One Engine to Serve’em All: Inferring Taint Rules Without Architectural Semantics

Zheng Leong Chua, Yanhao Wang, Teodora Băluţă, Prateek Saxena, Zhenkai Liang, Purui Su

National University of Singapore
Chinese Academy of Sciences
Importance of Taint Analysis

- Taint analysis tracks the information flow within a program
- Taint analysis is the basis for many security applications
  - Information leakage detection
  - Enforcing CFI
  - Vulnerability detection
  - ...

```c
1 int parse_buffer(char buffer[100], struct pkt_info *info) {
2     char check_flag;

3     check_flag = buffer[5] & 0x16;

4     err = init_pkt_info(info);

5     if (!err)
6         return err;

7     info->flag = check_flag;

8     /* ... */

9     strncpy(info->data, buffer + 6, 50);

10    info->seq = get_current_seq();

11   return OK;
12 }
```
Taint Analysis on Binaries

/* tainted input from network socket */
1 int parse_buffer(char buffer[100], struct pkt_info *info) {
   2   char check_flag;
   3
   4   check_flag = buffer[5] & 0x16;
   5
   6   err = init_pkt_info(info);
   7   if (!err)
   8       return err;
   9   info->flag = check_flag;
  10   /* ... */
  11   strncpy(info->data, buffer + 6, 50);
  12   info->seq = get_current_seq();
  13   return OK;
  14 }

Write binary taint rules based on instruction operational semantics

movsx   eax, byte ptr [rsi + 5]
and    eax, 16
mov    cl, al
mov    byte ptr [rbp - 25], cl

T[check_flag] = T[buffer+5]
Many Faces of Taint Rules

• What is the taint rule for and eax, 16?
• Main instruction semantics: eax = eax & 16

<table>
<thead>
<tr>
<th>Taint Engine 1</th>
<th>Taint Engine 2</th>
<th>Taint Engine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>if imm == 0 { T[eax] = 0 }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complexity of Taint Rules

• Input dependent propagation
• Size dependent propagation
• Architectural quirks for backwards compatibility

if (size == 64 || size == 32 || size == 16) {
    for (x = 0; x < size / 8; x++) {
        if (t1[x] & t2[x]) t1[x] = 1;
        else if (t1[x] and !t2[x])
            t1[x] = t1[x] & op2[x];
        else if (!t1[x] & t2[x])
            t1[x] = t2[x] & op1[x];
        else t1[x] = 0;
    } else if (size == 8) {
        // 0 if it’s lower 8 bits, 1 if it’s upper 8 bits
        pos1 = isUpper(op1); pos2 = isUpper(op2);
        if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
        else if (t1[pos1] & !t2[pos2])
            t1[pos1] = t1[pos1] & op2[pos2];
        else if (!t1[pos1] & t2[pos2])
            t1[pos1] = t2[pos2] & op1[pos1];
        else t1[pos1] = 0;}
    if (mode64bit == 1 and size == 64)
        for (x = 32; x < size; x++) t1[x] = 0;
Contributions

• A new way for representing **taint using influence**
  • Rather than instruction semantics

• An **inductive taint analysis** approach using **probe-and-observe**
  • With minimal architectural knowledge

• Our **tool**, TaintInduce, generates accurate taint rules for four architectures (x86, x64, AArch64, MIPS)
Problem (re-)definition

• Taint is defined as a collection of influence relations which are observed when executing the instruction as a black box.
Direct-Indirect Dependencies Using Influence

Direct dependency
• Same influence relation across all executions

Example: `mov eax, ebx`

Indirect dependency
• Multiple direct dependencies

Example: `mov eax, [ebx]`

Implicit dependency
• Influence relation changes across executions

Example: `cmovb eax, ebx`

**Example Diagrams:**
- Direct dependency:
  - eax → ebx

- Indirect dependency:
  - eax → mem_addr1
  - mem_addr1 → mem_val1
  - mem_val1 → eax

- Implicit dependency:
  - eax → ebx
  - OR
  - eax → eax
Soundness & Completeness

• No over-tainting: soundness

• No under-tainting: completeness

• Very hard to ensure sound and complete
  • Relax the requirements, aim to be useful in practice 😊
### Approach

**Observation**

*Instruction*

- cmovb eax, ebx

**Observation Engine**

**Observations**

- (10110..., 11100)
- (10111..., 11000)

**Inference Engine**

**Rule (Exact)**

- A → B
- X → A
- Y → Z

**Rule (General)**

- A → B
- X → A
- Y → Z
TaintInduce – Exact Mode

• Flip a bit and observe the output for changes.
  • $\Delta EBX_0 \rightarrow \Delta EAX_0$
  • $\Delta EBX_0 \rightarrow \Delta EBX_0$
• Influence (Inf) only valid if:
  • $EAX = 11100011$, $EBX = 00101000$
• Form a truth table with all of the collected observations.
  • True if there is a change, False otherwise
• Unseen values are conservatively set to False

```
mov eax, ebx
```

<table>
<thead>
<tr>
<th>EAX_7</th>
<th>EAX_6</th>
<th>EAX_5</th>
<th>EAX_4</th>
<th>EAX_3</th>
<th>EAX_2</th>
<th>EAX_1</th>
<th>EAX_0</th>
<th>EBX_7</th>
<th>EBX_6</th>
<th>EBX_5</th>
<th>EBX_4</th>
<th>EBX_3</th>
<th>EBX_2</th>
<th>EBX_1</th>
<th>EBX_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EAX_0</th>
<th>EAX_1</th>
<th>…</th>
<th>EBX_0</th>
<th>EBX_1</th>
<th>…</th>
<th>Inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>…</td>
<td>0</td>
<td>0</td>
<td>…</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>…</td>
<td>1</td>
<td>0</td>
<td>…</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>…</td>
<td>1</td>
<td>1</td>
<td>…</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>…</td>
<td>0</td>
<td>0</td>
<td>…</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>0</td>
</tr>
</tbody>
</table>
TaintInduce – Boolean Minimization

• Boolean minimization using ESPRESSO algorithm
• More succinct representation
  • Not a conjunction of all the observed states

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EAX_0 \land EAX_1 \land \ldots$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$EAX_0 \land EAX_1 \land \ldots$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\neg EAX_0 \land \neg EAX_1 \land \ldots$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\neg EAX_0 \land \neg EAX_1 \land \ldots$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>&lt;other observations&gt;</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>&lt;unobserved values&gt;</td>
<td>$\text{False}$</td>
</tr>
</tbody>
</table>

IF

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EAX_0 \land EAX_1 \land \ldots$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\neg EAX_0 \land \neg EAX_1$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\text{True}$</td>
</tr>
</tbody>
</table>

THEN $\left( EBX_0 \rightarrow EAX_0 \right)$
TaintInduce – Generalization Mode

• We carefully trade-off soundness for generalization
  • We allow the Boolean minimization algorithm to pick values for the unseen inputs by setting them to don’t care

<table>
<thead>
<tr>
<th>EAX₀ ^ EAX₁ ^ ...</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX₀ ^ EAX₁ ^ ...</td>
<td>True</td>
</tr>
<tr>
<td>!EAX₀ ^ !EAX₁ ^ ...</td>
<td>True</td>
</tr>
<tr>
<td>!EAX₀ ^ !EAX₁ ^ ...</td>
<td>True</td>
</tr>
<tr>
<td>...</td>
<td>Don’t Care</td>
</tr>
</tbody>
</table>

IF

Don’t Care          True

THEN  (EBX₀ → EAX₀)
Condition Identification – Behavior Grouping

cmovb eax, ebx

State Before

<table>
<thead>
<tr>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>CF</th>
<th>Memory Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

State After

<table>
<thead>
<tr>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>CF</th>
<th>Memory Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cmovb eax

State Before

<table>
<thead>
<tr>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>CF</th>
<th>Memory Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

State After

<table>
<thead>
<tr>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>CF</th>
<th>Memory Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ebx \rightarrow eax</th>
<th>eax \rightarrow eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF=1, EAX=542, EBX=19, ECX=7, ...</td>
<td>CF=0, EAX=12, EBX=4, ECX=1023,...</td>
</tr>
<tr>
<td>CF=1, EAX=32, EBX=3, ECX=0, ...</td>
<td>CF=0, EAX=42, EBX=11, ECX=13, ...</td>
</tr>
<tr>
<td>CF=1, EAX=873, EBX=32, ECX=1, ...</td>
<td>CF=0, EAX=2, EBX=3, ECX=33, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Boolean Minimization

IF

<table>
<thead>
<tr>
<th>CF=1</th>
<th>True</th>
</tr>
</thead>
</table>

THEN (EBX₀ → EAX₀)

ELSE (EAX₀ → EAX₀)
Evaluation

• Coverage and Correctness
  • How many instructions across multiple architectures can TaintInduce learn?

• Exploit Detection for real-world CVEs
  • Is the approach feasible in practice?

• Comparison with other tools
  • Is TaintInduce comparable to existing taint engines?
Coverage and Correctness

TaintInduce never over-taints for 71.51% of the instructions tested across 4 architectures: x86, x64, AArch 64, MIPS-I

Methodology: train for 100 seeds, test on 1000 random inputs for each instruction

<table>
<thead>
<tr>
<th></th>
<th>Arith</th>
<th>Comp</th>
<th>Jump</th>
<th>Move</th>
<th>Cond</th>
<th>FPU</th>
<th>SIMD</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>x64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AArch64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MIPS-I</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Exploit Detection for real-world CVEs

Detected taint at the sink in 24 / 26 of the exploit trace. Of the remaining 2, sink value is derived indirectly from the source.

• 26 CVEs from real-world programs
  • bind, sendmail, wu-ftpd, rpcss, mssql, atphttpd, ntpd, smbd, ghttpd, miniupnp, openjpeg, glibc, libsndfile, gnulib
  • Stack buffer overflows, heap corruption, floating-point division errors, integer divide-by-zero

• Track direct dependencies only similar to other approaches
Comparison with other Tools

Learns rules that propagate identically to existing tools between 93.27% and 99.5%.

<table>
<thead>
<tr>
<th>X86 Instructionsxw</th>
<th>Arith</th>
<th>Comp</th>
<th>Jump</th>
<th>Move</th>
<th>Cond</th>
<th>FPU</th>
<th>SIMD</th>
<th>Misc</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaintInduce</td>
<td>43</td>
<td>9</td>
<td>33</td>
<td>33</td>
<td>60</td>
<td>85</td>
<td>259</td>
<td>28</td>
<td>550</td>
</tr>
<tr>
<td>libdft</td>
<td>15</td>
<td>5</td>
<td>1</td>
<td>30</td>
<td>32</td>
<td>X</td>
<td>X</td>
<td>8</td>
<td>91</td>
</tr>
<tr>
<td>Triton</td>
<td>38</td>
<td>9</td>
<td>19</td>
<td>33</td>
<td>32</td>
<td>X</td>
<td>144</td>
<td>13</td>
<td>288</td>
</tr>
<tr>
<td>TEMU</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>13</td>
</tr>
</tbody>
</table>
Take Aways

• Re-define taint based on observations – propose an inductive approach with minimal architectural knowledge

• Reduces engineering effort and improves usability of taint

• TaintInduce works well in practice, comparable to existing manual tools
Backup Slides
Performance

- 24 hrs for 27 traces using 20 servers.
  - 23 hours for rule inference, 30 mins for taint propagation
- Rule inference time scales linearly with the amount of compute power.
Utility as a cross-referencing tool

• Found 20 bugs in existing taint tools, 17 errors in unicorn, 3 description errors in ISA instruction manuals

• Intel Software Developer’s Manual (bt r16/32, r16/32)
  • Manual states 3 or 5 bits, should be 4 or 5.

• Ambiguous behavior for tzcnt
  • If not support, silently fallback to bsf
Tool Implementation

Observation Engine

Observations

Inference Engine

Insn 1
Insn 2
Insn 3
Insn 4
Insn 5
Insn 6
Insn 7
...

Rule 1
Rule 2
Rule 3
Rule 4
Rule 5
Rule 6
Rule 7
...

Soundness & Completeness

• No over-tainting: $R_I(S,T)[j] \Rightarrow \exists i, S \mid T[i] \land ( < I, S, i, j > \in Inf )$

• No under-tainting: $\exists i, S \mid T[i] \land ( < I, S, i, j > \in Inf ) \Rightarrow R_I(S,T)[j]$

• Very hard to ensure sound and complete
  • Relax the requirements, aim to be useful in practice 😊
Inference Engine

• Exact mode – Sound & Complete w.r.t to seen states
Complexity of Creating Taint Rules

Taint rule for `and eax, 16`?

```c
if (size == 64 || size == 32 || size == 16) {
    for (x = 0; x < size / 8; x++) {
        if (t1[x] & t2[x]) t1[x] = 1;
        else if (t1[x] and !t2[x])
            t1[x] = t1[x] & op2[x];
        else if (!t1[x] & t2[x])
            t1[x] = t2[x] & op1[x];
        else t1[x] = 0;
    } else if (size == 8) {
        // 0 if it’s lower 8 bits, 1 if it’s upper 8 bits
        pos1 = isUpper(op1); pos2 = isUpper(op2);
        if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
        else if (t1[pos1] & !t2[pos2])
            t1[pos1] = t1[pos1] & op2[pos2];
        else if (!t1[pos1] & t2[pos2])
            t1[pos1] = t2[pos2] & op1[pos1];
        else t1[pos1] = 0;
    }
}

if (mode64bit == 1 and size == 64)
    for (x = 32; x < size; x++) t1[x] = 0;
```

What if we don’t have instruction manuals at all?