Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis

Silvio Cesare
<silvio.cesare@gmail.com>
Who am I and where did this talk come from?

- Ph.D. Student at Deakin University
- Book Author
- This talk covers some of my Ph.D. research.
Introduction

• Detecting bugs in binary is useful
  – Black-box penetration testing
  – External audits and compliance
  – Verification of compilation and linkage
  – Quality assurance of 3rd party software
Innovation in this work

• Performing static analysis on binaries by:
  – Using decompilation
  – And using data flow analysis on the high level results

• The novelty is in combining decompilation and traditional static analysis techniques
Formal Methods of Program Analysis

- Theorem Proving →
  \[
  \frac{R(r1) \rightarrow n1}{(r3 := r1, P) \Rightarrow P[pc = pc + 1, R[r3 \leftrightarrow n1]]}
  \]
  \[
  \{P\}S\{Q\}, \{Q\}T\{R\}
  \]
  \[
  \{P\}S;T\{R\}
  \]

- Abstract Interpretation →

- Model Checking →
Outline

• Decompilation
• Data Flow Analysis
• IL Optimisation
• Bug Detection
• Bugwise
• Future Work and Conclusion
• Control Flow Graphs represents control flow within a procedure

• Intraprocedural analysis works on a single procedure.
  – Flow sensitive analyses take control flow into account
  – Pointer analyses can be flow insensitive
Terminology (2)

- Call Graphs represent control flow between procedures
- Interprocedural analysis looks at all procedures in a module at once
  - Context sensitive analyses take into account call stacks
Decompilation
Decompilation overview

- Recovers source-level information from a binary

- Approach
  - Representing x86 with an intermediate language (IL)
  - Inferring stack pointers
  - Decompiling locals and procedure arguments
Wire – An Formal Language for Binary Analysis

- x86 is complex and big
- Wire is a low level RISC assembly style language
- Translated from x86
- Formally defined operational semantics

\[ \begin{align*}
R(r1) & \rightarrow n1 \\
M(n1) & \rightarrow n2 \\
(r3:*=*(r1), P) & \Rightarrow P[pc = pc + 1, R[r3 \leftrightarrow n2]]
\end{align*} \]

The LOAD instruction implements a memory read.
Reg_name("eax") = 0
Reg_name("ebx") = 1
Reg_name("zf") = 100

In the first part of the dead code equivalence proof we execute the instructions without the dead code.

\[
R(0) \rightarrow n1
\]
\[
n3 = n1 + 50
\]
\[
("BOPC_{ADD} 0,50,0", t) \Rightarrow t'
\]
\[
t' = P[pc = pc + 1, R[0 \rightarrow n1 + 50]]
\]

In the second part of the proof we execute the instructions with the dead code.

\[
R(0) \rightarrow n1
\]
\[
n3 = n1 - 50
\]
\[
("BOPC_{SUB} 0,50,0", s') \Rightarrow s''
\]
\[
t'' = P[pc = pc + 1, R[0 \rightarrow n3]]
\]
\[
t'' = P[pc = pc + 1, R[0 \rightarrow (n1 + 50) - 50]]
\]

\[
R(0) \rightarrow n1
\]
\[
n3 = 0
\]
\[
("ASSIGNC 0,0", t'') \Rightarrow t'''
\]
\[
t''' = P[pc = pc + 2, R[0 \rightarrow n1]]
\]
\[
t''' = P[pc = pc + 2, R[0 \rightarrow 0]]
\]

Now we can see that t'''-pc = s'-pc which means they are semantically equivalent when ignoring the effect the code has on the program counter. We also note that s' and s'' are semantically equivalent. We have thus proven the obfuscated and deobfuscated code samples are equivalent.
Stack Pointer Inference

- Proposed in HexRays decompiler - http://www.hexblog.com/?p=42

- Estimate Stack Pointer (SP) in and out of basic block
  - By tracking and estimating SP modifications using linear equalities

- Solve.

```c
// ESP at the entry point is zero
in_0 = 0
// ESP at return instructions is zero
out_4 = 0
// Equations derived from control flow edges:
in_1 - out_0 = 0
in_2 - out_1 = 0
in_3 - out_1 = 0
in_3 - out_2 = 0
in_4 - out_0 = 0
// Equations derived from block contents:
out_0 - in_0 = 0 // block does not change ESP
out_1 - in_1 <= 8 // because of 2 pushes
out_2 - in_2 = 0 // block does not change ESP
out_3 - in_3 = 0 // block does not change ESP
out_4 - in_4 = 0 // block does not change ESP
```
Local Variable Recovery

• Based on stack pointer inference
• Access to memory offset to the stack
• Replace with native Wire register

```
Imark  ($0x80483f5, , )
AddImm32 (%esp(4), $0x1c, %temp_memreg(12c))
LoadMem32 (%temp_memreg(12c), , %temp_op1d(66))
Imark  ($0x80483f9, , )
StoreMem32(%temp_op1d(66), , %esp(4))
Imark  ($0x80483fc, , )
SubImm32 (%esp(4), $0x4, %esp(4))
LoadImm32 ($0x80483fc, , %temp_op1d(66))
StoreMem32(%temp_op1d(66), , %esp(4))
Lcall  (, , $0x80482f0)
```
Procedure Parameter and Argument Recovery

• Based on stack pointer inference

• Offset relative to ESP/EBP indicates local or argument

• Arguments also live registers on procedure entry

<table>
<thead>
<tr>
<th>Free</th>
<th>(%local_28(186bc), , )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imark</td>
<td>($0x8048401, , )</td>
</tr>
<tr>
<td>Imark</td>
<td>($0x8048405, , )</td>
</tr>
<tr>
<td>Imark</td>
<td>($0x8048408, , )</td>
</tr>
<tr>
<td>PushArg32</td>
<td>($0x0, %local_28(186bc), )</td>
</tr>
<tr>
<td>Args</td>
<td>(, , )</td>
</tr>
<tr>
<td>Call</td>
<td>(, , *0x30)</td>
</tr>
</tbody>
</table>
Data Flow Analysis
Data Flow Analysis overview

• Data Flow Analysis (DFA) reasons about data

• DFA is conservative
  – It over-approximates
  – But should not under-approximate

• DFA is what an optimising compiler uses

• Analyses
  – Reaching Definitions
  – Upwards Exposed Uses
  – Live Variables
  – Reaching Copies
  – etc
Monotone Frameworks

- Models many data flow problems
- Sets of data entering (in) and leaving (out) of basic blocks
- Set up equations (forwards analysis)
  - Data entering or leaving basic block is initialised
  - Transfer function performs action on data in a basic block
    \[ out_b = \text{transfer function}(in_b) \]
  - Join operator combines predecessors in control flow graph
    \[ in_b = \text{join}(\{ p \mid p \in \text{predecessor}_b \}) \]
A reaching definition is a definition of a variable that reaches a program point without being redefined.
A Framework for Data Flow Analysis

• Forwards and backwards analysis

• Initialise in, out, gen, kill sets for each BB.

• Transfer function (forward analysis) is defined as:

\[
out[B] = gen[B] \cup (in[B] \setminus kill[B])
\]

• Join operator is Union or Intersection.
• Gen and Kill sets
  - gen[B] = \{ definitions that appear in B and reach the end of B\}
  - kill[B] = \{ all definitions that never reach the end of B\}

• Initialisation
  - out[B] = gen[B]

• Confluence Operator
  - Join = Union
  - in[B] = U out[P] for predecessors P of B
Upward Exposed Uses

• The uses of a definition

• Gen and Kill sets
  – gen[B] = \{ (s, x) | s is a use of x in B and there is no definition of x between the beginning of B and s\}
  – kill[B] = \{ (s, x) | s is a use of x not in B and B contains a definition of x\}

• Initialisation
  – in[B] = \{0\}

• Confluence Operator
  – Join = Union
  – out[B] = U in[S] for successors S of B
More Data Flow Problems

• Live Variables
  – A variable is live if it will be subsequently read without being redefined.

• Reaching Copies
  – The reach of a copy statement

• More DFA analyses used in optimising compilers
  – Available expressions
  – Very busy expressions
  – etc
An Iterative Solution

- Initialise
- Apply transfer function and join.
- Iterate over all nodes in the control flow graph
- Stop when the nodes’ data stabilise
- A “Fixed Point”
A Logic-based Solution

- Data flow can be analysed using logic
- Datalog is a syntactic subset of prolog
- Represent analyses and solve

\[
\text{Reach}(d,x,j):= \text{Reach}(d,x,i), \\
\text{StatementAt}(i,s), \\
!\text{Assigns}(s,x), \\
\text{Follows}(i,j).
\]

\[
\text{Reach}(s,x,j):= \text{StatementAt}(i,s), \\
\text{Assigns}(s,x), \\
\text{Follows}(i,j).
\]
Interprocedural Analysis

- Dataflow analysis works on the intraprocedural CFG
- So.. Make an interprocedural CFG (ICFG)
- Replace Calls with branches
- Replace Returns with branches back to callsite
- Apply monotone analysis
IL Optimisation
IL Optimisation overview

• Required to perform other analyses
  – Decompilation
  – Bug Detection

• Reduces the size of IL code

• Optimisations based on data flow analysis
  – Constant Folding and Propagation
  – Copy Propagation
  – Backwards Copy Propagation
  – Dead Code Elimination
  – etc
Constant Folding

- Motivation - replace $x=5 + 5$ with $x=10$
- For each arithmetic operator
  - If the reaching definition of each operand is a single constant assignment
  - Fold constants in instruction
Constant Propagation

• Motivation – reduce number of assignments

- $x = 34$
- $r = x + y$
- Print(r)

$\rightarrow$

• If all the reaching definitions of a variable have the same assignment and it is constant:
  – The constant can be propagated to the variable
Copy Propagation

• Motivation – reduce number of copies
  
  y=x
  z=2
  r=y+z
  Print(r)

• For a statement u where x is being used:
  – Statement s is the only definition of x reaching u
  – On every path from s to u there are no assignments to y.

• Or.. At each use of x where x=y is a reaching copy, replace x with y.
• Motivation – reduce number of copies

\[
\begin{align*}
  x &= 34 \\
  y &= 4 \\
  r_1 &= x + y \\
  r_2 &= r_1
\end{align*}
\]

\[
\begin{align*}
  x &= 34 \\
  y &= 4 \\
  r_2 &= x + y
\end{align*}
\]

• In Bugwise, both forwards and backwards copy propagation are required.
Dead Code Elimination

- Motivation – reduce number of instructions
- For any definition of a variable:
  - If the variable is not live, then eliminate the instruction.

\[
\begin{align*}
x &= 34 \quad (x \text{ is not live}) \\
x &= 10 \\
\text{Print}(x)
\end{align*}
\]

\[
\begin{align*}
x &= 10 \\
\text{Print}(x)
\end{align*}
\]
Bug Detection
Bug detection overview

- **Decompilation**
  - Transforms locals to native IL variables

- **Data Flow Analysis**
  - Reasons about IL variables
  - When variables are used and defined

- **Bug Detection**
  - `getenv()`
  - Use-after-free
  - Double free
Detect unsafe applications of `getenv()`

Example: `strcpy(buf, getenv("HOME"))`

For each `getenv()`
- If return value is live
- And it’s the reaching definition to the 2\textsuperscript{nd} argument to `strcpy()`/`strcat()`
- Then warn

P.S. 2001 wants its bugs back.
• For each `free(ptr)`
  – If `ptr` is live
  – Then warn

```c
void f(int x)
{
    int *p = malloc(10);
    dowork(p);
    free(p);
    if (x)
        p[0] = 1;
}
```
Double free

• For each free(ptr)
  – If an upward exposed use of ptr’s definition is free(ptr)
  – Then warn

• 2001 calls again

```c
void f(int x)
{
    int *p = malloc(10);
    dowork(p);
    free(p);
    if (x)
        free(p);
}
```
Bugwise
Implementation

• Built on my previous Malwise system

• Malwise is over 100,000 LOC C++

• Bugwise is a set of loadable modules

• Everything in this talk and more is implemented
getenv() bugs results

• Scanned entire Debian 7 unstable repository
• ~123,000 ELF binaries
• 30,450 not scanned.
• 85 bug reports
• 47 packages reported

| 4digits | ptop |
| acedb-other-belu | recordmydesktop |
| acedb-other-dotter | rlplot |
| bvi | sapphire |
| comgt | sc |
| csmash | scm |
| elvis-tiny | sgrep |
| fwm | slurm-llnl-slurmdbd |
| garmin-ant-downloader | statserial |
| gcin | stopmotion |
| gexec | supertransball2 |
| gmorgan | theorur |
| gopher | twpsk |
| gsoko | udo |
| gstm | vnc4server |
| hime | wily |
| le-dico-de-rene-cougnenc | wmpinboard |
| libreoffice-dev | wm PPP.app |
| libxgks-dev | xboing |
| lie | xemacs21-bin |
| lipe | xjdic |
| mp3rename | xmotd |
ELF Binary Sizes

- Linear growth with logarithmic scaling plus outliers
Cumulative getenv() bugs over time - sorted by binary size

- Linear or power growth?
getenv() bug statistics

- Probability (P) of a binary being vulnerable: 0.00067
- P. of a package being vulnerable: 0.00255
- P. of a package having a 2nd vulnerability given that one binary in the package is vulnerable: 0.52380

\[
P(A | B) = \frac{P(A \cap B)}{P(B)}
\]
Double free SGID games “xonix” in Debian 6

```c
memset(score_rec[i].login, 0, 11);
strncpy(score_rec[i].login, pw->pw_name, 10);
memset(score_rec[i].full, 0, 65);
strncpy(score_rec[i].full, fullname, 64);
score_rec[i].tstamp = time(NULL);

free(fullname);

if((high = freopen(PATH_HIGHSCORE, "w", high)) == NULL) {
    fprintf(stderr, "xonix: cannot reopen high score file\n");
    free(fullname);
    gameover_pending = 0;
    return;
}
```
Bugalyzer.com

**Bugwise** A binary-level bug detection service by FooCodeChu.

**Submission Details**

<table>
<thead>
<tr>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0c0cd3c06c0a0d09102566bf7b762449</td>
</tr>
</tbody>
</table>

**Results Summary** *(Permanent link to this report)*

1 Double frees detected.

**Results**

<table>
<thead>
<tr>
<th>Double free</th>
<th>Double free at 0x804c910 and 0x804c91c</th>
</tr>
</thead>
</table>

**Bugwise** A binary-level bug detection service by FooCodeChu.

**Submission Details**

<table>
<thead>
<tr>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>9e8a7c678b3465cf0eeef02ce691d47a</td>
</tr>
</tbody>
</table>

**Results Summary** *(Permanent link to this report)*

1 Buffer overflows detected.

**Results**

<table>
<thead>
<tr>
<th>Buffer overflow</th>
<th>Results of getenviron() overflow strcpy function</th>
</tr>
</thead>
</table>
EC2 Infrastructure

- Web Server
- SCM Server
  - Build Server
- HTTP Load Balancer
- Log Server
  - Reporting Server
  - Mail-1
  - Mail-2
  - Primary DNS
  - Secondary DNS
  - Bugzilla
  - File Server
  - Health Monitor
  - Malware Feeds

- Scan Server
- Scan Server
Future work and conclusion
Future Work

• Core
  – Summary-based interprocedural analysis
  – Context sensitive interprocedural analysis
  – Pointer analysis
  – Improved decompilation

• Bug Detection
  – Uninitialised variables
  – Unchecked return values
  – More evaluation and results
• Traditional static analysis can find bugs.

• Decompilation bridges the binary gap.

• Bugwise works on real Linux binaries.

• It is available to use.

• http://www.Bugalyze.com