Short Paper: Bounding Information Leakage Using Implication Graph

Ziyuan Meng
University of Central Missouri
Warrensburg, MO 64093
Missouri, United States
ziyuanmeng001@gmail.com

ABSTRACT

Quantitative information-flow analysis (QIF) is an important approach to assess confidentiality property of software systems. A crucial step towards its practical application is to develop automatic techniques for measuring the information leakage in a system. In this paper, we address this question in the context of deterministic imperative programs and under the min-entropy measure of information leakage. In this context, calculating the maximum leakage of a program reduces to counting the number of possible outputs that it can produce. Our approach is based on an new abstract domain over implication graphs. Unlike numeric abstract domains, implication graphs describe relationships among bits. By executing a program on the domain over implication graphs, we can determine some implication constraints among pairs of bits in the output. By counting the number of solutions to the implication constraints, we can deduce an upper-bound on the leakage. We present the mathematical definition of the abstract domain, and explore its effectiveness in measuring leakage on a few case studies.

Keywords

Privacy; Quantitative Information Flow; Abstract Interpretation

1. INTRODUCTION

Computer systems are prone to leaks of their confidential information. Eliminating such leaks completely is often impractical due to the existence of side channels. *Quantitative information flow* (QIF) addresses this problem by quantifying the amount of confidential information leaked by a system, with the goal of showing that it is "small" enough to be tolerated; this area has seen growing interest over the past decade [6, 7, 13, 23, 1, 10].

A crucial step towards the practical application of QIF is to develop automatic techiniques for measuring the amount of leakage in a system. This is an area that is now seeing a great deal of work [4, 19, 12, 14, 21, 20], utilizing model checking and abstract interpretation techniques.

PLAS'16, October 24 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2993600.2993605

```
1  if (X <= 4) { Y = X; }
2  else { Y = 12; }
```

Figure 1: Illustrative example program that leaks information from X to Y

In the literature, a variety of entropy-like measures have been proposed for quantifying information leakage. But, pleasantly, if we restrict our attention to *deterministic systems* and to their *capacity* (i.e. their maximum leakage over all prior distributions on the secret input), we have the following theorem [23, 5, 2]:

THEOREM 1.1. *The capacity of a deterministic system, whether measured by Shannon entropy or min-entropy, is the logarithm of the number of feasible outputs.*

Thus, calculating the maximum min-entropy leakage of a deterministic program reduces to counting the number of possible outputs that it can produce.

In our previous work [16, 17], we developed an approach to bound this quantity by determining *implication constraints* among the bits of the possible outputs. As an example, consider the C-like program in Figure 1 which takes a secret input `X` and produces an output `Y`. Here, we assume `X` and `Y` are 4-bit unsigned integers. By executing the program, `Y` can have 6 possible outputs: 0, 1, 2, 3, 4, 12, giving min-capacity of $\log 6 \approx 2.58$ bits. By viewing `Y` as a 4-bit vector, we can present the outputs in their binary form:

$$\{\texttt{0000}, \texttt{0001}, \texttt{0010}, \texttt{0011}, \texttt{0100}, \texttt{1100}\}$$

where we index the 4 bit positions from 3 down to 0. By studying the outputs, we notice that there is a relationship between each pair of bits. For instance, if we examine bits 3 and 1, we see that the possible combinations of values that they can take are $\{\texttt{00}, \texttt{01}, \texttt{10}\}$. We express this as $Nand(3,1)$—bits 3 and 1 can not be both at `1`. This can be expressed as *implications*: $3 \to \bar{1}$, $1 \to \bar{3}$. Here $\to$ stands for implication, $\bar{i}$ stand for negated bit $i, 0 \leq i \leq 3$. For bits 3 and 2, the possible combinations of values that they can take are $\{\texttt{00}, \texttt{01}, \texttt{11}\}$. We express this as $Leq(3,2)$—bit 3 is less than or equal to bit 2. This can also be expressed as implications: $3 \to 2, \bar{2} \to \bar{3}$. Among other pairs of bits in `Y`, we can determine the following constraints:

$$Nand(3,0): \quad 3 \to \bar{0}, 0 \to \bar{3}$$
$$Nand(2,1): \quad 2 \to \bar{1}, 1 \to \bar{2}$$

These implication constraints can be represented as a directed *implication graphs* whose nodes represent literals and whose edges represent logical implications, as shown in Figure 2. The implication graph provides an *over-approximation* of the feasible outputs. If we count the number of solutions to all the implications, we get an upper bound on the number of possible values of `Y`. Here there are 7 solutions to the implications, giving a maximum min-entropy leakage of at most $\log 7 \approx 2.80$ bits, which is close to the actual leakage of 2.58 bits. In our previous work [17], we relied on satisfiability modulo theories (SMT) solvers to verify the implication constraints among the bits in the output. In the small case studies, we found that implication graphs usually gave quite accurate bounds on leakage. However, SMT solvers have limited scalability due to state-explosion. When we scale to complex programs, it is very difficult for a SMT solver to verify an implication constraint which depends on the behavior of the entire program. Abstract interpretation has a much greater scalability. The challenge of applying abstract interpretation to QIF is that the side-channel leakage often occur at binary level implementation. In binary code, data is frequently transformed using bitwise operations for which the conventional word-level abstract domains are not well-suited.

In this paper, we present a new abstract interpretation approach based on implication graphs, and apply it to compute upper-bound on the information leakage in imperative deterministic programs. We show that implication graphs constitute an abstract domain. By executing a program using abstract states represented by implication graphs, we can determine *some* implication constraints among the bits in its feasible outputs. From the implication constraints, we can deduce an upper-bound on the number of feasible outputs hence on the leakage. By using implication graphs as an abstract domain, our approach avoids state-explosion problem, and have a greater potential to scale up to complex programs. Compared to word-level abstract domains, implication graphs also provide a better way to model the effects of bitwise operations commonly present in C code and machine code.

The rest of the paper is structured as follows: Section 2 presents formal framework of the abstract domain over implication graphs; Section 3 presents the abstract semantics for an imperative language; Section 4 presents preliminary QIF analysis experiments; Section 5 briefly discusses related work; Section 6 discusses future directions and conclude.

2. OUR FORMAL FRAMEWORK

In this section, we briefly review the mathematical definition of implication graphs. We further present the abstract domain over implication graphs.

2.1 Implication Graphs

Let $I = \{0, 1, 2, .., N-1\}$ be a finite set of *indices* for the bits in a program. We define *literals* $\hat{I} = I \cup \{\bar{i} \mid i \in I\}$. An *implication graph*, furthermore referred to as an I-Graph, over $\hat{I}$ is a directed graph whose nodes represent literals and whose edges represent implications [15, 3]. The I-Graphs have the property of being *skew-symmetric* [9, 11], since there is an edge from i to j iff there is an edge from $\bar{j}$ to $\bar{i}$. An I-Graph can be represented by an adjacency matrix. Figure 3 shows the adjacency matrix of the I-Graph in Figure 2. (To avoid clutter, we omit self-loops in the graphical representation in Figure 2.)

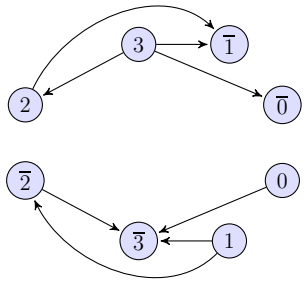

Figure 2: The I-Graph for {0000, 0001, 0010, 0011, 0100, 1100}

	3	2	1	0	$\bar{3}$	$\bar{2}$	$\bar{1}$	$\bar{0}$
3	1	1	0	0	0	0	1	1
2	0	1	0	0	0	0	1	0
1	0	0	1	0	1	1	0	0
0	0	0	0	1	1	0	0	0
$\bar{3}$	0	0	0	0	1	0	0	0
$\bar{2}$	0	0	0	0	1	1	0	0
$\bar{1}$	0	0	0	0	0	0	1	0
$\bar{0}$	0	0	0	0	0	0	0	1

Figure 3: The adjacency matrix for the I-Graph in Figure 2

2.2 Abstraction and Concretization

Given a set of literals $\hat{I}$, a *state* ρ is a mapping:

$$\rho : \hat{I} \to \mathbb{B}$$

where $\mathbb{B} = \{0, 1\}$, $\rho(\bar{i}) = \overline{\rho(i)}$. Since we want to analyze programs using abstract states represented by I-Graphs, we need to see I-Graphs (represented as adjacency matrices) as *abstract domain* for the *concrete domain* of sets of states [8, 18].

We define an *abstraction function* α that maps a set R of states to an I-Graph over $\hat{I}$:

$$\alpha(R)_{ij} \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if for all } \rho \in R,\ \rho(i) \le \rho(j) \\ 0, & \text{otherwise.} \end{cases}$$

Next, we define the *concretization function* γ that maps an I-Graph $\mathbf{m}$ to a set of states:

$$\gamma(\mathbf{m}) \stackrel{\text{def}}{=} \{\rho \mid \text{ for all } i, j, \text{ if } \mathbf{m}_{ij} = 1 \text{ then } \rho(i) \le \rho(j)\}$$

The key correctness property of the I-Graph domain is given by the following theorem, which ensures that when we calculate I-Graph $\mathbf{m} = \alpha(R)$, where R is the set of states, then we know that $\gamma(\mathbf{m})$ is a *superset* of R, implying that we thus *over-approximate* the set of feasible states.

THEOREM 2.1. *Given any set R of states, $R \subseteq \gamma(\alpha(R))$.*

Note that the implication constraints specified in an arbitrary I-Graph $\mathbf{m}$ may be incoherent. For instance, we might have $\mathbf{m}_{ij} = 1$ and $\mathbf{m}_{jk} = 1$, but $\mathbf{m}_{ik} = 0$. In this paper, we

restrict our attention to *coherent* I-Graphs with following definition:

DEFINITION 2.1. *An I-Graph* $\mathbf{m}$ *is coherent if there exists a non-empty set of states* R *such that* $\mathbf{m} = \alpha(R)$.

THEOREM 2.2. *If* $\mathbf{m}$ *is coherent, then* $\mathbf{m}$ *is transitively closed.*

2.3 $\trianglelefteq$ Order

The partial order $\trianglelefteq$ on the set of coherent I-Graphs is defined as follows:

$$\mathbf{m} \trianglelefteq \mathbf{n} \overset{\text{def}}{\Longleftrightarrow} \forall i, j, \mathbf{m}_{ij} \geq \mathbf{n}_{ij}.$$

The idea behind this definition is that for each pair of literals (i, j), $\mathbf{n}_{ij}$ is more relax than $\mathbf{m}_{ij}$. If a state ρ satisfies $\mathbf{m}$, it also satisfies $\mathbf{n}$. In the rest of this section, we will define *Union* and *Intersection* of I-Graphs with respect to $\trianglelefteq$ order. We will then extend $\trianglelefteq$ to form a complete lattice.

2.4 Union and Intersection

We can use boolean operator `and` to define point-wise least upper bound operator $\vee$ on coherent I-Graphs with respect to the $\trianglelefteq$ order:

$$(\mathbf{m} \vee \mathbf{n})_{ij} \overset{\text{def}}{=} \mathbf{m}_{ij} \texttt{ and } \mathbf{n}_{ij};$$

The following theorem proves that $\vee$ preserves coherence, and it over-approximates set union.

THEOREM 2.3.

1. *if* $\mathbf{m}$, $\mathbf{n}$ *are coherent, then* $\mathbf{m} \vee \mathbf{n}$ *is coherent.*
2. $\gamma(\mathbf{m} \vee \mathbf{n}) \supseteq \gamma(\mathbf{m}) \cup \gamma(\mathbf{n})$.

Similarly, we can use boolean operator `or` to define point-wise greatest lower bound operator $\wedge$ on coherent I-Graphs with respect to the $\trianglelefteq$ order:

$$(\mathbf{m} \wedge \mathbf{n})_{ij} \overset{\text{def}}{=} \mathbf{m}_{ij} \texttt{ or } \mathbf{n}_{ij};$$

The following theorem proves that $\wedge$ over-approximates set intersection.

THEOREM 2.4. $\gamma(\mathbf{m} \wedge \mathbf{n}) \supseteq \gamma(\mathbf{m}) \cap \gamma(\mathbf{n})$.

However, $\wedge$ does not preserve coherence. Moreover, the results of $\wedge$ might not be satisfiable. For instance, when $\mathbf{m}_{i\bar{i}} = 1$ and $\mathbf{n}_{\bar{i}i} = 1$, $\mathbf{m} \wedge \mathbf{n}$ contains contradicting implications: $i \to \bar{i}$ and $\bar{i} \to i$. Therefore, we need to first decide whether the result of $\wedge$ is satisfiable. If it is satisfiable, we then need to compute its transitive closure to preserve coherence. Fortunately, determining whether an I-Graph is satisfiable can be reduced to 2-SAT problem which can be solved in polynomial time. A transitive closure can also be computed in polynomial time [22].

To form a lattice on the set of coherent I-Graphs $\mathcal{M}$, we introduce the *least element* $\bot$, and extend $\wedge$ to $\mathcal{M}_\bot = \mathcal{M} \cup \{\bot\}$ as follows:

$$\mathbf{m} \sqcap \mathbf{n} \overset{\text{def}}{=} \begin{cases} \bot, & \text{if } \bot \in \{\mathbf{m}, \mathbf{n}\} \text{ or} \\ & \gamma(\mathbf{m} \wedge \mathbf{n}) = \emptyset \\ (\mathbf{m} \wedge \mathbf{n})^*, & \text{otherwise} \end{cases}$$

where $(\mathbf{m} \wedge \mathbf{n})^*$ is the transitive closure of $\mathbf{m} \wedge \mathbf{n}$. We can also extend $\trianglelefteq$ and $\vee$ to $\mathcal{M}_\bot$in an obvious way to get $\sqsubseteq$ and $\sqcup$. The *greatest element* $\top$ is the I-Graph without any implication between different literals:

$$\top_{ij} \overset{\text{def}}{=} \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

THEOREM 2.5. $(\mathcal{M}_\bot, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ *is a complete lattice.*

```
1  if (v̄3 · (v2 · v̄1 · v̄0 + v̄2)) {
2        v4 ← v0 ; v5 ← v1 ; v6 ← v2 ; v7 ← v3 ;
3  } else {
4        v4 ← 0 ; v5 ← 0 ; v6 ← 1 ; v7 ← 1 ;
5  }
```

Figure 4: binary-level translation of the illustrative example

3. ABSTRACT SEMANTICS

To analyze a C-like imperative program, we first translate it into instructions in a binary-level imperative language with the following syntax:

$$\begin{array}{rcl} e & ::= & v_i \mid 1 \mid 0 \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid \bar{e} \\ C & ::= & v_i \leftarrow e \\ & & C_1; C_2 \\ & & \text{if } (e) \text{ then } \{C_1\} \text{ else } \{C_2\} \\ & & \text{while } (e) \text{ do } \{C\} \end{array}$$

Here v_i is an indexed boolean variable with $i \in I$; operator symbol $+$ stands for boolean operator `or`; operator symbol $\cdot$ stands for boolean operator `and`; e is a boolean expression; $v_i \leftarrow e$ is an assignment statement which assigns the value of e to variable v_i.

Although the binary-level language seems very simple, it is powerful enough to describe the behavior of an array-free, C-like imperative program. For instance, if we view variable `X` in our illustrative example in Figure 1 as a 4-bit vector: (v_3, v_2, v_1, v_0), then `X <= 4` can be expressed as a boolean expression $\bar{v_3} \cdot (v_2 \cdot \bar{v_1} \cdot \bar{v_0} + \bar{v_2})$. By mapping bits in `X` to (v_3, v_2, v_1, v_0), and bits in `Y` to (v_7, v_6, v_5, v_4), we can translate our illustrative example in Figure 1 to a binary-level program shown in Figure 4. So far, we do the translation manually, leaving automation of the process to future work.

In this section, we define abstract operations and abstract transfer functions needed for the analysis of *tests* and *assignments* in the binary-level language. We further present the analysis for an entire program.

3.1 $\mathbf{One}_k$ and $\mathbf{Zero}_k$

Given a bit index $k \in I$, let $\mathbf{One}_k$ denote the I-Graph which represents the set of states in which k has a fixed value `1`. Formally, we define $\mathbf{One}_k$ as follows:

$$\mathbf{One}_k \overset{\text{def}}{=} \alpha(R),\ R = \{\rho \mid \rho(k) = 1, \rho(\bar{k}) = 0\}$$

By the definition of α in Section 2, $\mathbf{One}_k$ has a canonical form: it only contains the edges from all literals to k and the edges from $\bar{k}$ to all literals:

$$(\mathbf{One}_k)_{ij} = \begin{cases} 1, & \text{if } j = k \text{ or } i = \bar{k} \text{ or } i = j \\ 0, & \text{otherwise} \end{cases}$$

We define $\mathbf{Zero}_k$ representing states in which k has a fixed value 0 in a similar way.

3.2 Free Operation

Given an I-Graph $\mathbf{m}$, free operation $[\![v_k \leftarrow ?]\!]^\sharp(\mathbf{m})$ transforms $\mathbf{m}$ into a new I-Graph by eliminating all the implications associated with k or $\bar{k}$ in $\mathbf{m}$. Formally, we define $[\![v_k \leftarrow ?]\!]^\sharp(\mathbf{m})$ as follows:

$$([\![v_k \leftarrow ?]\!]^\sharp(\mathbf{m}))_{ij} \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } i \in \{k, \bar{k}\} \text{ or } j \in \{k, \bar{k}\} \\ \mathbf{m}_{ij}, & \text{otherwise} \end{cases}$$

3.3 Abstract Test

The conditionals in our binary-level language are boolean expressions. Given a boolean expression e, its effect on a set of states R is to select the states which can satisfy e. It is defined by the following concrete semantics:

$$[\![e]\!](R) \stackrel{\text{def}}{=} \{\rho | \rho \in R, [\![e]\!](\rho) = 1\}$$

where $[\![e]\!](\rho) \in \{0, 1\}$ is the evaluation of expression e on state ρ.

Given an I-Graph $\mathbf{m}$, abstract test $[\![e]\!]^\sharp(\mathbf{m})$ transforms $\mathbf{m}$ into another I-Graph which over-approximates the set of states $[\![e]\!](\gamma(\mathbf{m}))$. We first define an abstract atomic test $[\![v_i]\!]^\sharp(\mathbf{m})$, where v_i is a boolean variable, as follows:

$$[\![v_i]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} \mathbf{One}_i \sqcap \mathbf{m}$$

The idea behind this definition is to use $\mathbf{One}_i$ to restrict $\mathbf{m}$ so that i is fixed at 1. Similarly, we define $[\![\bar{v_i}]\!]^\sharp(\mathbf{m})$ as follows:

$$[\![\bar{v_i}]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} \mathbf{Zero}_i \sqcap \mathbf{m}$$

Let $\mathbf{m}_i = [\![v_i]\!]^\sharp(\mathbf{m})$ and $\mathbf{m}_{\bar{i}} = [\![\bar{v_i}]\!]^\sharp(\mathbf{m})$, the following theorem proves that $\mathbf{m}_i$ and $\mathbf{m}_{\bar{i}}$ can split $\mathbf{m}$ into two disjoint I-Graphs:

THEOREM 3.1.

1. $\gamma(\mathbf{m}_i) \cap \gamma(\mathbf{m}_{\bar{i}}) = \emptyset$.
2. $\gamma(\mathbf{m}_i) \cup \gamma(\mathbf{m}_{\bar{i}}) = \gamma(\mathbf{m})$.

The abstract test to a complex boolean expression e is defined by induction:

$$[\![e_1 \cdot e_2]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} [\![e_1]\!]^\sharp(\mathbf{m}) \sqcap [\![e_2]\!]^\sharp(\mathbf{m})$$

$$[\![e_1 + e_2]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} [\![e_1]\!]^\sharp(\mathbf{m}) \sqcup [\![e_2]\!]^\sharp(\mathbf{m})$$

$[\![\bar{e}]\!]^\sharp(\mathbf{m})$ is settled by transformation based on De-Morgan laws:

$$\begin{aligned} \overline{(e_1 \cdot e_2)} &= \bar{e_1} + \bar{e_2} \\ \overline{(e_1 + e_2)} &= \bar{e_1} \cdot \bar{e_2} \end{aligned}$$

The following theorem proves that abstract test is *sound*. Given a boolean expression e and an I-Graph $\mathbf{m}$, $[\![e]\!]^\sharp\mathbf{m}$ always *over-approximates* the set of states which are selected by e from $\gamma(\mathbf{m})$.

THEOREM 3.2. $\gamma([\![e]\!]^\sharp(\mathbf{m})) \supseteq [\![e]\!](\gamma(\mathbf{m}))$

3.4 Abstract Assignment

Given a boolean expression e and a boolean variable v_i, the effect of $v_i \leftarrow e$ on a set of states R is defined as follows:

$$[\![v_i \leftarrow e]\!](R) \stackrel{\text{def}}{=} \{\rho[v_i \mapsto u] | \rho \in R, u = [\![e]\!](\rho)\}$$

where $[\![e]\!](\rho) \in \{0, 1\}$ is the evaluation of expression e on state ρ.

Abstract assignment $[\![v_i \leftarrow e]\!]^\sharp(\mathbf{m})$ transforms $\mathbf{m}$ into another I-Graph which approximates the set of states $[\![v_i \leftarrow e]\!](\gamma(\mathbf{m}))$. We define $[\![v_i \leftarrow e]\!]^\sharp(\mathbf{m})$ in two separate situations: (a) When v_i does not appear in e. (b) When v_i appears in e.

3.4.1 When v_i does not appear in e

The assignment $v_i \leftarrow e$ destroys the previous value in v_i and replaces it with the value of e. Hence, the end effect of a boolean assignment $v_i \leftarrow e$ is to equate v_i with e. We can use *not exclusive or* to model the effect: $\overline{(v_i \oplus e)} = v_i \cdot e + \bar{v_i} \cdot \bar{e}$. The analysis of $v_i \leftarrow e$ thus can be reduced to the analysis of $v_i \cdot e + \bar{v_i} \cdot \bar{e}$. Given a boolean assignment $v_i \leftarrow e$ and an I-Graph $\mathbf{m}$, we first use *Free* operation $[\![v_i \leftarrow ?]\!]^\sharp(\mathbf{m})$ to eliminate all the implications associated with i or $\bar{i}$ in $\mathbf{m}$. We then use *abstract test* $[\![v_i \cdot e + \bar{v_i} \cdot \bar{e}]\!]^\sharp(\mathbf{m})$ to further transform the I-Graph. Thus the following definition:

$$[\![v_i \leftarrow e]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} [\![v_i \cdot e + \bar{v_i} \cdot \bar{e}]\!]^\sharp([\![v_i \leftarrow ?]\!]^\sharp(\mathbf{m}))$$

3.4.2 When v_i appears in e

The analysis on $v_i \leftarrow e$ becomes more complicated when v_i appears in e. Given an I-Graph $\mathbf{m}$, by Theorem 3.1, we can split it into two disjoint I-Graphs: $\mathbf{m}_i = [\![v_i]\!]^\sharp(\mathbf{m})$ and $\mathbf{m}_{\bar{i}} = [\![\bar{v_i}]\!]^\sharp(\mathbf{m})$. The effect of $[\![v_i \leftarrow e]\!](\gamma(\mathbf{m}))$ in the concrete semantics hence can be divided to two parts:

$$[\![v_i \leftarrow e]\!](\gamma(\mathbf{m}_i)) \cup [\![v_i \leftarrow e]\!](\gamma(\mathbf{m}_{\bar{i}}))$$

This can be further expressed as:

$$[\![v_i \leftarrow e[v_i \mapsto 1]]\!](\gamma(\mathbf{m}_i)) \cup [\![v_i \leftarrow e[v_i \mapsto 0]]\!](\gamma(\mathbf{m}_{\bar{i}}))$$

Here $e[v_i \mapsto 1]$ represents the boolean expression obtained by replacing v_i with 1 in e. Similarly, $e[v_i \mapsto 0]$ represents the expression obtained by replacing v_i with 0 in e. Therefore, the analysis of $v_i \leftarrow e$ can be defined as:

$$[\![v_i \leftarrow e[v_i \mapsto 1]]\!]^\sharp(\mathbf{m}_i) \sqcup [\![v_i \leftarrow e[v_i \mapsto 0]]\!]^\sharp(\mathbf{m}_{\bar{i}})$$

The following theorem proves that abstract assignment is *sound*. $[\![v_i \leftarrow e]\!]^\sharp(\mathbf{m})$ always *over-approximates* the set of states $[\![v_i \leftarrow e]\!](\gamma(\mathbf{m}))$.

THEOREM 3.3. $\gamma([\![v_i \leftarrow e]\!]^\sharp(\mathbf{m})) \supseteq [\![v_i \leftarrow e]\!](\gamma(\mathbf{m}))$

3.5 Program Analysis

Using the above abstract transfer functions, we define the analysis of an entire program by induction:

$$[\![C_1; C_2]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} [\![C_2]\!]^\sharp([\![C_1]\!]^\sharp(\mathbf{m}))$$

$$[\![\text{if } (e) \text{ then } C_1 \text{ else } C_2]\!]^\sharp(\mathbf{m}) \stackrel{\text{def}}{=} [\![C_1]\!]^\sharp([\![e]\!]^\sharp(\mathbf{m})) \sqcup [\![C_2]\!]^\sharp([\![\bar{e}]\!]^\sharp(\mathbf{m}))$$

So far, we handle a loop by unrolling it completely, leaving definition of widening operation and analysis of a loop to future work.

4. PRELIMINARY QIF ANALYSIS EXPERIMENTS

Our approach to bounding the min-capacity of a deterministic C-like program can be broken down into four major steps. The first step is to translate the program into instructions in the binary-level imperative language presented in section 3. The second step is to perform abstract interpretation using I-Graphs. The third step is to extract an I-Graph over bits of the program outputs from the result of the second step. The final step is to use a #SAT algorithm to count the number of instances that satisfy all the implication constraints in the I-Graph discovered in the third step;the logarithm of this number is an upper bound on the min-capacity.

For the illustrative example in Figure 1, our approach took 12ms to compute the same I-Graph over bits in `Y` as the one shown in Figure 2. Then, it took a lightweight #SAT solver RelSat [1] less than 1ms to determine that there are 7 solutions to the implications, implying a min-capacity of at most $\log 7 \approx 2.80$ bits.[2]

To see the effectiveness of our approach in 32-bit programs, consider the following "sanity check" program from [19], where `X`, `Y` are 32-bit unsigned integers. In this program, `Y` is influenced by `X` only when `X` is found to be within an acceptable range:

```
if (X < 16)
  Y = base + X;
else
  Y = base;
```

where `base` is a constant. The possible outputs range from `base` to `base`+15, giving a min-capacity of $\log 16 = 4$ bits.

An interesting property of this program is that `Y`'s final value depend on the initial value of `base`. When `base` is `0x00001000`, our analysis (using 40 ms) finds that the highest 28 bits of `Y` are fixed: bits 31 through 13 and bits 11 through 4 are fixed at `0`, and bit 12 is fixed at `1`.[3] (here we determine that bit i must be `0` by observing implication $i \rightarrow \bar{i}$; we determine that bit i must be `1` by observing implication $\bar{i} \rightarrow i$.) Moreover, there is no implication constraint among the lowest four bits of `Y`. The # SAT solver required 1 ms to determine that there are 16 solutions to these constraints, giving an exact min-capacity of 4 bits.

In contrast, when `base` is `0x7ffffffa`, the resulting I-Graph of `Y` is much more complex, since the possible outputs range from `0x7ffffffa` to `0x80000009`. Here, our analysis (using 60 ms) finds interesting implication constraints among the bits in `Y`. Namely, bits 30 through 4 are all equal to one another, forming a strongly connected component in the resulting I-Graph. Between bits 31 and 30, we find $31 \rightarrow \bar{30}$ and $\bar{30} \rightarrow 31$. Moreover, between bits 31 and 3, we find $\bar{31} \rightarrow 3$. (Here we omit other implications deducible by transitivity or skew-symmetry.) The # SAT solver required 1 ms to determine that there are 24 solutions to the implications, implying a min-capacity of at most $\log 24 \approx 4.58$ bits, which is close to the actual capacity of 4 bits.

The following 32-bit program from [19] uses clever bit operations to count the number of bits in `X` that are 1, and leaks this count to `Y`:

```
X = (X & 0x55555555) + ((X>>1) & 0x55555555);
X = (X & 0x33333333) + ((X>>2) & 0x33333333);
X = (X & 0x0f0f0f0f) + ((X>>4) & 0x0f0f0f0f);
X = (X & 0x00ff00ff) + ((X>>8) & 0x00ff00ff);
Y = (X + (X>>16)) & 0xffff;
```

It has 33 possible outputs, so its min-capacity is $\log 33 \approx 5.044$ bits. Here our analysis (using 170 ms) finds that the highest 26 bits of `Y` are fixed at `0`. Among the lowest 6 bits, we find 5 interesting implications: $5 \rightarrow \bar{4}$, $5 \rightarrow \bar{3}$, $5 \rightarrow \bar{2}$, $5 \rightarrow \bar{1}$, $5 \rightarrow \bar{0}$. These implications have exactly 33 instances, so our bound is exact.

5. RELATED WORK

We briefly mention some recent works that focus on the calculation of the capacity of deterministic programs. Köpf, Mauborgne, and Ochoa [14] develop an abstract interpretation for bounding capacity, and use it to show bounds on cache leaks in implementations of the AES cryptosystem. They developed a novel abstract domain which is suitable to describe memory and cache states. Phan, Malacaria, Tkachuk, and Păsăreanu [21] calculate the exact leakage by finding feasible program outputs through a SMT-based technique. This approach is more precise than ours. However, when there is a large number of feasible program outputs, finding all the feasible outputs is very expensive. In their follow-up work [20], they tackle a relatively easier problem: whether the information leakage in a program exceed a prespecified small amount.

6. CONCLUSION

We present an abstract interpretation based on a domain over implication graphs, and apply it to calculate upper-bounds on min-entropy leakage in simple yet intricate deterministic programs. In future work, there are several directions to explore in trying to scale up to the analysis of realistic programs. First, we would like to extend the abstract interpretaton to loop and array operations. Second, the translation of imperative programs into bit-level instructions needs to be automated and generalized. Third, we would like to evaluate our approach in real case studies.

7. REFERENCES

[1] M. Alvim, M. Andrés, and C. Palamidessi. Probabilistic information flow. In *Proc. 25th IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 314–321, 2010.

[2] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In *Proc. 25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 265–279, June 2012.

[3] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[4] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proc. 30th IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

[1] http://code.google.com/p/relsat/

[2] All our experiments have been conducted on a computer with a 2.3 GHz Intel Core i3-2310M and 8 GB DDR3-SDRAM main memory.

[3] We number the bits from 0 to 31, right to left.

[5] C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. In *Proc. 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009)*, volume 249 of *ENTCS*, pages 75–91, 2009.

[6] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 18(2):181–199, 2005.

[7] M. Clarkson, A. Myers, and F. Schneider. Belief in information flow. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW '05)*, pages 31–45, 2005.

[8] P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In *Building the Information Society*, pages 359–366. Kluwer Academic Publishers, 2004.

[9] A. V. Goldberg and A. V. Karzanov. Path problems in skew-symmetric graphs. *Combinatorica*, 16(3):353–âĂŞ382, 1996.

[10] S. Hamadou, V. Sassone, and C. Palamidessi. Reconciling belief and vulnerability in information flow. In *Proc. 31st IEEE Symposium on Security and Privacy*, pages 79–92, 2010.

[11] M. Heule, M. Järvisalo, and A. Biere. Efficient cnf simplification based on binary implication graphs. In *SAT*, pages 201–215, 2011.

[12] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proc. ACSAC '10*, pages 261–269, 2010.

[13] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 286–296, 2007.

[14] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proc. 24th International Conference on Computer-Aided Verification (CAV '12)*, pages 564–580, 2012.

[15] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 13:15–20, 1967.

[16] Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *Proc. Sixth Workshop on Programming Languages and Analysis for Security (PLAS '11)*, pages 1:1–1:12, 2011.

[17] Z. Meng and G. Smith. Faster two-bit pattern analysis of leakage. In *Proc. 2nd International Workshop on Quantitative Aspects in Security Assurance (QASA '13)*, pages 2:1–2:14, 2013.

[18] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, Mar. 2006.

[19] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85, 2009.

[20] Q.-S. Phan and P. Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 283–292, New York, NY, USA, 2014. ACM.

[21] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Software Engineering Notes*, 37(6):1–5, Nov. 2012.

[22] L. Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 404–412, 2003.

[23] G. Smith. On the foundations of quantitative information flow. In L. de Alfaro, editor, *Proc. 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS '09)*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302, 2009.