Quantifying Information Leaks via Model Counting Modulo Theories

Sang Phan

Queen Mary University of London

April 28, 2015
Information Flow

Secret input (H)  Public input (L)

Program P

Public Output (O)

Non-interference

Secret input (H)  Public input (L)

Program P

Public Output (O)

Information leaked
Non-interference is unachievable

```c
int check(int H, int L){
    int O;
    if (L == H)
        O = ACCEPT;
    else O = REJECT;
    return O;
}
```

password check

Leaks = Secrecy before observing - Secrecy after observing

\[
\Delta_{E}(X_{H}) = E(X_{H}) - E(X_{H}|X_{O})
\]
Theorem of Channel Capacity

\[ \Delta_E(X_H) \leq \log_2(|O|) \]

- has been proved for Shannon entropy and Rényi’s min-entropy
- holds for all possible distributions of \(X_H\).
- is the basis of state-of-the-art techniques for Quantitative Information Flow analysis.

Definition

Quantitative Information Flow (QIF) is the problem of counting \(N\), the number of possible outputs of a given program \(P\).
O is stored as a bit vector \( b_1 b_2 \ldots b_M \).

Assume we have a first-order formula \( \varphi_P \) such that:

- \( \varphi_P \) contains a set of Boolean variables \( V_I := \{ p_1, p_2, \ldots, p_M \} \)
- \( p_i = \top \) if and only if \( b_i \) is 1, and \( p_i = \bot \) if and only if \( b_i = 0 \)

**Counting outputs of** \( P \equiv \) **Counting models of** \( \varphi_P \) **w.r.t.** \( V_I \)
Model Counting Modulo Theories

SAT \rightarrow \text{generalize to first-order theories} \rightarrow \text{SMT}

\downarrow

generalize to counting models

\#SAT
Model Counting Modulo Theories

SAT

generalize to first-order theories

generalize to counting models

#SAT

generalize to first-order theories

SMT

generalize to counting models

#SMT
DPLL Modulo Theories

\[ \text{DPLL}(\mathcal{T}) = \text{DPLL} + \mathcal{T}\text{-solver} \]

\[ \varphi := \{ \neg(x > 10) \lor A_1 \} \land \{ (x > 10) \lor \neg A_1 \} \land \{ \neg A_3 \lor (x < 1) \} \]

\[ \mathcal{B}A(\varphi) := \{ \neg B_1 \lor A_1 \} \land \{ B_1 \lor \neg A_1 \} \land \{ \neg A_2 \lor B_2 \} \]

\[ \mu^P = A_1 \land B_1 \land A_2 \land B_2 \Rightarrow \mathcal{T}\text{-solver}(\mu) \text{ returns inconsistent.} \]

\[ \mu^P = A_1 \land B_1 \land \neg A_2 \land \neg B_2 \Rightarrow \mathcal{T}\text{-solver}(\mu) \text{ returns consistent.} \]
Two approaches:

- Use formal methods to mimic $\text{DPLL}(\mathcal{T})$.
- Generate $\varphi_P$, then using $\text{DPLL}(\mathcal{T})$. 
for all $i$ from 1 to $M$ do

$b_i = (0 >>> (i - 1)) \& 1$

if ($b_i == 1$) then

$p_i \leftarrow \top$

else

$p_i \leftarrow \bot$

Figure: Program instrumentation to build the set $V_i$

The algorithm consists of two components:

- A procedure to enumerate bit configurations (similar to DPLL)
- A model checker to check the existence of the bit configurations (similar to the $\mathcal{T}$-solver)
assert !(p1 && p2 && p3 && p4 && p5);
A program analysis technique that has several applications, in particular automated test generation.

Executing programs with symbols instead of concrete inputs.
QIF analysis using Symbolic Execution

Symbolic Execution as DPLL Modulo Theories

\[ \text{DPLL}(\mathcal{T}) = \text{DPLL} + \mathcal{T}\text{-solver} \]
\[ \text{Symbolic Executor} = \text{Boolean Executor} + \mathcal{T}\text{-solver} \]

Add conditions to test each bits of the output:

\[
\text{for all } i \text{ from } 1 \text{ to } M \text{ do} \\
b_i = (0 >>> (i - 1)) \& 1 \\
\text{if } (b_i == 1) \text{ then} \\
p_i \leftarrow \top \\
\text{else} \\
p_i \leftarrow \bot
\]

Figure: Program instrumentation to build the set \( V_i \)
(H ≥ 16) and (H < 16): program conditions.

p₁, p₂, ..: additional conditions.
Program transformation

\begin{align*}
L &= 8; \\
\text{if } (H < 16) & \quad O = H + L; \\
\text{else} & \quad O = L;
\end{align*}

\begin{align*}
(L_1 = 8) & \quad \land \\
(G_0 = H_0 < 16) & \quad \land \\
(O_1 = H_0 + L_1) & \quad \land \\
(O_2 = L_1) & \quad \land \\
(O_3 = g?O_1 : 0_3) & \end{align*}

Figure: A simple program encoded into a first-order formula

Formula instrumentation to build the set $V_I$:

\begin{align*}
\text{(assert } (= (= #b1 ((\_ extract 0 0) O_3)) p_1))
\end{align*}
QIF analysis using a SMT solver

Use APIs provided by an SMT solver

**Blocking clause**

After finding a model

\[ \mu = l_0 \land l_1 \land \cdots \land l_m \land \cdots \]

Add the clause:

\[ \text{block} = \neg l_0 \lor \neg l_1 \lor \cdots \lor \neg l_m \]

**Depth-first search**

Two components:

- A DPLL like procedure to enumerate truth assignments.
- Use the SMT solver to check consistency of the truth assignments.
Tools selected:

- Model Checking: CBMC (Ansi C)
- Symbolic Execution: Symbolic PathFinder (Java bytecode)
- Program transformation: CBMC
- SMT solver: z3

Benchmarks include:

- Vulnerabilities in Linux kernel
- Anonymity protocols
- A Tax program from the European project HATS (Java)

Assumptions: all programs have bounded loops, no recursion.
Some of the experiments:

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Policy</th>
<th>LoC</th>
<th>sqifc time</th>
<th>selfcomp time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Sanitization</td>
<td>-</td>
<td>&lt; 10</td>
<td>11.898</td>
<td>timed out</td>
</tr>
<tr>
<td>CVE-2011-2208</td>
<td>64</td>
<td>&gt; 200</td>
<td>22.759</td>
<td>119.117</td>
</tr>
<tr>
<td>CVE-2011-2208</td>
<td>256</td>
<td></td>
<td>88.196</td>
<td>timed out</td>
</tr>
<tr>
<td>CVE-2011-1078</td>
<td>8</td>
<td>&gt; 200</td>
<td>10.380</td>
<td>13.853</td>
</tr>
<tr>
<td>CVE-2011-1078</td>
<td>64</td>
<td></td>
<td>37.899</td>
<td>timed out</td>
</tr>
<tr>
<td>CRC</td>
<td>8</td>
<td>&lt; 30</td>
<td>1.209</td>
<td>0.498</td>
</tr>
<tr>
<td>CRC</td>
<td>32</td>
<td></td>
<td>8.657</td>
<td>timed out</td>
</tr>
</tbody>
</table>

**Figure**: Times are in seconds, timeout is 30 minutes. In the first case study, “-” means the policy is not specified.
Some of the experiments:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Leaks</th>
<th><strong>sqifc time</strong></th>
<th>CBMC</th>
<th><strong>sqifc++ time</strong></th>
<th>aZ3</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data sanitization</td>
<td>4</td>
<td>11.898</td>
<td>0.165</td>
<td>0.086</td>
<td></td>
<td>0.251</td>
</tr>
<tr>
<td>Implicit flow</td>
<td>2.81</td>
<td>5.033</td>
<td>0.169</td>
<td>0.049</td>
<td></td>
<td>0.218</td>
</tr>
<tr>
<td>Population count</td>
<td>5.04</td>
<td>17.278</td>
<td>0.162</td>
<td>0.398</td>
<td></td>
<td>0.560</td>
</tr>
<tr>
<td>Mix and duplicate</td>
<td>16</td>
<td>-</td>
<td>0.154</td>
<td>136.947</td>
<td></td>
<td>137.101</td>
</tr>
<tr>
<td>Masked copy</td>
<td>16</td>
<td>-</td>
<td>0.175</td>
<td>18.630</td>
<td></td>
<td>18.805</td>
</tr>
<tr>
<td>Sum query</td>
<td>4.81</td>
<td>64.557</td>
<td>0.162</td>
<td>0.133</td>
<td></td>
<td>0.295</td>
</tr>
<tr>
<td>Ten random outputs</td>
<td>3.32</td>
<td>64.202</td>
<td>0.160</td>
<td>0.093</td>
<td></td>
<td>0.253</td>
</tr>
<tr>
<td>CRC (8)</td>
<td>3</td>
<td>2.551</td>
<td>0.184</td>
<td>0.099</td>
<td></td>
<td>0.283</td>
</tr>
<tr>
<td>CRC (32)</td>
<td>5</td>
<td>7.755</td>
<td>0.193</td>
<td>0.325</td>
<td></td>
<td>0.518</td>
</tr>
</tbody>
</table>

Figure: Leaks are in bits. aZ3 runs with the DFS-based algorithm. Times are in seconds, “-” means timeout in one hour. Total time of sqifc++ is the sum of CBMC time and aZ3 time.
Conclusions

Program transformation

\[ P \quad \text{program transformation} \quad \varphi_P \]

QIF

\#SMT

Formal methods

DPLL(\(\mathcal{T}\))

Two approaches:

- Use formal methods to mimic DPLL(\(\mathcal{T}\)).
  - QIF analysis using Model Checking.
  - QIF analysis using Symbolic Execution.
- Generate \(\varphi_P\), then using DPLL(\(\mathcal{T}\)).
  - Generate \(\varphi_P\) using program transformation.
  - Extend an SMT solver for \#SMT.
THANK YOU FOR YOUR ATTENTION!