Reconstructing Control Flow Graphs with Jakstab

Johannes Kinder

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Formal Methods vs. Binaries

Behavioral malware detection

Find vulnerabilities in closed source software

Checking security properties of “apps”, drivers, tools
My Own Story

• Behavioral malware detection
  • *Description logic for malware specifications* [KKSV DIMVA’05][KKSV TDSC’10]
  • 17 May: Tayssir Touili on their follow-up work to this

• Brittle tool chain
  • *Needs a CFG, use IDA Pro?*
  • *IDA Pro chokes on malware*

• If you want something done right, do it yourself!
Jakstab

• Jakstab
  • Initial prototype [KV CAV’08]
  • Sound control flow reconstruction [KVZ VMCAI’09]

• Beyond control-flow graphs
  • Device driver verification [KV FMCAD’10]
  • Integrating traces [KK VMCAI’12]
  • Deobfuscation [K WCRE’12]
• Static CFG Reconstruction
• Jakstab
• Alternating CFG Reconstruction
• Deobfuscating Virtualized Code
Major problem: Indirect jumps

Disassembler

CFG creator

Static analysis

Spec

\[
x = 7
\]

\[
x = 2
\]

\[
x = 3
\]

\[
x = 1
\]

\[
x = -2
\]
Resolving Indirect Jumps

• Use data flow analysis to resolve indirect jumps!
• But: Data flow analysis needs a control flow graph to work on
• Chicken and egg problem?  
  [Theiling00, SchwarzDebrayAndrews02]
Transfer Function

Resolve Decoder + Translator

Continue until fixpoint over abstract states and CFG

Worklist

Spec

Assume (eax = 8)

Assume (eax = 9)

Resolve

CFG

\( pc = 3 \)
\( eax \in [8;9] \)

\( pc = 8 \)
\( eax \in [8;8] \)

\( pc = 9 \)
\( eax \in [9;9] \)
Example – CFG Reconstruction

<table>
<thead>
<tr>
<th>a</th>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if (x ≤ 0) goto 13</td>
</tr>
<tr>
<td>5</td>
<td>x := 1</td>
</tr>
<tr>
<td>10</td>
<td>goto 21</td>
</tr>
<tr>
<td>12</td>
<td>halt</td>
</tr>
<tr>
<td>13</td>
<td>x := 24</td>
</tr>
<tr>
<td>18</td>
<td>x := x – 5</td>
</tr>
<tr>
<td>21</td>
<td>x := x – 1</td>
</tr>
<tr>
<td>24</td>
<td>goto x</td>
</tr>
</tbody>
</table>

Example:

```
 0: if (x ≤ 0) goto 13
 5: x := 1
10: goto 21
12: halt
13: x := 24
18: x := x – 5
21: x := x – 1
24: goto x
```
## Example – CFG Reconstruction

### Data Flow (Transfer Function)

<table>
<thead>
<tr>
<th>$a$</th>
<th>statement</th>
<th>$D(a)(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if ($x \leq 0$) goto 13</td>
<td>$\top$</td>
</tr>
<tr>
<td>5</td>
<td>$x := 1$</td>
<td>$\top$</td>
</tr>
<tr>
<td>10</td>
<td>goto 21</td>
<td>{1}</td>
</tr>
<tr>
<td>12</td>
<td>halt</td>
<td>{12}</td>
</tr>
<tr>
<td>13</td>
<td>$x := 24$</td>
<td>$\top$</td>
</tr>
<tr>
<td>18</td>
<td>$x := x - 5$</td>
<td>{18, 24}</td>
</tr>
<tr>
<td>21</td>
<td>$x := x - 1$</td>
<td>{1, 13, 19}</td>
</tr>
<tr>
<td>24</td>
<td>goto $x$</td>
<td>{0, 12, 18}</td>
</tr>
</tbody>
</table>

### Control Flow Graph (Resolve)

```
assume x > 0
x := 1
assume true
x := x - 5
assume x = 18
x := x - 1
assume x = 12
```

### Fixpoint!

The data flow function $D(a)(x)$ and the control flow graph (CFG) show the dependencies and flows of control and data through the program's execution.

Fixpoint: The values of $x$ at the end of the program execution are determined by the flow of data and control through the CFG.
Theoretical Results

**Correctness:** Every solution computed by the algorithm is an over-approximation of the concrete CFG.

**Termination:** The algorithm terminates in finite time if the domain satisfies ascending chain condition.

**Quality:** The algorithm calculates the most precise CFG with respect to the abstract domain.
Outline

- Static CFG Reconstruction
- Jakstab
- Alternating CFG Reconstruction
- Deobfuscating Virtualized Code
Jakstab

• Software architecture
  • Disassembly + CFG reconstruction + static analysis
  • Combination of abstract domains
  • Written in Java, ca. 40 KLOC

• Target Architectures
  • 32 bit x86, but extensible
    • Extended by others to Intel MCS-96 Microcontroller
  • Windows PE, Linux ELF, raw binaries
Implementation Schema

Composite Analysis
- Analysis 1
- Analysis 2, ...

Fixpoint Algorithm

Transformer Factory

Disassembler

Program Object
- Assembly Map
- Statement Map
- CFG

Architecture Object

Transfer function
- Binary
- Offset
- Instruction

State
- Location
- Statement
- Edges

Resolve

Reached Set

CPAs
- Init Location

Edges

SSL spec

State
- Location
- Instruction

Statement
Available Analyses

• Actively used
  • Bounded Address Tracking
  • Constant Propagation
  • Forward expression substitution

• Experimental
  • Intervals
  • K-Sets
  • Call-stack
Analysis Challenges

• Memory accesses
  • Memory one large array
  • Pervasive pointer arithmetic

• Aliasing
  • All variables may be pointers
  • No types: pointers may alias with whole memory

• Procedures
  • Call and ret instructions treated as jumps
  • No classical interprocedural analysis possible
Partitioned Memory Model

- Memory partitioned into regions
- Addresses are a pair (region, offset)

Regular Integers
- treated as pointers to the global address space
- can be of "type" 1...128 bits
Abstract Address Domain

\( (T_R, T_{32}) \)

Lattice for 32 bit values

Write to target address:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (T_R, T_{32}) )</td>
<td>Entire memory may contain value</td>
</tr>
<tr>
<td>( (\text{region}, T_{32}) )</td>
<td>All locations in region may contain value</td>
</tr>
<tr>
<td>( (\text{region, offset}) )</td>
<td>Single location must contain value</td>
</tr>
</tbody>
</table>

\( (\text{global, } 0) \) \( \ldots \) \( (\text{global, } 1) \) \( \ldots \) \( (\text{stack, } -4) \) \( \ldots \) \( (\text{alloc}_1, 0) \) \( (\text{alloc}_1, 0) \) \( \ldots \) \( \ldots \)
Bounded Address Tracking

• “Bounded” emulation
  • Set of memory configurations for each location
  • Bound on number of values per variable per set
  • Computes over-approximation in finite time

• Stepwise widening
  • First, stop tracking precise offsets
  • If value count keeps increasing, also stop tracking memory region
**BAT – Example**

1. $x := \text{malloc}(10)$
   - $x = (a_1, 0)$
   - $a_1: []$

2. $b := x$
   - $x = (a_1, 0)$
   - $b = (a_1, 0)$
   - $a_1: []$
   - $x = (a_1, 1)$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0)]$
   - $x = (a_1, T_{32})$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$

3. $m_{32}[x] := 0$
   - $x = (a_1, 0)$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0)]$
   - $x = (a_1, 1)$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$
   - $x = (a_1, T_{32})$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$

4. $x := x + 1$
   - $x = (a_1, 1)$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0)]$
   - $x = (a_1, 2)$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$
   - $x = (a_1, T_{32})$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$

5. `assume x ≥ b + 10`
   - $x = (a_1, T_{32})$
   - $b = (a_1, 0)$
   - $a_1: [(g, 0), (g, 0)]$

**Bound per variable per location:**
- Offset: 2
- Region: 2

**Weak update to region:**
- $a_1 ← (g, 0)$
Verifying Device Driver Binaries

- **Objective**
  - “Static Driver Verifier” (SDV) for driver binaries
  - Show feasibility on real world code
  - Compare to state-of-the-art CodeSurfer/x86

- **Analysis Environment**
  - Use SDV's OS model and analysis harness
  - Specifications expressed as assertions in harness
API Usage Rules for Drivers

- API usage rules encoded using state variables and assertions
- Two specifications:

IoAcquire/ReleaseCancelSpinLock-Rule

PendedCompletedRequest-Rule
# Verifying Driver Binaries

<table>
<thead>
<tr>
<th>Driver</th>
<th>States</th>
<th>Instr</th>
<th>Time (s)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>krnldrvr</td>
<td>378</td>
<td>413</td>
<td>2s</td>
<td>OK</td>
</tr>
<tr>
<td>sioctrl</td>
<td>3947</td>
<td>630</td>
<td>7s</td>
<td>OK</td>
</tr>
<tr>
<td>tracedrv</td>
<td>486</td>
<td>439</td>
<td>2s</td>
<td>OK</td>
</tr>
<tr>
<td>cancel-startio</td>
<td>633</td>
<td>759</td>
<td>2s</td>
<td>OK</td>
</tr>
<tr>
<td>cancel-sys</td>
<td>600</td>
<td>780</td>
<td>2s</td>
<td>OK</td>
</tr>
<tr>
<td>moufiltr</td>
<td>3830</td>
<td>722</td>
<td>9s</td>
<td>Spec Violation</td>
</tr>
<tr>
<td>event</td>
<td>663</td>
<td>690</td>
<td>2s</td>
<td>OK</td>
</tr>
<tr>
<td>kbfiltr</td>
<td>3834</td>
<td>726</td>
<td>8s</td>
<td>Spec Violation</td>
</tr>
<tr>
<td>toastmon</td>
<td>4853</td>
<td>977</td>
<td>9s</td>
<td>OK</td>
</tr>
<tr>
<td>diskperf</td>
<td>19772</td>
<td>1409</td>
<td>46s</td>
<td>OK</td>
</tr>
<tr>
<td>fakemodem</td>
<td>13994</td>
<td>1887</td>
<td>24s</td>
<td>Mem Safety Violation</td>
</tr>
<tr>
<td>flpydisk</td>
<td>186543</td>
<td>1782</td>
<td>39m 34s</td>
<td>OK</td>
</tr>
<tr>
<td>mouclass</td>
<td>3055</td>
<td>1763</td>
<td>8s</td>
<td>False Positive</td>
</tr>
<tr>
<td>sermouse</td>
<td>1888</td>
<td>1293</td>
<td>4s</td>
<td>False Positive</td>
</tr>
<tr>
<td>SerEnum</td>
<td>5213</td>
<td>1503</td>
<td>8s</td>
<td>OK</td>
</tr>
</tbody>
</table>

3 GHz Opteron, 4 GB Heap Space
Jakstab – Summary

• Algorithm
  • Integrates disassembly, control flow reconstruction, and static analysis
  • Solves chicken & egg problem, yields optimal CFG

• Implementation
  • Supports multiple analyses, main analysis: Bounded Address Tracking
  • Successfully applied to driver verification

[KV CAV’08] [KVZ VMCAI’09] [KV FMCAD’10]
Outline

• Static CFG Reconstruction
• Jakstab
• Alternating CFG Reconstruction
• Deobfuscating Virtualized Code
Transfer function for assignments / assume

\[ f^\# \left[ m[A] = E \right] \]

\[ f^\# \left[ a = E \right] \]

\( \gamma \left[ E \right] \)

Worklist

Spec

Decoder

Translator

S

S_1

S_2

Transfer Fun

Assume (eax = 8)

Assume (eax = 14)

Resolve

CFG

jmp eax

assume (eax = 8)

assume (eax = 14)
0: x := choice(10,15,20)
3: y := choice(4,6)
6: jmp x
10: y := 1
12: jmp 28
15: y := 2
17: jmp 28
20: x := x + y
22: jmp x
24: jmp 28
26: y := 3
28: return
Degenerate Over-Approximation

0: \( x := \text{hash}(I) \)
3: \( y := \text{choice}(4, 6) \)
6: ©mp \( x \)
10: \( y := 1 \)
12: ©mp 28
15: \( y := 2 \)
17: ©mp 28
20: \( x := x + y \)
22: ©mp \( x \)
24: ©mp 28
26: \( y := 3 \)
28: return

\[ \gamma[x]S = \mathbb{Z} \]
Abstract State

\[ S = (S^#, S^b) \]

Over-approximate

\[ S^# \]

Under-approximate

\[ S^b \]

Concrete values for \( e \)

\[ \gamma^#(e) S^# \]

\[ \gamma^b(e) S^b \]
Alternating Semantics

\[
\chi((S^\#, S^b), x) \quad : \quad \gamma^\#[x]S^\# = \mathbb{Z}
\]

\[
\gamma^\#[x]S^\# = \{24, 26\}
\]

\[
\gamma^b[x]S^b = \{24\}
\]

\[
x := \text{hash}(I)
\]

\[
y := \text{choice}(4,6)
\]

\[
x := x + y
\]

\[
y := 3
\]

\[
\text{return}
\]

\[
\chi(S, e) \text{ controls which approximation is used}
\]
Alternating Semantics

- Parameterized semantics framework
  - Lifts semantics for assignments and assumptions to semantics for jumps that constructs a CFG
  - Instantiated by concrete, under- or over-approximate, or alternating semantics

- Alternating semantics
  - Combines under- and over-approximate semantics
  - Predicate $\chi$ controls which to use for evaluating jump targets
Alternating Semantics

\[
\frac{[m[A] := E]_\ell}{\langle \ell, S, G \rangle \rightarrow \langle \ell', S', G \uplus (\ell, \ell') \rangle}
\]

**Assignment**

\[
\frac{\chi(S, E) \quad (1, V) \in \gamma^b[B, E]S^b \quad f^\circ[\text{assume } B \land E = V](S) = S'}{\langle \ell, S, G \rangle \rightarrow \langle V, S', G \uplus (\ell, V) \rangle}
\]

**Jump-True\(^b\)**

\[
\frac{\neg\chi(S, E) \quad (1, V) \in \gamma^b[B, E]S^b \quad f^\circ[\text{assume } B \land E = V](S) = S'}{\langle \ell, S, G \rangle \rightarrow \langle V, S', G \uplus (\ell, V) \rangle}
\]

**Jump-True\(^b\)**

\[
\frac{0 \in \gamma[B]S \quad f^\circ[\text{assume } \neg B](S) = S'}{\langle \ell, S, G \rangle \rightarrow \langle \ell', S', G \uplus (\ell, \ell') \rangle}
\]

**Jump-False**

\[
\frac{\text{[assume } E\text{]}_\ell \quad f^\circ[\text{assume } E](S) = S'}{\langle \ell, S, G \rangle \rightarrow \langle \ell', S', G \uplus (\ell, \ell') \rangle}
\]

**Assume**

\[
\frac{[\text{halt}]_\ell}{\langle \ell, S, G \rangle \rightarrow \langle \ell, S, G \rangle}
\]

**Halt**
Relative Soundness

- Over-approximates the program restricted to the under-approximations used in reconstruction

Allows proofs over program partitions!
## Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Src</th>
<th>Inst</th>
<th>Trace</th>
<th>b only</th>
<th>(#) only</th>
<th>(\nabla) only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inst</td>
<td>Cvg</td>
<td>FP</td>
</tr>
<tr>
<td>demo1</td>
<td>asm</td>
<td>9</td>
<td>49K</td>
<td>7</td>
<td>78%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>demo2</td>
<td>asm</td>
<td>38</td>
<td>87K</td>
<td>29</td>
<td>76%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>demo3</td>
<td>asm</td>
<td>35</td>
<td>87K</td>
<td>17</td>
<td>49%</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>astar</td>
<td>C++</td>
<td>29645</td>
<td>1.0M</td>
<td>10738</td>
<td>36%</td>
<td>7.2s</td>
</tr>
<tr>
<td>bzip</td>
<td>C</td>
<td>29257</td>
<td>531K</td>
<td>9660</td>
<td>33%</td>
<td>5.3s</td>
</tr>
<tr>
<td>lbm</td>
<td>C</td>
<td>5057</td>
<td>840K</td>
<td>2943</td>
<td>58%</td>
<td>2.2s</td>
</tr>
<tr>
<td>omnetpp</td>
<td>C++</td>
<td>171592</td>
<td>4.6M</td>
<td>30627</td>
<td>18%</td>
<td>27.1s</td>
</tr>
<tr>
<td>milc</td>
<td>C</td>
<td>47382</td>
<td>12.0M</td>
<td>14085</td>
<td>30%</td>
<td>13.4s</td>
</tr>
<tr>
<td>specrand</td>
<td>C</td>
<td>16937</td>
<td>413K</td>
<td>4720</td>
<td>28%</td>
<td>2.9s</td>
</tr>
</tbody>
</table>

**Over-approximation:** Constant propagation only

**Under-approximation:** Single trace per file
Alternating CFR – Summary

• Generalization and formalization of the hybrid disassembly idea
• Under-approximation can allow useful results when over-approximation fails
• Can be combined with directed test generation for improved precision

[KK VMCAI 12]
Outline

- Static CFG Reconstruction
- Jakstab
- Alternating CFG Reconstruction
- Deobfuscating Virtualized Code
Obfuscation

• Hides code from humans and tools
  • Anti-disassembly, anti-analysis, anti-emulation
  • Used for benign and malicious code

• Virtualization Obfuscation
  • Considered one of the strongest obfuscation schemes
  • Uses Virtual Machine to hide real code

• Why is static analysis of virtualization-obfuscated code fundamentally hard?
• Can we use it for deobfuscation?
void foo (int x) {
    int y = 10;
    y++;
    y++;
    if (x > 0) {
        y++;
    } else {
    }
    apiCall(y);
}
Virtualization Obfuscation

```c
void foo (int x) {
    int y = 10;
    y++;
    y++;
    if (x > 0) {
        y++;
    }
    else {}  
    apiCall(y);
}
```

Compile to Bytecode

code = { 52, 01, 02, 05, 01, 03, 01, 08, 00, 06, 03, 01, 18, 01, 00 }
data = { 00, 00, 10, 05 }

Replace Code by Interpreter

conditional jump distance
int vpc = 0, op1, op2;
while (true) {
    switch(code[vpc]) {
        case 03: // increment
            op1 = code[vpc + 1];
            data[op1]++;
            vpc += 2;
            break;
        case 08: // conditional jump
            op1 = code[vpc + 1];
            op2 = code[vpc + 2];
            if (data[op1] <= 0)
                vpc += data[op2]
            else
                vpc += 3;
            break;
        case 18: // call function
            op1 = code[vpc + 1];
            apiCall(data[op1]);
            vpc += 2;
            break;
        case 52: // assignment
            op1 = code[vpc + 1];
            op2 = code[vpc + 2];
            data[op1] = data[op2];
            vpc += 3;
            break;
        default: // halt
            return;
    } // end switch
} // end while
• 1 interpreter case = many original locations
• Interpreter loop head shared among all

Original CFG

Obfuscated CFG

y = 10
y++

x > 0
x <= 0

y++
apiCall(y)

y++

x > 0
x <= 0

y++
apiCall(y)
Domain Flattening

\[ x \in [-\infty; \infty] \]
\[ y \in [-\infty; \infty] \]

\[ x \in [-\infty; 0] \]
\[ y \in [12; 12] \]

\[ x \in [1; \infty] \]
\[ y \in [13; 13] \]

\[ x \in [-\infty; \infty] \]
\[ y \in [10; 10] \]

\[ x \in [-\infty; \infty] \]
\[ y \in [11; 11] \]

\[ x \in [-\infty; \infty] \]
\[ y \in [12; 12] \]

\[ x \in [-\infty; \infty] \]
\[ y \in [-\infty; \infty] \]

Location Sensitive Analysis → Location Insensitive Analysis
VPC Lifting

- Virtualization flattens one dimension of location
- Idea: track VPC and use as additional dimension
  - Separate states with differing VPC values
    
    ```
    int y = 10;     y++;   
    vpc ∈ [0; 0]   
    dy ∈ [0; 0]   
    vpc ∈ [3; 3]   
    dy ∈ [10; 10] 
    ```
  - Join states with equal VPC values
    
    ```
    if (…) {}     else {} 
    vpc ∈ [12; 12] 
    dy ∈ [12; 13]   
    vpc ∈ [12; 12] 
    dy ∈ [13; 13] 
    ```
Each box has unique pair (pc, vpc)
data[1]++;

- Constant propagation
- Dead code elimination
- Jump threading

Each node has unique pair \((pc, vpc)\)

Original CFG
Evaluation

• Implemented in Jakstab
  • *Processes obfuscated binaries*

• Analysis
  • *VPC-lifted variant of Bounded Address Tracking*
  • *Apply value bound per VPC value*

• Targets
  • *Code samples processed with research obfuscator (University of Arizona)*
## Preliminary Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>Similarity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>tamperproof guard</td>
<td>22%</td>
<td>53%</td>
<td>22s</td>
</tr>
<tr>
<td>search tree</td>
<td>24%</td>
<td>44%</td>
<td>198s</td>
</tr>
<tr>
<td>matrix multiply</td>
<td>32%</td>
<td>72%</td>
<td>203s</td>
</tr>
<tr>
<td>stuxnet</td>
<td>26%</td>
<td>72%</td>
<td>190s</td>
</tr>
</tbody>
</table>

- **Similarity**
  - *Naive graph matching algorithm*
  - *Baseline: similarity of obfuscated code to original*
  - *Ongoing work!*
Devirtualization – Summary

• Identified domain-flattening as fundamental difficulty

• Introduced VPC-sensitivity as countermeasure

• Algorithm for reconstruction of original CFG
Summary

• Jakstab
  • Integrates disassembly, CFG reconstruction, and static analysis
  • Solves chicken & egg problem of binary analysis
  • Extensible

• Reconstruct control flow graphs
  • Statically
  • By alternation of static and dynamic analysis
  • From “virtualization obfuscated” code

http://www.jakstab.org