Context-Sensitive Analysis of Obfuscated x86 Executables

Arun Lakhotia(1), Davidson Boccardo(2), Anshuman Singh(1), and Aleardo Manacero Jr.(2)

(1)University of Louisiana at Lafayette, USA
(2)Paulista State University (UNESP), Brazil

PEPM 2010 (01/19/10)
Madrid, Spain

Disassembled binary with procedures: An example

```
Main:

L1:   PUSH 4
L2:   PUSH 2
L3:   CALL Max
L4:   PUSH 6
L5:   PUSH 4
L6:   CALL Max
L7:   PUSH 0
L8:   CALL ExitProcess
```

```
Max:

L9:    MOV eax, [esp+4]
L10:   MOV ebx, [esp+8]
L11:   CMP eax, ebx
L12:   JG  L14
L13:   MOV eax, ebx
L14:   RET 8
```

Context-sensitive interprocedural data-flow analysis - Classical methods

- Call-string
  - Sharir and Pnueli's k-call string method that maps a call string to its k-length suffix.
  - Emami *et al.*'s method of reducing recursive paths in a call string by a single node.
- Procedure summary
- Inlining

Assumptions of call string based approaches

- The program uses special instructions like call and ret that can be identified and paired statically.
- Valid/invalid paths in ICFG can be described in terms of appropriate pairing of call-ret edges.

Call and Ret are atomic

Call and Ret are atomic in the sense that they:

- Transfer control; and
- Change context

Call obfuscation

Call and Ret can be obfuscated using instructions that transfer control and change context separately. Call obfuscation can be employed by:

- Malware writers $\Rightarrow$ to hide malicious behavior and to evade detection.
- Software developers $\Rightarrow$ to protect intellectual property and to increase security.

Call obfuscation using *push/ret* instructions

```
Main:

L1:    PUSH   4
L2:    PUSH   2
L3:    PUSH   offset  [L6]
L4:    PUSH   offset  [L13]
L5:    RET
L6:    PUSH   6
L7:    PUSH   4
L8:    PUSH   offset  [L11]
L9:    PUSH   offset  [L13]
L10:   RET
L11:   PUSH   0
L12:   CALL   ExitProcess
```

```
Max:

L13:   MOV    eax, [esp+4]
L14:   MOV    ebx, [esp+8]
L15:   CMP    eax, ebx
L16:   JG     L18
L17:   MOV    eax, ebx
L18:   RET    8
```

Call obfuscation using *push/jmp* instructions

```
Main:

L1:    PUSH   4
L2:    PUSH   2
L3:    PUSH   offset [L5]
L4:    JMP    Max
L5:    PUSH   6
L6:    PUSH   4
L7:    PUSH   offset [L9]
L8:    JMP    Max
L9:    PUSH   0
L10:   CALL   ExitProcess
```

```
Max:

L11:   MOV    eax, [esp+4]
L12:   MOV    ebx, [esp+8]
L13:   CMP    eax, ebx
L14:   JG     L16
L15:   MOV    eax, ebx
L16:   RET    8
```

Motivation

Classical call string based analyses are not directly applicable for context-sensitive analysis of binaries that have obfuscated calls. This is because:

- They are tied to semantics of procedure call and return statements of high-level languages, and therefore, call and ret instructions of assembly language.

Proposed method

Objective: Design of a context-sensitive analysis based on program semantics and abstract interpretation resilient from call and ret obfuscation attacks.

Steps

1. Context abstractions (generic versions independent of ICFG based definitions)
2. Context-trace semantics (can not rely on ICFG based soundness results)
3. Language (a simple assembly language without call and ret)
4. Stack context (to model change of context)
5. Transfer of control (is modeled using value-set analysis)
6. Derive the context sensitive analyzer from context-insensitive one
7. Prove soundness of our analysis

Generalized notion of contexts

- Opening and closing instructions are defined by:
  - $\llparenthesis \subseteq I$ - the set of instructions that open contexts.
  - $\rrparenthesis \subseteq I$ - the set of instructions that close contexts.
- For example, in the conventional interprocedural analysis, the set $\llparenthesis$ contains the call instructions and $\rrparenthesis$ contains the ret instructions.
- A context-string is a sequence of instructions that open contexts, represented by $\llparenthesis^* \subseteq I^*$.

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:

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

- $\llparenthesis^*$ and $\llparenthesis^k$ form a Galois insertion with the abstraction map α_k

ℓ-context

- $\llparenthesis^{\ell}$ represent the set of sequence that open contexts with size $\leq |\llparenthesis|$ and have cyclic sequence represented by $+$.
- For example, the term c^{+} represents all cyclic context strings from c to c.
- A map $\alpha_{\ell} : \llparenthesis^{*} \rightarrow \llparenthesis^{\ell}$ can be defined such that $\llparenthesis^{*}$ and $\llparenthesis^{\ell}$ form a Galois insertion with the abstraction map α_{ℓ}.

Examples of context abstractions

Context	2-Context	ℓ-Context
$c_2 c_1$	$c_2 c_1$	$c_2 c_1$
$c_2 c_3 c_2 c_1$	$c_2 c_3 \top$	$c_2^+ c_1$
$c_2 c_4 c_2 c_1$	$c_2 c_4 \top$	$c_2^+ c_1$
$c_2 c_4 c_2 c_3 c_2 c_1$	$c_2 c_4 \top$	$c_2^+ c_1$
$c_2 c_3 c_2 c_4 c_2 c_1$	$c_2 c_3 \top$	$c_2^+ c_1$
$c_3 c_2 c_4 c_2 c_1$	$c_3 c_2 \top$	$c_3 c_2^+ c_1$
$c_2 c_4 c_2 c_1$	$c_2 c_4 \top$	$c_2^+ c_1$
$c_5 c_2 c_4 c_2 c_1$	$c_5 c_2 \top$	$c_5 c_2^+ c_1$
$c_3 c_5 c_2 c_4 c_2 c_1$	$c_3 c_5 \top$	$c_3 c_5 c_2^+ c_1$
$c_5 c_5 c_2 c_4 c_2 c_1$	$c_5 c_5 \top$	$c_5^+ c_2^+ c_1$
$c_2 c_1$	$c_2 c_1$	$c_2 c_1$
ϵ	ϵ	ϵ

Context-trace semantics

- A context-trace is a pair of a context string and a trace $(\nu, \sigma) \in (\llparenthesis^* \times \Sigma^*)$.
- The set of all context-traces of a program, denoted by $\wp(\llparenthesis^* \times \Sigma^*) \equiv \llparenthesis^* \to \wp(\Sigma^*)$, gives its context-trace semantics.

Language

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:

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

Mapping Call and Ret in our language

- An instruction "*Call l*" may be mapped to the following sequence of instructions in our language:

$$l_0 : esp = esp - 1 \,\textbf{.}\, eip = l_1$$
$$l_1 : *esp = l_2 \,\textbf{.}\, eip = l$$

  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:

$$l_0 : esp = esp + 1 \,\textbf{.}\, eip = *esp$$

Stack Context

- Idea: To have the information about instructions that manipulate the stack pointer as a part of the context.
- The stack context can be described as the set of opening contexts and closing contexts represented by domains $(\!|_{asm} \subseteq I \times \mathbb{N}$ and $|\!)_{asm} \subseteq I \times \mathbb{N}$ resp. that are defined as:

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

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

- A context string is a sequence belonging to $(\!|^*_{asm}$. Abstractions k-context and l-context can be applied to $(\!|^*_{asm}$ to reduce the complexity of the analysis.

Transfer of control

- Upon execution of each instruction the instruction pointer register, *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.
- We use Balakrishnan and Reps' Value-Set Analysis (*VSA*) to recover information about the contents of memory locations and registers. VSA uses the domain $RIC = \mathbb{N} \times \mathbb{Z} \times \mathbb{Z}$ to abstract $\wp(\mathbb{Z})$.

Derivation of a static analyzer

The analysis is derived from a chain of Galois connections linking the concrete domain $\wp((I \times \mathit{Store})^*)$ to the analysis domain $I \rightarrow \mathit{AbStore}$. The steps of the derivation are:

- The set $\wp((I \times \mathit{Store})^*)$, called set of traces, is approximated to trace of sets, represented by $(\wp(I \times \mathit{Store}))^*$.
- The trace of sets is equivalent to $(I \rightarrow \wp(\mathit{Store}))^*$. This sequence of mapping of instructions to set of stores can be approximated to $I \rightarrow \wp(\mathit{Store})$.
- Finally, a Galois connection between $\wp(\mathit{Store})$ and *AbStore* completes the analysis.

Deriving the context-sensitive analyzer

Starting from concrete domain $(\!|^*_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*)$ and the domain for Venable *et al.*'s context insensitive analyzer $I \rightarrow R + L \rightarrow ASG \times RIC$, we obtain our context sensitive analyzer analyzer $\hat{(\!|}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC$ using the following results:

1. $(\!|^*_{asm} \sqsubseteq \hat{(\!|}^{\ell}_{asm}$
2. $\wp(\mathbb{Z}) \sqsubseteq RIC$
3. $(\!|^* \xrightarrow{\Pi} \wp(\Sigma^*) \equiv \wp(\Sigma^*)$

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 $\Sigma = I \times R + L \to \mathbb{Z}$.

- The context-trace semantics of the context-sensitive analyzer is given by the least fixpoint of the function $\mathcal{F}^{\#}$: $(\hat{\llparenthesis}^{\ell}_{asm} \to I \to R + L \to RIC) \longrightarrow (\hat{\llparenthesis}^{\ell}_{asm} \to I \to R + L \to RIC)$.

Soundness

Lemma

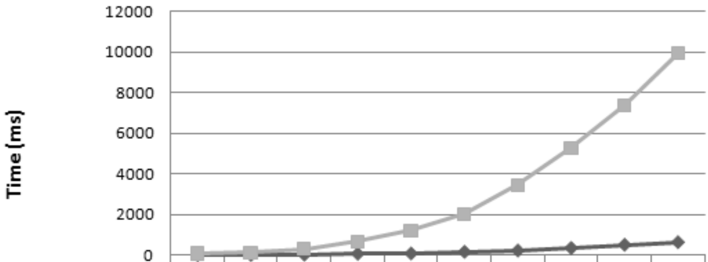
$$\llparenthesis^{*}_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^{*}) \sqsubseteq \hat{\llparenthesis}^{\ell}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC.$$

It follows from the lemma and the fixpoint transfer theorem that $\mathcal{F}^{\#}$ is a sound approximation of $\mathcal{F}_c$.

DOC (Detector of Obfuscated Calls)

- We implemented our derived analysis in a tool called DOC.
- We studied the improvements in analysis of obfuscated code resulting from the use of our ℓ-context-sensitive version of Venable *et al.*'s analysis against its context-insensitive version.
- We performed the analysis using two sets of programs:
  - Programs in the first set were hand-crafted with a certain known obfuscated calling structure.
  - The second set contains W32.Evol.a, a metamorphic virus that employs call obfuscation.

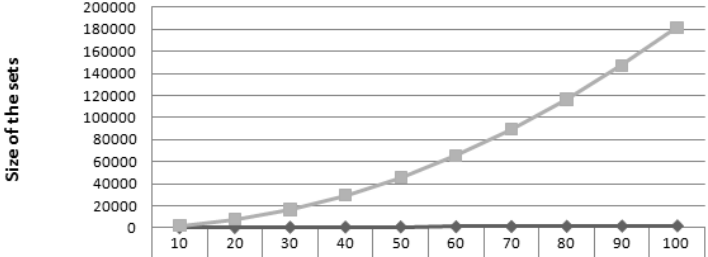
Time evaluation



	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

Number of "call" sites

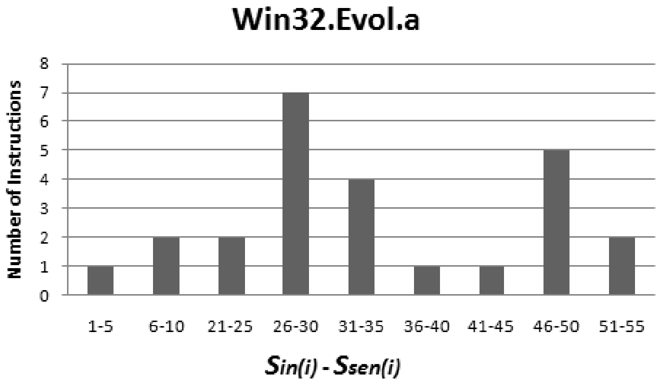
Size of sets evaluation



	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

Number of "call" sites

Histogram of evaluations for Win32.Evol.a



Conclusions

- Developed a method for performing context sensitive analysis of binaries in which calling contexts cannot be discerned.
- Systematically derived generic versions of Sharir and Pnueli's k-suffix call-strings abstractions and Emami *et al.*'s strategy of abstracting calling-contexts (referred to as *l*-context in our work).
- Introduced the concept of stack-context, used in *lieu* of calling context, to perform context sensitive analysis of binaries that use call obfuscation.
- Proposed a general method for deriving sound context-sensitive analysis from context-insensitive one.