\bot, \epsilon, a)], b)$

Table 2. Examples of mapping contexts and T-contexts

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 2 provides examples of contexts and their corresponding "T-contexts", i.e., the corresponding terms in $\llparenthesis_T$.

We now define an abstraction map $\alpha_\ell : \llparenthesis_T \rightarrow \llparenthesis^\ell$ as follows:

$$\forall t \in \llparenthesis_T; s \in \llparenthesis_T^*; c \in \llparenthesis$$

$$(\alpha_\ell\ \bot) \triangleq \epsilon$$

$$(\alpha_\ell\ (t, s, c)) \triangleq (\alpha_\ell\ t).c^{(\alpha_{\mathbb{B}}\ |s|)}$$

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 $\llparenthesis_T$ as follows:

$$\forall t, t_1, t_2 \in \llparenthesis_T; \forall s_1, s_2 \in \llparenthesis_T^*; \forall c_1, c_2 \in \llparenthesis$$

$$\bot \sqsubseteq_T t,$$

$$(t_1, s_1, c_1) \sqsubseteq_T (t_2, s_2, c_2) \Leftrightarrow t_1 \sqsubseteq_T t_2 \wedge s_1 \sqsubseteq_T^* s_2 \wedge c_1 \sqsubseteq c_2$$

It can be shown that $\sqsubseteq_T$ is reflexive, anti-symmetric, and transitive, and thus defines a partial order on $\llparenthesis_T$.

LEMMA 5.3. *α_ℓ is surjective and additive.*

Proof Follows from structural induction. ▮

Once again, a context-sensitive analysis may be derived by defining $\llparenthesis$ and $\rrparenthesis$, and the appropriate context abstraction $\llparenthesis^\ell \sqsubseteq \llparenthesis^{Abs}$.

6. 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 obfuscated. The semantics of the classic *call* and *ret* instructions consists 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–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 [14].

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 call graph [15]. Most assembly languages do not provide any mechanism to encapsulate procedures. Thus, disassemblers use *call* and *ret* instructions to determine procedure boundaries and to create the call graph [11]. When these instructions are obfuscated, the procedures identified and the call graph created may be questionable and any subsequent interprocedural analyses circumspect.

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:

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

Figure 2. An x86-like assembly language.

6.1 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 to manipulate the stack pointer and the instruction pointer, both of which are registers in the IA32 architecture. Thus, our language captures the essential properties needed to present our algorithm for performing context-sensitive analysis of obfuscated assembly programs.

Figure 2 presents the syntax of the language we use to model our analysis. A program p in this language consists of a sequence of instructions ($seq(i)$). 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, 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.

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 2 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, 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.

Figure 3 presents the semantics of our language. 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. 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: $(\mathcal{I}^p\ i\ \delta) = \{(i', \delta') \mid (i', \delta') \in (\mathcal{I}\ i\ \delta), i' \in p,\ and\ \delta, \delta' \in \Delta^p\}$.

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

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

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 $s_0 \in \Sigma_i^p$ the set of initial states. Following Section 4, the concrete context-trace semantics can be obtained by the least fixpoint of $\mathcal{F}_c : \llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*) \longrightarrow \llparenthesis^* \xrightarrow{\Pi} \wp(\Sigma^*)$ which is mapped from $\mathcal{F} : \wp(\Sigma^*) \longrightarrow \wp(\Sigma^*)$.

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

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

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

Consider the class of programs in which (a) an instruction that modifies the stack pointer always increments or decrements 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 $\llparenthesis'^{\ell}_{asm} \rightarrow Abstore$ or $\llparenthesis'^{k}_{asm} \rightarrow Abstore$ 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:

$$\llparenthesis_{asm} \triangleq \{(i, n) \mid \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) - n \wedge n > 0\}$$

$$\rrparenthesis_{asm} \triangleq \{(i, n) | \exists \delta, \delta' : \delta' \in (\mathcal{I}\ i\ \delta) \wedge (\delta'\ esp) = (\delta\ esp) + n \wedge n > 0\}$$

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

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

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

The function π_{asm} above is a counter-part of the function π described in Section 4. It performs the *push* and *pop* operations on a 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 concrete domains $\llparenthesis_{asm}$ and $\rrparenthesis_{asm}$ form infinite lattices because $\mathbb{N}$ is infinite. Besides, the value of n, representing the size of a context, may not be statically computable. Hence, we need an abstraction of these domains. We use the lattice of 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 $\forall n, n_1, n_2 \in \mathbb{N} : \bot \sqsubseteq_N n \sqsubseteq_N \top$, 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 the 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 just 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. Empirical investigation is needed to ascertain the impact of this increase on the expected computational complexity. Similarly, encoding of the stack graph using binary-decision diagrams (BDD) may lead to efficient computation the exponential relation, as previously reported for context-sensitive pointer analysis [31, 33].

6.3 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

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)

Transition relation:

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

Semantic functions:

$$
\begin{array}{l}
F_{esp} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta \\
\quad F_{esp}\ z\ \delta = [esp \mapsto ((\delta\ esp) + z)]\delta \\
F_{reset} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta \\
\quad F_{reset}\ z\ \delta = \ [esp \mapsto z]\delta \\
F_{*esp} : \mathbb{Z} \rightarrow \Delta \rightarrow \Delta \\
\quad F_{*esp}\ z\ \delta = [l' \mapsto z]\delta, \text{ where } l' = \delta\ esp \\
F_{assign} : R \rightarrow \mathbb{Z} \rightarrow \Delta \rightarrow \Delta \\
\quad F_{assign}\ r\ z\ \delta = [r \mapsto z]\delta \\
F_{*assign} : R \rightarrow \mathbb{Z} \rightarrow \Delta \rightarrow \Delta \\
\quad F_{*assign}\ r\ z\ \delta = [l' \mapsto z]\delta, \text{ where } l' = \delta\ r \\
F_{expr} : E \rightarrow \Delta \rightarrow \mathbb{Z} \\
\quad F_{expr}[\![l]\!]\delta = l \\
\quad F_{expr}[\![z]\!]\delta = z \\
\quad F_{expr}[\![r]\!]\delta = \delta\ r \\
\quad F_{expr}[\![*r]\!]\delta = \delta\ l, \text{ where } l = \delta\ r \\
\quad F_{expr}[\![e_1\ op\ e_2]\!]\delta = F_{expr}[\![e_1]\!]\delta\ op\ F_{expr}[\![e_2]\!]\delta \\
F_{bool} : B \rightarrow \Delta \rightarrow \mathcal{B} \\
\quad F_{bool}[\![true]\!]\delta = true \\
\quad F_{bool}[\![false]\!]\delta = false \\
\quad F_{bool}[\![e_1 < e_2]\!]\delta = F_{expr}[\![e_1]\!]\delta < F_{expr}[\![e_2]\!]\delta \\
\quad F_{bool}[\![\neg b]\!]\delta = \neg F_{bool}[\![b]\!]\delta \\
\quad F_{bool}[\![b_1\ \&\&\ b_2]\!]\delta = F_{bool}[\![b_1]\!]\delta \wedge F_{bool}[\![b_2]\!]\delta
\end{array}
$$

Figure 3. Semantics for our model language of Figure 2.

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) [1] 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 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 $*r$.

6.4 Semantic domain for context-sensitive Venable *et al.*'s algorithm

We now discuss the derivation of semantic domain for the context sensitive version of Venable's *et al.*'s algorithm, a static analyzer that tracks 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) [14] with Reps and Balakrishnan's Value-set Analysis (VSA) [1].

The ASG domain was introduced by [14] 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 and Kumar 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 top of stacks. 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 the ASG and VSA domains to derive the context-insensitive analysis $I \rightarrow R + L \rightarrow ASG \times RIC$. We derive a context-sensitive equivalent of this analysis using the domain $\hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC$. 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 [1] to accelerate fixpoint computation.

6.5 Soundness

The concrete context-trace semantics is given by the least fixpoint 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 fixpoint 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 6.1. $\llparenthesis^*_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*) \sqsubseteq \hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC$.

Proof From Lemma 5.3 and Balakrishnan and Reps' [1] follows that $\llparenthesis^*_{asm} \sqsubseteq \hat{\llparenthesis}^{\ell}_{asm}$ and $\wp(\mathbb{Z}) \sqsubseteq RIC$. Then, it follows that:

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

It follows from lemma 6.1 and the fixpoint transfer theorem that $\mathcal{F}^{\#}$ is a sound approximation of $\mathcal{F}_c$. Though, $\mathcal{F}^{\#}$ may not be complete *w.r.t* $\mathcal{F}_c$.

7. Empirical evaluation

We now present the results of an empirical evaluation of context-sensitive analysis of obfuscated programs. We study the improvements in analysis of obfuscated code resulting from the use of our ℓ-

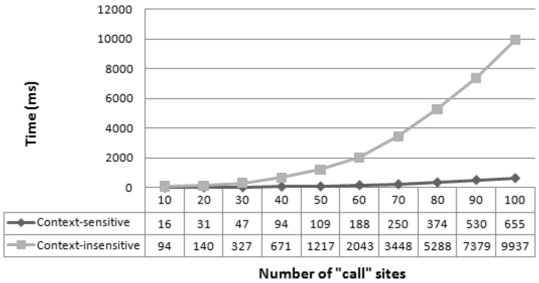


Figure 4. Time evaluation of the set of hand-crafted, obfuscated programs.

context-sensitive version of Venable *et al.*'s analysis [28] against its context-insensitive version. The two versions of the analysis were implemented on the Eclipse workbench and the evaluations 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 also more efficient.

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 nonetheless reveal how the performance varies with the growth of context. 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.

For quantitative comparison we use two metrics: time, measured as CPU time in milliseconds, and size of the sets, measured as 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 $\mathcal{S}_{in}(i)$, and that for context-sensitive analysis is denoted by $\mathcal{S}_{sen}(i)$.

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 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 4 plots the time for analyzing the 10 programs and Figure 5 plots the sizes of the value sets. The results show that for this limited class of programs, the computational cost of context-sensitive analysis grows linearly with the number of contexts. In contrast, the cost of 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.

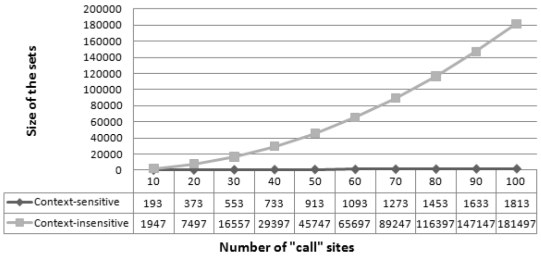


Figure 5. Size of the sets evaluation of the set of hand-crafted, obfuscated programs.

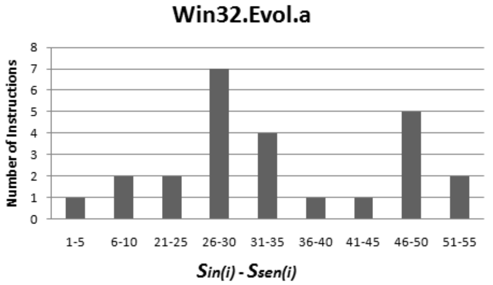


Figure 6. Histogram of approximations for Win32.Evol.a.

Figure 5 shows the size of the value sets for all stores in the whole program in the context-sensitive analysis ($\mathcal{S}_{sen}$) and context-insensitive analysis ($\mathcal{S}_{in}$). Observe that $\mathcal{S}_{sen}$ grows linearly with the growth of contexts; however, $\mathcal{S}_{in}$ 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 the metamorphic virus 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 analysis for each instruction. Since the sizes resulting from context-insensitive analysis are always higher, we compute the difference as $\mathcal{S}_{in}(i) - \mathcal{S}_{sen}(i)$, for instruction i.

Figure 6 presents a histogram that shows the number of instructions where the context-insensitive analysis gives larger sets (for various intervals of differences). The data shows an improvement in precision with 25 of 98 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 the Venable *et al.*'s context-insensitive analysis.

8. Conclusions

We have presented a method for performing context-sensitive analysis for binaries in which calling-contexts cannot be discerned. 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 to protect intellectual property.

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. 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) [1].

We adapt prior work on context-sensitive 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 [25]. We generalize the concept using abstract interpretation and define contexts using trace semantics. We implement a context-sensitive variant of Venable *et al.*'s analysis that combines the VSA and ASG domains [28]. Empirical evaluation shows that context-sensitive analysis using ℓ-context leads to several order of magnitude improvement in the running time and improvement in precision.

While the method improves upon the state-of-the-art by enabling context-sensitive analysis of obfuscated programs 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. Further empirical investigation is needed to study the impact of this increase on the computational complexity of the analysis.

Acknowledgements. We are thankful to Michael Venable for his assistance with the implementation.

References

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

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

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

[4] P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In *IFIP World Computer Congress*, pages 359–366, Toulouse France, August 2004.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, USA, January 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks by abstract interpretation. In *Proc. Principles of Programming Languages (POPL)*, pages 269–282, San Antonio, TX, USA, January 1979.

[7] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proc. Principles of Programming Languages (POPL)*, pages 377–388, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

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

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Not.*, 29(6):242–256, 1994. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/773473.178264.

[10] D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 122–133, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258927.

[11] IdaPro. Ida pro - disassembler, Last accessed October 2010. URL `http://www.hex-rays.com/idapro/`.

[12] U. P. Khedkar, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

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

[14] A. Lakhotia and E. U. Kumar. Abstack stack graph to detect obfuscated calls in binaries. In *4th IEEE Int. Workshop on Source Code Analysis and Manipulation(SCAM)*, Chicago, Illinois, 2004.

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

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

[17] A. Lal and T. Reps. Improving pushdown system model checking. In *Proc. Computer-Aided Verification, 2006*. Springer-Verlag, 2006.

[18] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *Compiler Construction, 15th International Conference, volume 3923 of LNCS*, pages 47–64. Springer, 2006.

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

[20] John McDonald. Delphi falls prey, Last accessed October 2010. URL `http://www.symantec.com/connect/blogs/delphi-falls-prey`.

[21] Matthew Might and Olin Shivers. Environment analysis via Δcfa. *SIGPLAN Not.*, 41(1):127–140, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1111320.1111049.

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

[23] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, October 2005.

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

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

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

[27] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358198.358210.

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

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

[30] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. The design space of metamorphic malware. In *2nd International Conference on i-Warfare and Security (ICIW)*, 2007.

[31] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996859.

[32] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 831–836, New York, NY, USA, 2005. ACM. ISBN 1-59593-058-2. doi: http://doi.acm.org/10.1145/1065579.1065798.

[33] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 145–157, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5.