Information Flow

Yue Duan

INFORMATION FLOW

1. Introduction to information flow
2. How to track information flow
3. Paper presentation

Quantitative Information Flow as Network Flow Capacity

by Stephen McCamant, Michael D. Ernst

PLDI’08

Introduction to Information Flow

From Wiki,

Information flow in an information theoretical context is the transfer of information from a variable x to a variable y in a given process. Not all flows may be desirable.

Introduction to Information Flow

Explicit flow: explicitly leak information to a publicly observable variable

```
int a = 0;
int b = a;
```

Should be handled relatively easily.

Introduction to Information Flow

Implicit flow: leakage of information through the program control flow.

```
if (a == 3)
    b = 5;
else
    b = 7;
```

Practically more difficult to handle.

How to track information flow

Dynamic taint analysis: run a program and observe which computations are affected by predefined taint sources such as user input.

First mark input data from untrusted sources tainted, then monitor program execution to track how the tainted attribute propagates.

How to track information flow

Example of taint logic: (From TaintDroid)

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

How to track information flow

Potential shortcomings:

1. Overtainting problem
   - Tainting may explode, rendering taint analysis useless.
2. No quantitative measurement
3. Weak in implicit flow handling

Paper presentation

Quantitative Information Flow as Network Flow Capacity

by Stephen McCamant, Michael D. Ernst

PLDI’08

Goal: Determine how much information about a program’s secret inputs is revealed by its public outputs.

Motivation

Tainting approach:
- good at detecting illegal flow
- cannot give a precise measurement of secret information

Motivation

- Subset of inputs are *secret*.
- Subset of outputs are *public*.
- Express confidentiality as a limit on number of secret bits revealed in public outputs.
- Goal: Develop scheme for dynamic quantitative information flow analysis

Dynamic Max-Flow analysis

Key idea of the solution:

Information-flow = a network flow capacity

- Information channels = a network of limited-capacity pipes
- Amount of secret information can be revealed = maximum flow through the network

Dynamic Max-Flow analysis

Flow Graph Construction

- Edges represent values
  - capacities = # bits of data they can hold.
- Nodes represent basic operations
- A source node = all secret inputs
- A sink node = all public outputs
- Directed and acyclic graph

Dynamic Max-Flow analysis

To limit the potential information flow, new node is added.

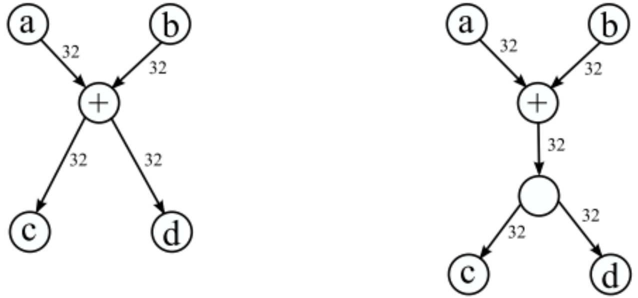


Figure 1. Two possible graphs representing the potential information flow in the expression `c = d = a + b`, where each variable is a 32-bit integer. The graph on the left permits 32 bits of information to flow from `a` to `c`, and a different 32 bits to flow from `b` to `d`. To avoid this, our tool uses the graph on the right.

Dynamic Max-Flow analysis

- **Implicit Flow** are caused by branches, pointers, arrays.
- Each implicit flow operation as part of a larger computation with defined outputs.
- Edges are added to connect each implicit flow operation to the outputs of the enclosed computation.

Dynamic Max-Flow analysis

Consider computing a square root. If the square root is computed by code that uses a loop or branches on the secret value, these implicit flows can be conservatively accounted for by assuming that they might all affect the computed square root value

Dynamic Max-Flow 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

Dynamic Max-Flow analysis

Example of enclosure regions.

- Edges from implicit flow operations to the enclosure node and from that enclosure node to outputs.

```
/* 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);
}
```

Figure 2. C code to print all the occurrences of the most common punctuation character is a string. For instance, when run on its own source code, the program produces the output "........". As detailed in Section 2.4, our tool reports that this execution reveals 9 bits of information about the input.

Dynamic Max-Flow analysis

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

```
/* 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);
}
```

Figure 2. C code to print all the occurrences of the most common punctuation character is a string. For instance, when run on its own source code, the program produces the output "........". As detailed in Section 2.4, our tool reports that this execution reveals 9 bits of information about the input.

Dynamic Max-Flow analysis

Edge capacity

- 2-way branch: add edge with a 1-bit capacity
- Pointer op: add edge with capacity equal to number of secret bits

```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);
}
```

Figure 2. C code to print all the occurrences of the most common punctuation character is a string. For instance, when run on its own source code, the program produces the output "........". As detailed in Section 2.4, our tool reports that this execution reveals 9 bits of information about the input.

Dynamic Max-Flow analysis

Reveals 9 bits the secret input:

- 1 bit of which character is more common
- 8 bits from the count

```
/* 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);
}
```

Figure 2. C code to print all the occurrences of the most common punctuation character is a string. For instance, when run on its own source code, the program produces the output "........". As detailed in Section 2.4, our tool reports that this execution reveals 9 bits of information about the input.

Soundness and Consistency

Soundness:

A bound of *k* is *sound* iff there is also a code c where for each message i, Alice and Bob could have communicated i using exactly *k* bits.

Soundness and Consistency

- Assume Divide(a,b) returns c = a/b
- Alice controls inputs a,b
- Bob sees public output c
- a=2,b=0 for “Attack”
- a=4,b=1 for “No attack”
- Code c: 1 Attack, 0 No attack
- 1 bit bound is **sound**

Soundness and Consistency

Consistency over multiple executions

- 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

Implementation

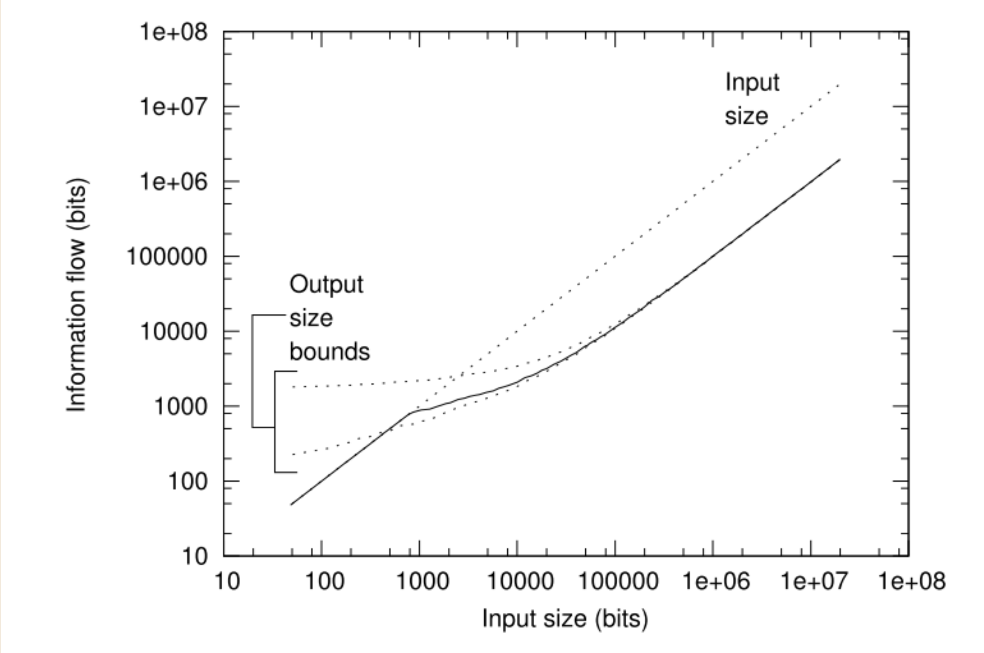
- Dynamic Instruction rewriting via Valgrind.
- Associate positive integer **tags** with any values that could contain secret information
  - —Registers, each byte in memory gets a tag
- —Tag == 0 means no secret information, not necessary to include in graph

Efficient Max Flow

- Solving for maximum flow takes *O(VE)*
  - —V = # of vertices
  - —E = # of edges
- Plan: Linear in actual program runtime
- Solution: Collapse edges, nodes to shrink graph size

Efficient Max Flow

- Performance



Checking Flow Bound

- A cut = set of edges whose removal disconnects the source from the sink.
- Use Classic max Classic max-flow-min-cut theorem to find max flow
  - 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

Checking Flow Bound

Once a maximum flow has been discovered, the tool computes a cut by

- enumerate the nodes on the source side of the cut by depth-first search
- the cut edges are those that connect nodes reached in the DFS to nodes not reached.

Checking Flow Bound

After getting the cut, we do a checking.

- Taint-based checking: Checking that no secret information reaches the output other than across a given cut.
  - The cut edges correspond to annotations that clear the taint bits on data

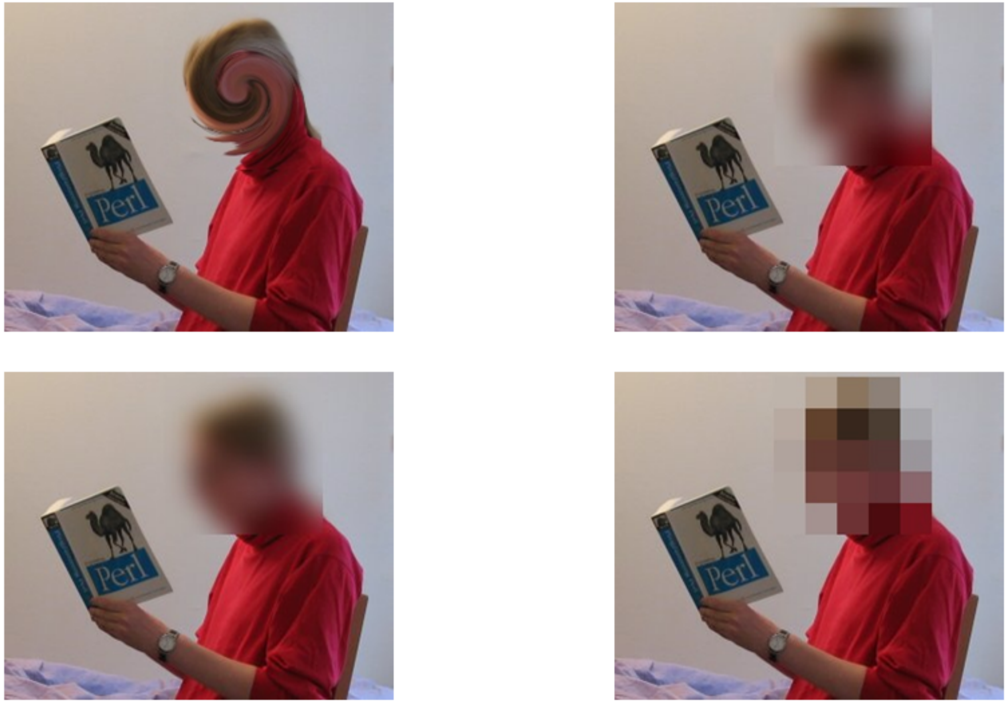
Case Studies

Performed case study on 5 programs.

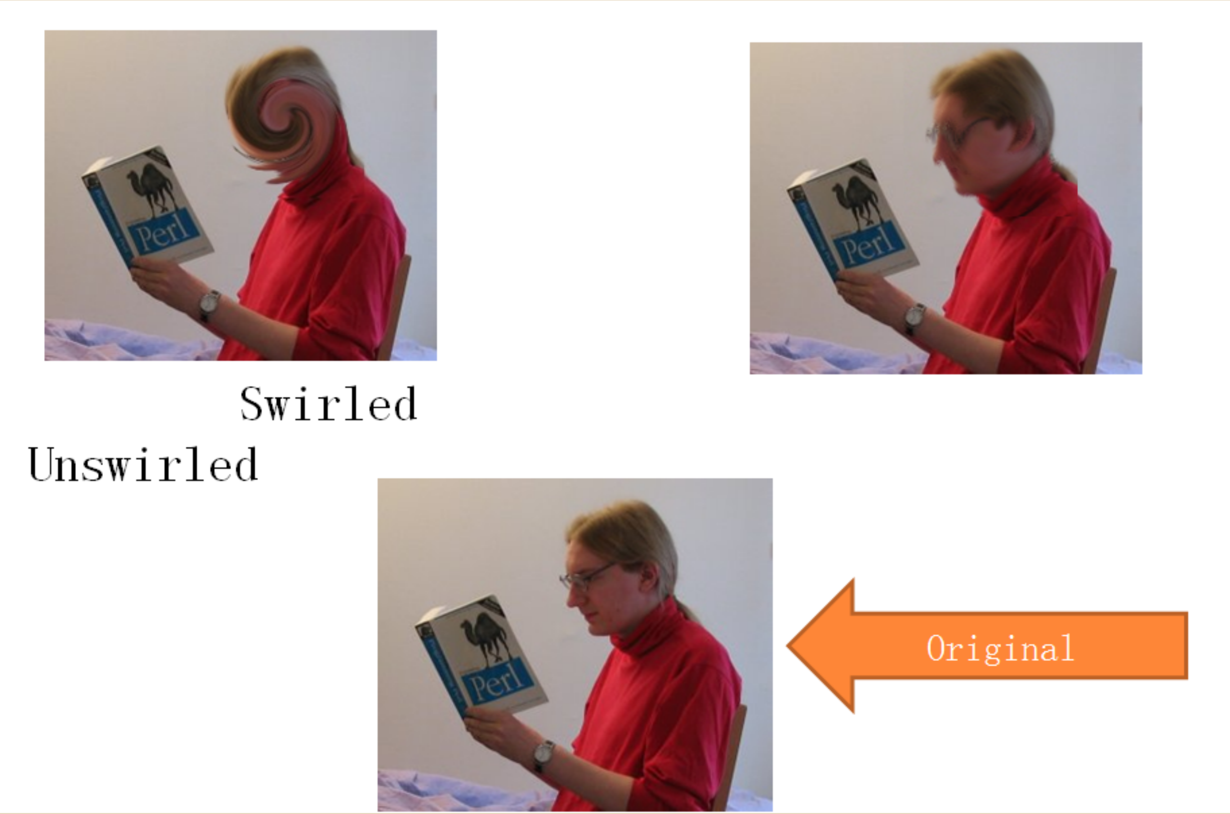
- ImageMagick is a suite of programs for converting and transforming bitmap images.
- Evaluate some of its transformations to assess how much information about the original they preserve.

Case Studies

Which one hides information the best?



Case Studies



That’s it! Thanks!