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 in the sense that they:

- Transfer control; and

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

- **Malware writers** ⇒ to hide malicious behavior and to evade detection.
- **Software developers** ⇒ 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

<table>
<thead>
<tr>
<th>Main:</th>
<th>Max:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: PUSH 4</td>
<td>L11: MOV eax, [esp+4]</td>
</tr>
<tr>
<td>L2: PUSH 2</td>
<td>L12: MOV ebx, [esp+8]</td>
</tr>
<tr>
<td>L4: JMP Max</td>
<td>L14: JG L16</td>
</tr>
<tr>
<td>L5: PUSH 6</td>
<td>L15: MOV eax, ebx</td>
</tr>
<tr>
<td>L6: PUSH 4</td>
<td>L16: RET 8</td>
</tr>
<tr>
<td>L7: PUSH offset [L9]</td>
<td></td>
</tr>
<tr>
<td>L8: JMP Max</td>
<td></td>
</tr>
<tr>
<td>L9: PUSH 0</td>
<td></td>
</tr>
<tr>
<td>L10: CALL ExitProcess</td>
<td></td>
</tr>
</tbody>
</table>
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.
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:
  - $\Vert \subseteq I$ - the set of instructions that open contexts.
  - $\triangledown \subseteq I$ - the set of instructions that close contexts.

- For example, in the conventional interprocedural analysis, the set $\Vert$ contains the \texttt{call} instructions and $\triangledown$ contains the \texttt{ret} instructions.

- A \textit{context-string} is a sequence of instructions that open contexts, represented by $\Vert^* \subseteq I^*$. 
Let \( \mathcal{L}^k \) represent the set of sequences of opening contexts of length \( \leq k \) and \( k + 1 \) length sequences created by appending \( \top = \bigcup \emptyset \) to \( k \)-length sequences of opening contexts.

An element of \( \mathcal{L}^k \) is called a \( k \)-context. We can establish a map \( \alpha_k : \mathcal{L}^* \rightarrow \mathcal{L}^k \) as:

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

\( \mathcal{L}^* \) and \( \mathcal{L}^k \) form a Galois insertion with the abstraction map \( \alpha_k \).
\( \ell \)-context

- \( \ell^\ell \) represent the set of sequence that open contexts with size \( \leq |\ell| \) and have cyclic sequence represented by \(+\).

- For example, the term \( c^+ \) represents all cyclic context strings from \( c \) to \( c \).

- A map \( \alpha_\ell : \ell^* \rightarrow \ell^\ell \) can be defined such that \( \ell^* \) and \( \ell^\ell \) form a Galois insertion with the abstraction map \( \alpha_\ell \).
Examples of context abstractions

<table>
<thead>
<tr>
<th>Context</th>
<th>2-Context</th>
<th>(\ell)-Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_2c_1)</td>
<td>(c_2c_1)</td>
<td>(c_2c_1)</td>
</tr>
<tr>
<td>(c_2c_3c_2c_1)</td>
<td>(c_2c_3)</td>
<td>(c_2^+c_1)</td>
</tr>
<tr>
<td>(c_2c_4c_2c_1)</td>
<td>(c_2c_4)</td>
<td>(c_2c_1)</td>
</tr>
<tr>
<td>(c_2c_4c_2c_3c_2c_1)</td>
<td>(c_2c_4)</td>
<td>(c_2c_1)</td>
</tr>
<tr>
<td>(c_2c_3c_2c_4c_2c_1)</td>
<td>(c_2c_3)</td>
<td>(c_2^+c_1)</td>
</tr>
<tr>
<td>(c_3c_2c_4c_2c_1)</td>
<td>(c_3c_2)</td>
<td>(c_3c_2c_1)</td>
</tr>
<tr>
<td>(c_2c_4c_2c_1)</td>
<td>(c_2c_4)</td>
<td>(c_2c_1)</td>
</tr>
<tr>
<td>(c_5c_2c_4c_2c_1)</td>
<td>(c_5c_2)</td>
<td>(c_5^+c_1)</td>
</tr>
<tr>
<td>(c_3c_5c_2c_4c_2c_1)</td>
<td>(c_3c_5)</td>
<td>(c_3c_5c_2^+c_1)</td>
</tr>
<tr>
<td>(c_5c_5c_2c_4c_2c_1)</td>
<td>(c_5c_5)</td>
<td>(c_5^+c_2^+c_1)</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>(\epsilon)</td>
<td>(\epsilon)</td>
</tr>
</tbody>
</table>
A context-trace is a pair of a context string and a trace \((\nu, \sigma) \in (L^* \times \Sigma^*)\).

The set of all context-traces of a program, denoted by \(\wp(L^* \times \Sigma^*) \equiv \wp(\Sigma^*)\), gives its context-trace semantics.
**Syntactic Categories:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b \in B$</td>
<td>boolean expressions</td>
</tr>
<tr>
<td>$e, e' \in E$</td>
<td>integer expressions</td>
</tr>
<tr>
<td>$i \in I$</td>
<td>instructions</td>
</tr>
<tr>
<td>$l, l' \in L \subseteq \mathbb{Z}$</td>
<td>labels</td>
</tr>
<tr>
<td>$z \in \mathbb{Z}$</td>
<td>integers</td>
</tr>
<tr>
<td>$p \in P$</td>
<td>programs</td>
</tr>
<tr>
<td>$r \in R$</td>
<td>references</td>
</tr>
</tbody>
</table>

**Syntax:**

- $e ::= l | z | r | r' | e_1 \, op \, e_2$
  \quad (op \in \{+, -, \ast, /,...\})

- $b ::= true \mid false \mid e_1 < e_2 \mid \neg b$

- $b1 \&\& b2$

- $i ::= l : esp = esp + e . eip = e' |
  \quad l : esp = e . eip = e' |
  \quad l : *esp = e . eip = e' |
  \quad l : r = e . eip = e' |
  \quad l : *r = e . eip = e' |
  \quad l : if (b) eip = e; eip = l'$

- $p ::= seq(i)$
An instruction “Call l” may be mapped to the following sequence of instructions in our language:

\[
\begin{align*}
l_0 : \text{esp} &= \text{esp} - 1 . \ eip = l_1 \\
l_1 : \ast \text{esp} &= l_2 . \ eip = l
\end{align*}
\]

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:

\[
\begin{align*}
l_0 : \text{esp} &= \text{esp} + 1 . \ eip = \ast \text{esp}
\end{align*}
\]
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 \( \mathcal{L}_{asm} \subseteq I \times \mathbb{N} \) and \( \mathcal{M}_{asm} \subseteq I \times \mathbb{N} \) resp. that are defined as:

\[
\mathcal{L}_{asm} \triangleq \{(i, n) | \exists \delta, \delta' : \delta' \in (\mathcal{I} i \delta) \land (\delta' \ esp) = (\delta \ esp) - n\}
\]

\[
\mathcal{M}_{asm} \triangleq \{(i, n) | \exists \delta, \delta' : \delta' \in (\mathcal{I} i \delta) \land (\delta' \ esp) = (\delta \ esp) + n\}
\]

A context string is a sequence belonging to \( \mathcal{L}_{asm}^* \). Abstractions k-context and l-context can be applied to \( \mathcal{L}_{asm}^* \) 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 $\wp((I \times \text{Store})^*)$ to the analysis domain $I \rightarrow \text{AbStore}$. The steps of the derivation are:

- The set $\wp((I \times \text{Store})^*)$, called set of traces, is approximated to trace of sets, represented by $\wp(I \times \text{Store})^*$.  

- The trace of sets is equivalent to $(I \rightarrow \wp(\text{Store}))^*$. This sequence of mapping of instructions to set of stores can be approximated to $I \rightarrow \wp(\text{Store})$.

- Finally, a Galois connection between $\wp(\text{Store})$ and $\text{AbStore}$ completes the analysis.
Deriving the context-sensitive analyzer

Starting from concrete domain \( \langle \ast_{asm} \rangle \xrightarrow{\prod_{asm}} \wp(\Sigma^*) \) 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 \( \langle \cdot \rangle_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC \) using the following results:

1. \( \langle \ast_{asm} \rangle \sqsubseteq \langle \cdot \rangle_{asm} \)
2. \( \wp(\mathbb{Z}) \sqsubseteq RIC \)
3. \( \langle \ast \prod \rangle \wp(\Sigma^*) \equiv \wp(\Sigma^*) \)
The concrete context-trace semantics is given by the least fixpoint of the function
\[ F_c : (I_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*)) \rightarrow (I_{asm} \xrightarrow{\Pi_{asm}} \wp(\Sigma^*)), \]
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
\[ F^\# : (\hat{l}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC) \rightarrow (\hat{l}_{asm} \rightarrow I \rightarrow R + L \rightarrow RIC). \]
Soundness

Lemma

\[ \Pi_{\text{asm}} \xrightarrow{L_{\text{asm}}} \varphi(\Sigma^*) \sqsubseteq L^\ell_{\text{asm}} \rightarrow I \rightarrow R + L \rightarrow RIC. \]

It follows from the lemma and the fixpoint transfer theorem that \( F^\# \) is a sound approximation of \( F_c \).
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 $\ell$-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

![Graph showing Time Evaluation]

<table>
<thead>
<tr>
<th>Number of &quot;call&quot; sites</th>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>16</td>
<td>94</td>
</tr>
<tr>
<td>20</td>
<td>31</td>
<td>140</td>
</tr>
<tr>
<td>30</td>
<td>47</td>
<td>327</td>
</tr>
<tr>
<td>40</td>
<td>94</td>
<td>671</td>
</tr>
<tr>
<td>50</td>
<td>109</td>
<td>1217</td>
</tr>
<tr>
<td>60</td>
<td>188</td>
<td>2043</td>
</tr>
<tr>
<td>70</td>
<td>250</td>
<td>3448</td>
</tr>
<tr>
<td>80</td>
<td>374</td>
<td>5288</td>
</tr>
<tr>
<td>90</td>
<td>530</td>
<td>7379</td>
</tr>
<tr>
<td>100</td>
<td>655</td>
<td>9937</td>
</tr>
</tbody>
</table>
Size of sets evaluation

![Graph showing the size of sets evaluation with two lines representing context-sensitive and context-insensitive cases. The x-axis represents the number of "call" sites, ranging from 10 to 100. The y-axis represents the size of the sets, ranging from 0 to 200,000. The table below the graph lists the size of sets for different number of "call" sites:

<table>
<thead>
<tr>
<th>Number of &quot;call&quot; sites</th>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>193</td>
<td>1947</td>
</tr>
<tr>
<td>20</td>
<td>373</td>
<td>7497</td>
</tr>
<tr>
<td>30</td>
<td>553</td>
<td>16557</td>
</tr>
<tr>
<td>40</td>
<td>733</td>
<td>29397</td>
</tr>
<tr>
<td>50</td>
<td>913</td>
<td>45747</td>
</tr>
<tr>
<td>60</td>
<td>1093</td>
<td>65697</td>
</tr>
<tr>
<td>70</td>
<td>1273</td>
<td>89247</td>
</tr>
<tr>
<td>80</td>
<td>1453</td>
<td>116397</td>
</tr>
<tr>
<td>90</td>
<td>1633</td>
<td>147147</td>
</tr>
</tbody>
</table>
| 100                    | 1813              | 181497             |]
Histogram of evaluations for Win32.Evol.a

Win32.Evol.a

Number of Instructions

$\text{Sin}(i) - \text{Ssen}(i)$

- 1-5
- 6-10
- 21-25
- 26-30
- 31-35
- 36-40
- 41-45
- 46-50
- 51-55

0 1 2 3 4 5 6 7 8
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.