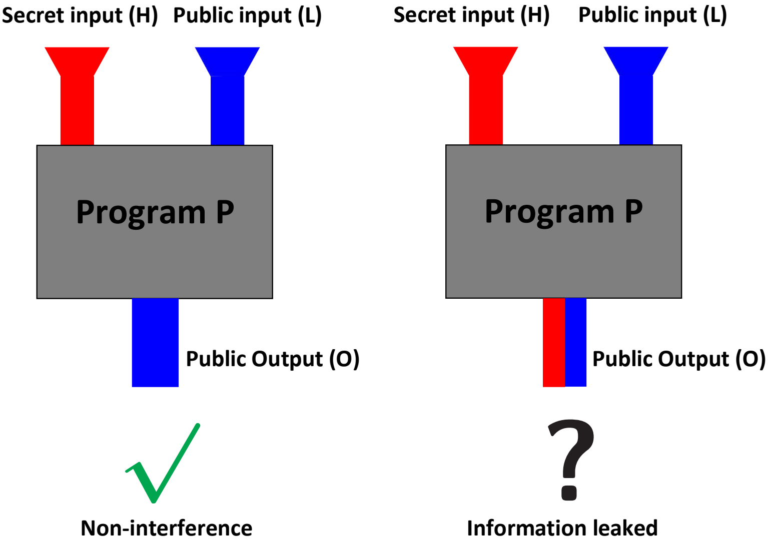
Quantifying Information Leaks via Model Counting Modulo Theories

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)

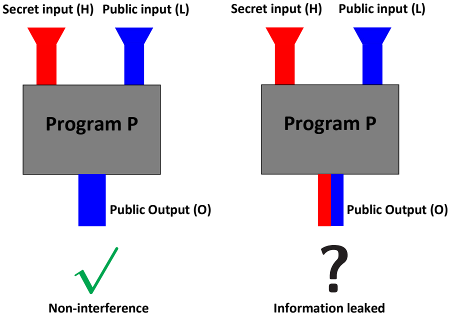
?

Information leaked

Non-interference is unachievable

```
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

$$\Delta_E(X_H) = E(X_H) - E(X_H|X_O)$$

Quantifying Information Leaks

Theorem of Channel Capacity

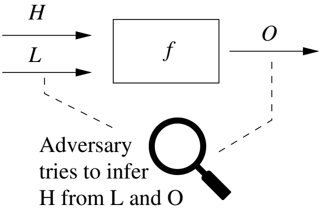
$$\Delta_E(X_H) \leq \log_2(|O|)$$

- 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.

Quantifying as Counting



- O is stored as a bit vector $b_1 b_2 \dots b_M$.
- Assume we have a first-order formula φ_P such that:
    - φ_P contains a set of Boolean variables $V_I := \{p_1, p_2, .., p_M\}$
    - $p_i = \top$ 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 φ_P w.r.t. V_I

Model Counting Modulo Theories

$$\begin{tikzcd}
\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] \arrow[d, "\text{generalize to counting models}"] & \text{SMT} \\
\#\text{SAT} &
\end{tikzcd}$$

Model Counting Modulo Theories

$$\begin{tikzcd}
\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] \arrow[d, "\text{generalize to counting models}"] & \text{SMT} \arrow[d, "\text{generalize to counting models}"'] \\
\#\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] & \#\text{SMT}
\end{tikzcd}$$

Lazy Satisfiability Modulo Theories

DPLL Modulo Theories

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

$$\begin{aligned} \varphi := \{&\neg(x > 10) \vee A_1\} \wedge \\ \{&(x > 10) \vee \neg A_1\} \wedge \\ \{&\neg A_3 \vee (x < 1)\} \end{aligned} \qquad \begin{aligned} \mathcal{BA}(\varphi) := \{&\neg B_1 \vee A_1\} \wedge \\ \{&B_1 \vee \neg A_1\} \wedge \\ \{&\neg A_2 \vee B_2\} \end{aligned}$$

$\mu^P = A_1 \wedge B_1 \wedge A_2 \wedge B_2 \Rightarrow \mathcal{T}$-solver($\mu$) returns **inconsistent**.
$\mu^P = A_1 \wedge B_1 \wedge \neg A_2 \wedge \neg B_2 \Rightarrow \mathcal{T}$-solver($\mu$) returns **consistent**.

QIF as #SMT

$$\begin{tikzcd}
\texttt{P} \arrow[r] & \varphi_P \\
\text{QIF} & \#\text{SMT} \\
\text{Formal methods} & \text{DPLL}(\mathcal{T})
\end{tikzcd}$$

Two approaches:

- Use formal methods to mimic DPLL($\mathcal{T}$).
- Generate φ_P, then using DPLL($\mathcal{T}$).

QIF analysis using Model Checking

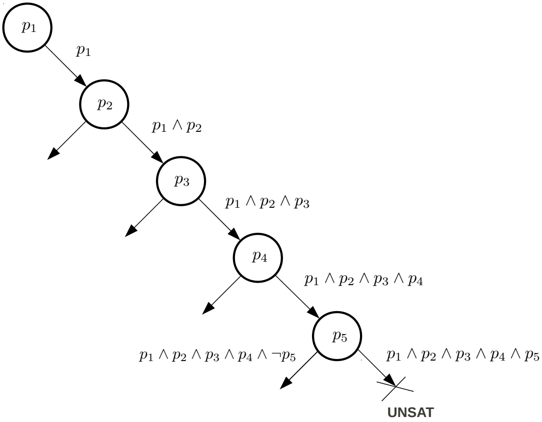
```
for all $i$ from 1 to $M$ do
    $b_i = (O >> (i - 1))\ \&\ 1$
    if $(b_i == 1)$ then
        $p_i \leftarrow \top$
    else
        $p_i \leftarrow \bot$
```

Figure : Program instrumentation to build the set V_I

The algorithm consists of two components:

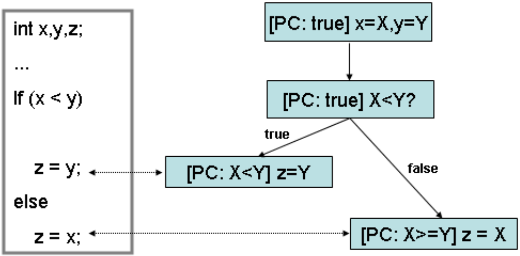
- A procedure to enumerate bit configurations (similar to DPLL)
- A model checker to check the existence of the bit configurations (similar to the $\mathcal{T}$-solver)

QIF analysis using Model Checking



assert !(p_1 && p_2 && p_3 && p_4 && p_5);

Symbolic Execution



- 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

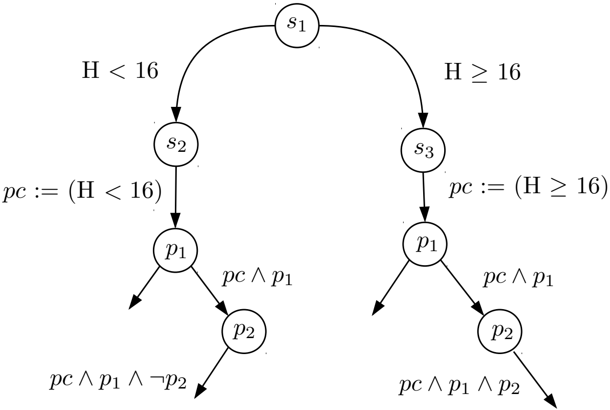
$$\begin{array}{lcl} \text{DPLL}(\mathcal{T}) & = & \text{DPLL} + \mathcal{T}\text{-solver} \\ \text{Symbolic Executor} & = & \text{Boolean Executor} + \mathcal{T}\text{-solver} \end{array}$$

Add conditions to test each bits of the output:

```
for all $i$ from 1 to $M$ do
    $b_i = (\texttt{O} >> (i - 1))\ \&\ 1$
    if $(b_i == 1)$ then
        $p_i \leftarrow \top$
    else
        $p_i \leftarrow \bot$
```

Figure : Program instrumentation to build the set V_I

QIF analysis using Symbolic Execution



- $(H \geq 16)$ and $(H < 16)$: program conditions.
- $p_1, p_2, ..$: additional conditions.

QIF analysis using a #SMT solver

Program transformation

```
L = 8;
if (H < 16)
   O = H + L;
else
   O = L;
```

$$
\begin{array}{ll}
(L_1 = 8) & \wedge \\
(G_0 = H_0 < 16) & \wedge \\
(O_1 = H_0 + L_1) & \wedge \\
(O_2 = L_1) & \wedge \\
(O_3 = g ? O_1 : \mathrm{O}_3) &
\end{array}
$$

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

Formula instrumentation to build the set V_I:

`(assert (= (= #b1 ((_ extract 0 0)` O_3`))` p_1`))`

QIF analysis using a #SMT solver

Use APIs provided by an SMT solver

Blocking clause

After finding a model

$$\mu = l_0 \wedge l_1 \wedge \cdots \wedge l_m \wedge \ldots$$

Add the clause:

$$\texttt{block} = \neg l_0 \vee \neg l_1 \vee \cdots \vee \neg l_m$$

Depth-first search

Two components:

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

Implementation

- 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.

Evaluation

Some of the experiments:

Case Study	**Policy**	**LoC**	**sqifc** time	**selfcomp** time
Data Sanitization	-	< 10	11.898	timed out
CVE-2011-2208	64	> 200	22.759	119.117
CVE-2011-2208	256		88.196	timed out
CVE-2011-1078	8	> 200	10.380	13.853
CVE-2011-1078	64		37.899	timed out
CRC	8	< 30	1.209	0.498
CRC	32		8.657	timed out

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

Evaluation

Some of the experiments:

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

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

$$\mathtt{P} \xrightarrow{\text{program transformation}} \varphi_P$$

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 φ_P, then using DPLL($\mathcal{T}$).
  - Generate φ_P using program transformation.
  - Extend an SMT solver for #SMT.

THANK YOU FOR YOUR ATTENTION!