Quantitative Information Flow as Network Flow Capacity

Stephen McCamant, Michael D. Ernst

Presented by: Bryan Ta
Quantitative Information Flow

• Belief-based approach
• The paper: network flow capacity approach to measure how much information about the “secret inputs” can be inferred from the “public outputs”
• Tainting approach:
  – good at detecting illegal flow
  – cannot give a precise measurement of secret information

Key Ideas of the solution

• Information-flow = a network flow capacity.
  – Information channels = a network of limited-capacity pipes
  – Secret information = an incompressible fluid.
  -> Amount of secret information can be revealed = maximum flow through the network
Flow graph construction

• Edges represent values
  – capacities = # bits of data they can hold.
• Nodes represent basic operations
  – in-degree of a node = the operation’s arity.
• A source node = all secret inputs
• A sink node = all public outputs
• Directed and acyclic graph
  – Edges from older nodes to newer nodes

Implicit flows

• Indirect flows are caused by branches, pointers, arrays
  – Affect later execution
• Each implicit flow operation as part of a larger computation with defined outputs.
• Incompletely treated in dynamic taint analysis
Enclosure regions

- Mark a single-exit control-flow region
- Declare locations the enclosed code might write to
- Specified by annotations, Inferred using static analysis

- Each enclosure region = a distinguished node,
- Edges from implicit flow operations to the enclosure node
- Edges from that enclosure node to outputs.

Example

- Implicit flows:
  - input buffer and num dot
  - num dot and common
  - num and the output

```c
/* Print all the "."s or ","s, whichever is more common. */
void count_punct(char *buf) {
    unsigned char num_dot = 0, num_qm = 0, num;
    char common, *p;
    ENTER_ENCLOSE(num_dot, num_qm);
    while (p = buf; *p != '\0'; p++)
        if (*p == '.')
            num_dot++;
        else if (*p == ',')
            num_qm++;
    LEAVE_ENCLOSE();
    ENTER_ENCLOSE(common, num);
    if (num_dot > num_qm) {
        /* ","s were more common. */
        common = '.'; num = num_dot;
    } else {
        /* "."s were more common. */
        common = ','; num = num_qm;
    }
    LEAVE_ENCLOSE();
    /* print "num" copies of "common". */
    while (num--)
        printf("%c", common);
}
```
Bit-capacity analysis

• Each edge is labeled with a bound
• Edge capacity (the bound) = its value’s number of secret bits
  – Using dynamic tainting at bit level
• Capacity of implicit flow edges = number of possible different executions:
  – E.g. a two-way branch capacity = 1 bit

\[
\begin{align*}
s_{a\oplus b} &= s_a \mid s_b \\
&s_{a\&b} = (s_a \mid s_b) \& (v_a \mid v_b) \\
&s_{a\oplus b} = s_a \mid s_b \mid (((v_a \& s_a) + (v_b \& s_b)) \oplus ((v_a \& s_a) + (v_b \& s_b))) \\
&s_{a\cdot b} = s_a \mid s_b \mid -(s_a \mid s_b)
\end{align*}
\]

01010111 \oplus 0000?_1?_0?_1?_1 = 0 1 0 1 ?_1?_1?_0
01010111 \& 0000?_1?_0?_1?_1 = 0 0 0 0 0 ?_0?_1
01010111 + 0000?_1?_0?_1?_1 = 0 1 ?_1?_0?_0?_1?_0
01010111 \cdot 0000?_1?_0?_1?_1 = ?_1?_1?_1?_1?_1?_1?_1
Example

- reveals 9 bits the secret input:
  - 1 bit of which character is more common,
  - 8 bits from the count.
- Without implicit flows and enclosure regions:
  - unsound result.
  - a leak of 1 bit each time a value from the input buffer was compared to a constant

```c
1 /* Print all the "."s or "]"s, whichever is more common. */
2 void count_punct(char *buf) {
3     unsigned char num_dot = 0, num_qm = 0, num;
4     char common, *p;
5     ENTER_ENCLOSE(num_dot, num_qm);
6     while (*p = buf; *p != '\0'; p++)
7         if (*p == '.')
8             num_dot++;
9         else if (*p == ']')
10             num_qm++;
11     LEAVE_ENCLOSE();
12     ENTER_ENCLOSE(common, num);
13     if (num_dot > num_qm) {
14         /* "."s were more common. */
15         common = "."; num = num_dot;
16     } else {
17         /* "]"s were more common. */
18         common = "]"; num = num_qm;
19     }
20     LEAVE_ENCLOSE();
21 /* print "num" copies of "common". */
22 while (num--)
23     printf("%c", common);
24 }
```

Soundness

- “A bound of \( k \) bits is sound if an adversary could have communicated the same information by sending a \( k \)-bit message directly”
- Kraft’s inequality for soundness:
  \[
  \sum_i 2^{-k(i)} \leq 1.
  \]
- A flow of 0 bits = non-interference
Consistency over multi runs

• Combines the graphs from multiple executions and analyzes together.
• Merges all the edges at the “same” program location into a single edge
  – capacity = sum of the original capacities
• Done in almost-linear time

Pseudo code for combining graphs

```plaintext
// Disjoint sets containing either original graph nodes or pairs of locations and SOURCE or TARGET markers */
U = UnionFind uf;
enum EndpointType { SOURCE, TARGET };

/* One edge class for each code location */
let destinationcombined; /* Total capacity of all the edges in a class */
let total, n = 0;

/* Input loop: collect edges and sum capacities */
for each original edge (u, v) in location, capacity
  combined.add(location); total[location] += total[location] + capacity;
uf.union(u, location, SOURCE);
uf.union(v, location, TARGET);
}

/* Construct representative for each set of merged nodes */
Map<UnionFindSet, Node> node; /* for each set x in uf.sets */
  node[s] := fresh_label();
}
node[uf.find(source)] := source;
node[uf.find(sink)] := sink;

/* Output combined edges in terms of merged nodes */
for each loc in combined {
  Node x := node[uf.find([loc, SOURCE])];
  Node y := node[uf.find([loc, TARGET])];
  output_edge(x, y, total[loc]);
}
```
Implementation

- Static analysis + dynamic enforcement
- Flowcheck
- Valgrind framework

Graph construction by value tagging

- Representation of nodes = Tags
  - a tag with each secret value
  - Maintained in parallel with the secrecy bit-masks
- Representation of edges
  - every edge is unique,
    - no need in-memory representation.
    - the memory usage does not grow as the graph becomes larger
  - edges are combined
    - combined Graph
Pseudo code for graph construction

```c
3  Node regNode[NUM_REGISTERS]; /* Node identities for each register */
4  Node memNode[2*sizeof(int)]; /* Node identities for each memory byte */
5  Node public, source, sink;
6  Node leak = newNode(); /* Implicit flows go here */
7
8  for each instruction "insn" {
9      switch (insn) {
10          case "r_j = r_i";
11              regNode[i] = public;
12          case "r_j = r_i";
13              regNode[i] = regNode[j];
14          case "r_j = r_i + r_k";
15              regNode[i] = regNode[j];
16          case "r_j = r_i + r_k";
17              Node n = newNode();
18              newEdge(regNode[j], n);
19              newEdge(regNode[i], n);
20              Node m = newNode();
21              newEdge(n, m);
22              regNode[i] = m;
23          case "if r_j > 0 goto L";
24              newEdge(regNode[j], leak);
25          case "r_j = load r_a";
26              Node n = newNode();
27              for each b (0 .. 3)
28                  newEdge(lookupNode(regs[a] + b), n);
29              newEdge(regNode[v], m);
30          case "store r_a, r_v"
31              for each b (0 .. 3)
32                  newNode(regs[a] + b) = regNode[v];
33          case "input r_i"
34              Node n = newNode();
35              newEdge(source, m);
36              regNode[i] = n;
37          case "output r_j"
38              Node n = newNode();
39              newEdge(regNode[j], m);
40              newEdge(source, m);
41              newEdge(n, sink);
42              Node l = newNode();
43              newEdge(l, leak);
44              leak := l;
45          }
46      execute insn;
47  }
```

Pseudo code for graph construction

```c
3  Node regNode[NUM_REGISTERS];
4  Node memNode[2*sizeof(int)];
5  Node public, source, sink;
6  Node leak = newNode();
7  for each instruction "insn" {
8      switch (insn) {
9          case "r_j = r_i"
10              regNode[i] = public;
11          case "r_j = r_i"
12              regNode[i] = regNode[j];
13          case "r_j = r_i + r_k"
14              regNode[i] = regNode[j];
15          case "r_j = r_i + r_k"
16              Node n = newNode();
17              newEdge(regNode[j], n);
18              newEdge(regNode[i], n);
19              Node m = newNode();
20              newEdge(n, m);
21              regNode[i] = m;
22          case "if r_j > 0 goto L"
23              newEdge(regNode[j], leak);
24          case "r_j = load r_a"
25              Node n = newNode();
26              for each b (0 .. 3)
27                  newEdge(lookupNode(regs[a] + b), n);
28              newEdge(regNode[v], m);
29          case "store r_a, r_v"
30              for each b (0 .. 3)
31                  newNode(regs[a] + b) = regNode[v];
32          case "input r_i"
33              Node n = newNode();
34              newEdge(source, m);
35              regNode[i] = n;
36          case "output r_j"
37              Node n = newNode();
38              newEdge(regNode[j], m);
39              newEdge(source, m);
40              newEdge(n, sink);
41              Node l = newNode();
42              newEdge(l, leak);
43              leak := l;
44          }
45      execute insn;
46  }
```
Optimizations

Optimizing large-region operations
• The output can be a large data structure,
  – too slow if modifying tag individually
• Use a limited-size set of region descriptors
  – 10 contiguous memory locations
  – a list of 30 addresses excepted.
  – > 30 exceptions, shrunk or eliminated.
• Record Operations by modifying the descriptors

Optimizations

Graph collapsing by code location
• Flow graphs are very large and well-structured
• Simplified by combining edges with the same code location (same implementation as edges labelling)
Checking Future Runs

• A cut = set of edges whose removal disconnects the source from the sink.
• Classic max-flow-min-cut theorem:
  “The value of any flow is bounded by the capacity of any cut, and the maximum flows are those with the same value as the minimum-capacity cuts”

2 checking techniques

Tainting-based checking
  – no secret information reaches the output other than across a given cut
  – reuse the bit-level tainting analysis

Output-comparison checking
  – uninstrumented
• What are dis/adv of this work as compared to belief-based information flow approach?
• Is the tool practical? What circumstance?

Limitations

• say nothing about other possible executions (testing and debugging purpose only)
• Soundness without precision
• which parts of the output contain secret information?
Motivation

- How to implement: “Data influenced x or more bits by the network may not be loaded into the program counter”? 
- Dynamic taint analysis: “data derived from the network may not be loaded into the program counter”
  - binary attribute: either tainted or untainted
Channel Capacity

- Information theory: maximum amount of information a channel can transmit
- Information flow analysis: maximum flow over all distributions of input
- Contributions:
  - Measure channel capacity (influence)
  - Distinguish between true attacks and false positives produced by dynamic taint analysis

Dynamic Taint Analysis Limitations

- False positives: remove danger from untrusted data without removing their taint.

Example 1 Sanity check

```plaintext
if t < 16 then
  v := base + 1
else
  v := base
end if
```

Example 2 Arithmetic restriction: mask

```plaintext
v := base + (t & 0x0f)
```
Dynamic Taint Analysis Limitations

• False negatives: implicit flows

<table>
<thead>
<tr>
<th>Example 3 Table lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>V := table[I]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 4 Implicit flow via branching</th>
</tr>
</thead>
<tbody>
<tr>
<td>if I == 0 then</td>
</tr>
<tr>
<td>V := 0</td>
</tr>
<tr>
<td>else if I == 1 then</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>else if I == 255 then</td>
</tr>
<tr>
<td>V := 255</td>
</tr>
<tr>
<td>end if</td>
</tr>
</tbody>
</table>

Influence for better dynamic taint analysis

• Influence: more graduated attr > binary attr

program as a deterministic computation:

\[ P(I, I_{aux}) \rightarrow V. \]

I : possibly malicious inputs (e.g. data read from the network),
I_{aux} : all other inputs, I_{aux} = c.

• The feasible value set of V = the set of all values assigned to V by P.
• The quantitative influence of I over V = log 2 of the size of the feasible value set.
• Provide more accurate information than taint analysis
Example 1 Sanity check
if $I < 16$ then
    $V := \text{base} + I$
else
    $V := \text{base}$
end if

Feasible value set: $[\text{base, base} + 15]$, Quantitative influence: 4 bits

Example 2 Arithmetic restriction: mask
$V := \text{base} + (I \& 0x0f)$

Example 3 Table lookup
$V := \text{table}[I]$

Quantitative influence: logarithm of the number of unique values in the table

Example 4 Implicit flow via branching
if $I == 0$ then
    $V := 0$
else if $I == 1$ then
    ...
else if $I == 255$ then
    $V := 255$
end if

Feasible value set: 256
The influence: 8 bits

Measuring Quantitative Influence

• Key idea:
  – Model program as a predicate over $I$, $I_{aux}$ and $V$
  – Use a decision procedure to reason about $\{V\}$ for which the predicate is satisfied.
Measuring Quantitative Influence

• Step 1: convert the program to a logic predicate
  – Binary -> IR
  – calculate the weakest precondition over the program.
  – quantifier-free first-order logic.
• Step 2: pose a series of queries to a decision procedure (STP) to learn about the feasible value set.
• Step 3: calculate the quantitative influence from the feasible value set.

Step 1: program to a logic predicate

For large programs:
Compute the influence over a single control-flow path.
  - An execution trace (TEMU) -> a straight-line program
  - Feasible value set: a subset of the whole program’s
  - Quantitative influence: lower bound
Step 2: Predicate to feasible value set

• The decision procedure: STP
  – Input: (from step 1) the predicate in a quantifier-free first order logic over bit vectors and arrays.
  – Output:
    • The predicate is satisfiable?
    • A variable assignment to satisfy the predicate.

Building blocks:
• Point feasibility query:
  – Check whether a particular value is feasible or not.
• Range feasibility query:
  – Check whether a feasible value exists in a range or not
• Exhaust-up-to-c:
  – Range feasibility query c time until there are no more feasible values.
• Density sampling:
  – Use point feasibility queries with random points within some range
• Probabilistic model counting:
  – Add k random parity constraints and check whether the augmented model is still feasible.
Step 3: Feasible value set to quantitative influence

- Logarithm of the total number of feasible points.

**Evaluation**

```c
int implicit(int input)
{
  int output = 0;
  if (input == 0) output = 0;
  else if (input == 1) output = 1;
  else if (input == 6) output = 6;
  else output = 0;
  return output;
}
```

*Figure 1. Implicit flow*

```c
int popcount(unsigned int i) {
  i += (i & 0x55555555) + ((i >> 1) & 0x55555555);
  i += (i & 0x33333333) + ((i >> 2) & 0x33333333);
  i += (i & 0x0f0f0f0f) + ((i >> 4) & 0x0f0f0f0f);
  i += (i & 0x00ff00ff) + ((i >> 8) & 0x00ff00ff);
  return (i > 0) & 0xfff;
}
```

*Figure 2. Population count*

```c
unsigned int mix_copy(unsigned int x) {
  unsigned y = ((x >> 63) * x) & 0xffffffff;
  return y | (y << 1);
}
```

*Figure 3. Mix and duplicate*
Evaluation

Pros and Cons

• Pros:
  – sound and precise (vs. network flow capacity approach)
  – No assumption about input probability distribution
  – Distinguish false positive and false negative in taint analysis
  – Post-analysis step

• Cons:
  – slow even when considering only one execution path
  – Deterministic programs.