Precise Quantitative Information Flow Analysis Using Symbolic Model Counting

Vladimir Klebanov*

Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
klebanov@kit.edu

Abstract. Quantitative information flow analyses (QIF) are a class of techniques for measuring the amount of confidential information leaked by a program to its public outputs. QIF analyses can be approximative or precise, offering different trade-offs. In this paper, we lift a particular limitation of precise QIF. We show how symbolic model counting replaces explicit leak enumeration with symbolic computation, thus eliminating the associated bottleneck.

1 Introduction

Recently, there has been a surge in research on quantitative information flow analysis (QIF). The research is motivated by the observation that it is not feasible to completely prevent information leaks (i.e., the flow of confidential information to public ports) in realistic programs. Instead, practical security analysis demands a measure of leaked information in order to decide if a leak is tolerable.

QIF analyses can be precise or approximative, offering different trade-offs. The only precise analysis that we are aware of is [1], where the authors present a two-stage approach. The first stage—program analysis—uses an off-the-shelf model checker to compute a summary of information flow in the program. This summary takes the form of an equivalence relation describing which confidential inputs are indistinguishable by public program outputs in a given attack scenario. The second stage—quantification—uses model generation and counting techniques to transform this relation into a variety of information-theoretical metrics.

So far, precise QIF has been applied to quite small examples only, and even there, a number of severe limitations on program complexity and state space had to be put up with. These limitations have been dictated by two bottlenecks: the first is the limited performance of exhaustive program analyses on programs with a high number of paths and/or states (in stage one). The second is the explicit leak enumeration in the quantification stage.

In this paper, we (only) address bottleneck II (quantification stage). We present a new quantification stage based on existing advanced model counting

* This work was supported by the German National Science Foundation (DFG) under the priority programme 1496 "Reliably Secure Software Systems – RS3."

technology of *extended Barvinok counting*. The technique is completely symbolic, thus eliminating the explicit leak enumeration and allowing precise QIF to scale up to programs with a large number of leaks.

Solving bottleneck I in precise QIF is an independent problem, which we do not address here. We are currently investigating the use of user-supplied and/or synthesized program specifications for this purpose.

We illustrate both bottlenecks using examples from the literature in Section 2. Related work is surveyed in Section 3, and basic QIF notions are introduced in Sections 4–6. Section 7 gives an overview of counting techniques, while Section 8 introduces an approach for precise QIF based on it. Section 9 describes our implementation using the KeY deductive verification system [4] and the barvinok counting tool [14], which we used to carry out the experiments. Section 10 gives a complete description of one such experiment. Section 11 concludes.

2 Examples for Bottlenecks in QIF

In the following, we present examples for the QIF bottlenecks outlined in the Introduction. As representatives of the problematic program classes we use programs given in [1]. Each example illustrates a particular issue and a limitation that had to be imposed until now in order to be able to treat the example with existing precise QIF approaches. We start with bottleneck II, which can be eliminated using the technique that we propose.

Bottleneck II (quantification of large leaks) Some programs have inherently large leaks. The program `l = h1 + h2 + h3;` has no low inputs, yet the number of possible outputs (and thus equivalence classes of the indistinguishability relation) is very high. Generating an explicit representative for each equivalence class and measuring the size of each class is infeasible. To treat the example, [1] limits the input domain and assumes $0 \leqslant \mathtt{h}_i \leqslant 9$. The technique that we propose lifts this limitation.

Large leakage can also be due to a large number of experiments, each leaking only a little. The password checker program

```
if (secret==guess) { access=1; } else { access=0; }
```

with a password `secret` of sufficient length is quite secure under a few guessing attacks, but will unavoidably reveal the complete secret upon an exhaustive brute-force attack.[1] So far, the latter fact can be deduced from the verification conditions generated by the program analysis stage, but a measurement of the individual equivalence class sizes is infeasible due to the enumeration bottleneck. The technique that we propose can symbolically compute equivalence class sizes of arbitrarily fine equivalence relations (the identity relation being just an extreme example).

[1] In practice, such attacks are mitigated by introducing delays or shutting down the system after a few unsuccessful attempts.

Bottleneck I (program analysis of programs with a large number of paths/states) An example is the program

```
lo = 0; while (h >= l) { h = h - l; lo = lo + 1; }        (1)
```

modeling an "electronic purse" that receives a secret input `h` with the balance of a bank account and debits a fixed amount `l` from this account until the balance is insufficient for another debit transaction. Upon termination, the program outputs the number of successful debit transactions stored in the variable `lo`. This number reveals partial information about the initial balance of the account. To analyze the leak (resp. residual uncertainty), [1] bounds the initial balance: $0 \leqslant \mathtt{h} < 20$. When considering a single experiment ($\mathtt{l} = 5$), the running time of the model checker is reported as 24 seconds. This bears no good news for scaling the analysis up to higher values of `h`. Using a deductive verification system in exhaustive exploration mode, we could speed up the computation of $\approx_p^E$ for the same bounds by approximately an order of magnitude (see Section 9), which is still too slow to be practicable. As mentioned in the Introduction, we ignore this bottleneck for now.

An extreme instance of bottleneck I occurs when there is a possibility of non-terminating program runs. Due to the finite execution requirements, QIF analyses based on exhaustive space exploration or sampling are limited to terminating programs only.

3 Related Work

There is a large body of work on demonstrating absence of information leaks in programs, which we cannot survey here.

A remarkable result in the field of QIF and the only precise analysis we are aware of is [1]. We have already discussed it to some extent in the introduction and will continue to do so in the following.

The most relevant approximative QIF techniques are [8] and [10]. The former work is concerned with checking quantitative leakage bounds ("does the program leak more than X?") via bounded model checking. It does not use a dedicated counting tool, but encodes detection of (small) leaks as a model checking problem. The latter work is concerned with addressing the scalability issues of [1]. In order to maintain automation, the approach gives up precise computation of the leak and opts for an approximative characterization, deriving lower and upper leak bounds (when measuring residual min-entropy) via abstract interpretation and concolic testing respectively. Measurement of residual Shannon entropy relies in addition on randomized sampling, and its result is only probably correct, with the user choosing the desired confidence level.

A theoretical account of the hardness of quantifying information flow in programs is given in [16], though this work is only concerned with framing QIF as a program analysis (i.e., without employing counting technology).

A seminal work reasoning about information flow in program logics is [5], showing different approaches to formalize and prove both program security (absence of leaks) and insecurity (presence of leaks) in Dynamic Logic for Java and the KeY prover. The self-composition technique was first presented in a workshop version of [5] and received further theoretical treatment and its name in [2]; it was also studied from the point of view of verification in [12].

4 The Formal Framework

We use an instance of Dynamic Logic [6] to state assertions about programs. This allows a unified and concise formulation, but does not mean that the tools used for reasoning about programs have to be based on this formalism. The results can be easily transferred to systems using Hoare Logic, symbolic model checking (for terminating programs), etc.

We assume a deterministic, imperative programming language with integer variables. For succinctness, we will denote several related program variables or terms as $\bar{\mathrm{v}}$, $\bar{t}$, etc. and assume that all operations happen component-wise.

Assertion syntax Our formal framework extends standard first-order logic with arithmetics with two predicate transformers. For every program p and every formula ϕ, $\langle p \rangle \phi$ ("diamond") and $[p]\phi$ ("box") are formulas. The diamond is a weakest precondition predicate transformer (also known as $wp(p, \phi)$). The box is a weakest liberal precondition predicate transformer (also known as $wlp(p, \phi)$). The formula $\psi \to [p]\phi$ has the same intuitive meaning as the triple $\{\psi\}p\{\phi\}$ in Hoare Logic. Our logic is closed under subformula relation; in particular box and diamond operators can appear nested.

Semantics The semantics of the logic is based on the notion of a (program) state, i.e., a first-order structure assigning (among other things) values to program variables. We presume an appropriate signature and refer to the set of all possible states that are based on it as S.

The transition relation $\rho_p \subseteq S \times S$ gives meaning to a program p as a relation between its initial and final states. The definition of the programming language fixes ρ_p for every syntactically valid program p. In this paper, we only consider deterministic programs, so all relations ρ_p are actually partial functions: for every initial state, there is at most one final state.

Furthermore, for any given state $s \in S$: (1) Terms and formulas without box or diamond operators have the meaning as usual in first-order logic. (2) The diamond formula $\langle p \rangle \phi$ is true in s, if the program p started in s terminates and the formula ϕ is true in the state $\rho_p(s)$ reached upon termination. (3) The meaning of a box formula is the same, but termination is not required: $[p]\phi$ is true in s, if either p does not terminate when started in s, or $\langle p \rangle \phi$ is true in s.

A formula is *logically valid* if it is true in every state.[2]

5 Basics of Information Flow

For reasoning about information flow, we classify parts of the state according to a (simple) security lattice:

[2] Thus, there is implicit universal quantification over program variables.

- The signature marks each program variable either as *high* (confidential) or as *low* (publicly observable/changeable).[3]
- According to the above distinction, we define projection functions $\cdot_{hi}$ and $\cdot_{lo}$. Each state $s \in S$ is a pair of its high component s_{hi} and its low component s_{lo}, and $S = S_{hi} \times S_{lo}$.

The attacker The attacker model is as follows. Assume a run of a program p, with an initial state $s = (s_{hi}, s_{lo})$, and the final state $s' = \rho_p(s) = (s'_{hi}, s'_{lo})$. The attacker knows p, s_{lo}, and s'_{lo} (but not s_{hi}, s'_{hi}, or any intermediate states). The goal of the attacker is to learn something about s_{hi}.

The amount of information leaked by the program (and thus the success of the attacker) depends on the number of program runs that the attacker can study. Each such run is called in terminology of [1] an *experiment*, and it is uniquely characterized by the low component s_{lo} of the initial state. We assume that the attacker can freely choose s_{lo}. The analysis presented below is parameterized by a set of experiments E.

Describing information leaks The canonical way to describe information leakage of a program is by grouping confidential inputs that lead to the same public output in a given attack scenario.

Definition 1 (Indistinguishability relation). *For a given program p and a set of low state components* (experiments) *E, the* indistinguishability relation $\approx^E_p \subseteq S_{hi} \times S_{hi}$ *is*

$$\approx^E_p = \{(s_{1hi}, s_{2hi}) \mid \textit{for all } e \in E : \big(\rho_p((s_{1hi}, e))\big)_{lo} = \big(\rho_p((s_{2hi}, e))\big)_{lo}\} \ . \quad (2)$$

Intuitively, $\approx^E_p$ is an "equivalence relation on the set of possible secret inputs. Two inputs are in the same equivalence class whenever the program produces the same result on both inputs. By observing the output of the program, the attacker can then only deduce the secret input up to its [...] equivalence class." [1]

More precisely, for a given p and E, $\approx^E_p$ is an equivalence relation on high state components. Taking any two states with (1) their high components in the same $\approx^E_p$-equivalence class, and (2) their low components identical and in E as initial states for running p will lead to the same observable (i.e., low) final state. We do not consider the case that programs may not terminate in this paper.

Secure programs have a coarse indistinguishability relation, while insecure a fine one. If the indistinguishability relation is identity (very fine), then all equivalence classes are singleton sets, and each low final state corresponds uniquely to a high initial state: the attacker has perfect knowledge. Conversely, the coarsest indistinguishability relation $\approx^E_p = S_{hi} \times S_{hi}$ with only one equivalence class means that the attacker learns nothing about the high inputs observing the low outputs (a scenario known as "non-interference").

[3] This definition forces us to mark local variables as high, but this restriction has no practical consequence.

6 Security Metrics

Given the number and sizes of equivalence classes of $\approx_p^E$, it is possible to compute a range of security measures summarizing information flow (leakage) in a program. The leaked information is the difference between the attacker's initial uncertainty about the secret inputs and the remaining uncertainty after observing the output of the program [11]: "Leakage = initial uncertainty − remaining uncertainty". In the following, we concentrate on quantifying the remaining uncertainty.

We assume that a set of experiments E is fixed, giving rise to the indistinguishability relation $\approx_p^E$ with n equivalence classes $S_{hi} = C_1 \uplus \ldots \uplus C_n$. We assume that the secret inputs are modeled by a random variable H ranging over S_{hi}, and the public outputs are modeled by a random variable L ranging over S_{lo}. The program p restricts the values of H and L that can occur simultaneously. Furthermore, we assume that H follows a uniform distribution, i.e., that all secret inputs are equally likely.

Under these assumptions, the following measures (among other) can be computed: (1) conditional guessing entropy $G(H|L)$ or the expected number of guesses required to determine H after observing L, (2) conditional minimal guessing entropy $\hat{G}(H|L)$ or the expected number of guesses to determine H after observing a value of L corresponding to the weakest secret (smallest class in $\approx_p^E$), (3) conditional Shannon entropy $H(H|L)$ or the lower bound in bit on the expected message length needed to communicate the remaining secret about H after observing L, (4) conditional min-entropy $H_\infty(H|L)$ or a measure in bit reflecting the probability of correctly determining H in a single guess after observing L. The formulas for computing the measures can be found in [11, 1].

It should be noted that different measures have significantly different properties and are appropriate for different scenarios. It may also be necessary to consider several measures in order to give dependable operational guarantees. We refer to [11] for a survey.

7 Counting for QIF

QIF problems are intimately related to counting. In this section, we describe the extended Barvinok counting technique (developed by Verdoolaege et al.) that we use in this paper, as well as its relations to other counting techniques.

Standard Barvinok counting If an equivalence class of $\approx_p^E$ can be described by a system of linear integer inequalities $A\bar{x} \geqslant \bar{b}$ (i.e., if it is a bounded integer polytope), then Barvinok's algorithm [3] can be used to count the number of integer solutions to this system—read elements of the class.

[1] uses an implementation of the algorithm from the LattE framework to do so for each class in separation. This requires enumerating the classes, which is the bottleneck of the approach.

Extended Barvinok counting We propose to use an extension of Barvinok's algorithm [14] implemented in the barvinok tool [13] to eliminate the counting bottleneck of precise QIF. The advantages of the extended counting algorithm are the symbolic nature of computation as well as the additional operations it offers.

barvinok implements (among others) the following operations: (1) set construction $\{\bar{x} \mid \Phi(\bar{x})\}$ resp. `{[x1,...,xn]:...}`, where Φ is a formula over linear arithmetics, (2) relation construction $\{(\bar{a},\bar{b}) \mid \Phi(\bar{a},\bar{b})\}$ resp. `{[a]->[b]:...}`, (3) inverse relation computation R^{-1}, (4) relation range computation $ran\, R$, (5) set projection $\{\bar{x} \mid \exists\bar{v}.\, \Phi(\bar{x},\bar{v})\}$, (6) lexicographic optimization (*lexmin*) on sets (the smallest element) and relations $lexmin\; R = \big\{(\bar{a},\bar{b}) \in R \mid \bar{b} = lexmin\,\{\bar{x} \mid (\bar{a},\bar{x}) \in R\}\big\}$, and finally (7) cardinality computation on sets $|\cdot|$ resp. `card` and relations $|R| = \lambda p.|S(p)|$ with $S(p) = \{s \mid (p,s) \in R\}$ (number of elements in the image of each domain element). The result of cardinality computations is a (symbolic) quasi-polynomial in parameters of the set constraint.

All operations are carried out exactly and completely symbolically. The runtime performance of extended Barvinok counting is not dependent on the range of individual variables. For all our queries, the barvinok tool produced replies instantaneously. The details of how we use these powerful capabilities will be given in Section 8.

Scope and limitations Barvinok-based counting only works when the input of the program under analysis are integer vectors of fixed length. The program may use complicated data structures (arrays, objects, etc.) as long as they do not cross the boundaries of the program (and the program analysis supports them). Furthermore, only programs can be analyzed that admit a sufficiently precise specification in quantifier-free logic with linear arithmetics. Unfortunately, this property cannot be effectively determined from the program syntax. It is, nonetheless, satisfied by a significant class of programs.

An approach to deal with non-linear $\approx_p^E$ descriptions is proposed in [7]. Due to the finiteness of machine integers, such descriptions can be translated into a (large) purely propositional formula ("bit-blasting"). Then, one of the available #SAT solvers is used to count the number of solutions to the SAT problem given by the formula. As far as we understand, this approach does not scale as well as Barvinok counting.

8 Detecting and Quantifying Information Flow

This section describes the actual approach for precise QIF. It uses the standard technique of self-composition [5, 2, 12] in the program analysis stage (essentially as in [1]). We also briefly describe the quantification stage used in [1] before proposing a replacement.

We assume that a program analysis is available implementing a verification condition generation operator $vc(\Psi)$. For every formula Ψ in the sense of Section 4 containing programs, $vc(\Psi)$ returns an equivalent formula in first-order logic with theories (without programs). [1] uses an iterative procedure implemented on top of the model checker ARMC to compute $vc(\cdot)$ for terminating programs. In our experiments, we have used the deductive verification system KeY for this purpose (cf. Section 9). In general, numerous systems exist that offer such functionality.

To deal with loops, the $vc(\cdot)$ operator is typically parameterized by a loop invariant. In this paper, we assume that all loops are bounded and treated by unwinding instead.

Stage 1: Program analysis In the first stage, the approach uses a program analysis to compute a logical description of $\approx_p^E$. We use self-composition, i.e., employ two copies of the program $p(\bar{\mathtt{h}}, \bar{\mathtt{l}})$ with renamed variables: $p_1 := p(\bar{\mathtt{h}}_1, \bar{\mathtt{l}}_1)$ and $p_2 := p(\bar{\mathtt{h}}_2, \bar{\mathtt{l}}_2)$. The goal is to determine a program-free formula $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ such that

$$s_{1\,hi} \approx_p^E s_{2\,hi} \text{ iff } \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \text{ is true in a state } (s_{1\,hi} \oplus s_{2\,hi}, s_{lo}) \ ,$$

where s_{lo} is some low state component and $s_{1\,hi} \oplus s_{2\,hi}$ is a high state component where the values of $\bar{\mathtt{h}}_1$ are the same as the values of $\bar{\mathtt{h}}$ in $s_{1\,hi}$ and the values of $\bar{\mathtt{h}}_2$ are the same as the values of $\bar{\mathtt{h}}$ in $s_{2\,hi}$.

Transcribing the characterization of indistinguishability (Definition 1) in program logic we obtain:

$$\Psi \equiv E(\bar{\mathtt{l}}_1) \wedge (\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2) \wedge \mathit{Pre} \wedge \langle p_1 \rangle \langle p_2 \rangle (\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2) \ . \tag{3}$$

The predicate $E(x)$ describes the set of experiments E. For modeling reasons, one may sometimes wish to restrict the set of initial states by including *Pre*, which is a (symmetric) precondition over the high vocabulary of p_1 and p_2. In this case, the obtained Φ only describes an equivalence relation on the high state components satisfying *Pre*.

The desired logical description of $\approx_p^E$ is then immediately $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \equiv vc(\Psi)$.

Stage 2: Enumerative model generation and counting (to be replaced)

> **Stage 2a: Model generation** After Stage 1, Stage 2a uses model generation techniques to compute a representative system $\{r_i\}$ for $\approx_p^E$. The representative system is computed by an iterative algorithm, which repeatedly asks a model generator for a representative value (i.e., a satisfying assignment to $\bar{\mathtt{h}}_1$ in $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$) that is not equivalent to any of the previously computed representatives. The technical basis for this computation may be the Omega calculator or an SMT solver.
>
> **Stage 2b: Model counting** This stage uses the standard version of Barvinok's algorithm to determine the size of individual equivalence classes. The algorithm takes a set of formulas $\Phi(\bar{\mathtt{h}}, \bar{r}_i)$ as input and returns the (concrete) number of integer assignments to $\bar{\mathtt{h}}$ satisfying each formula. This is the size of the $\bar{r}_i$-equivalence class. [1] uses the LattE framework as an implementation of the algorithm.

Stage 2: Symbolic model counting (our proposal)

Proposition 1. *Given a symbolic counting procedure implementing the operations of Section 7, the size of the equivalence classes of $\approx_p^E$, can be computed as a closed expression as follows:*

$$\{|C_i|\} = |(lexmin\{(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \mid \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)\})^{-1}| \ . \tag{4}$$

The number of equivalence classes n can be computed as

$$n = |\, ran\ lexmin\{(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \mid \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)\}| \ . \tag{5}$$

Example 1. Assume the following description of $\approx_p^E$:

$$\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \equiv\ \bar{\mathtt{h}}_1 = \bar{\mathtt{h}}_2\ mod\ 3 \wedge 0 \leqslant \bar{\mathtt{h}}_1 \leqslant 4 \wedge 0 \leqslant \bar{\mathtt{h}}_2 \leqslant 4$$

(we are choosing small domains for illustration purposes only). It is easy to see

$$\begin{aligned} \approx_p^E =& \{(4,4), (1,4), (3,3), (0,3), (2,2), (4,1), (1,1), (3,0), (0,0)\} \\ lexmin \approx_p^E =& \qquad\qquad\qquad\qquad\qquad\quad \{(2,2), (4,1), (1,1), (3,0), (0,0)\} \ . \end{aligned}$$

Again, these point enumerations are for illustration purposes only. The counting procedure symbolically computes:

$$\begin{aligned} n = |\, ran\ lexmin \approx_p^E |\ = 3 \qquad\qquad & (\text{“number of points in the image”}) \\ \{|C_i|\} = |(lexmin \approx_p^E)^{-1}| = \{2 - \lfloor (1 + h_2)/3 \rfloor \mid 0 \leqslant h_2 \leqslant 2\} \ . & \end{aligned}$$

8.1 Computing Security Metrics

Finally, the symbolic counting results must be combined with the appropriate formula(s) to compute the various security metrics described in Section 6. This calculation can be carried out symbolically, without reintroducing bottleneck II of explicit enumeration. Even if the obtained quasi-polynomials (symbolic terms) encoding the number and size of the equivalence classes are complex, algebraic simplification laws (replacing repeated addition by multiplication, etc.) allow efficiently exploiting their structure.

In difficult cases, applying a computer algebra system may be necessary to calculate the security metrics. In our experience, Mathematica [15] had no problems doing so even for quite unwieldy terms.

9 Implementation

Our implementation consists of two publicly available tools. For program analysis we used the KeY system v1.6 [4, 9] and for counting barvinok v0.35 [13].

The KeY system is a deductive verification system (i.e., a theorem prover) for Java based on Java Dynamic Logic. It includes the box and diamond operators as part of the input syntax, so the program formulas shown in this paper can be supplied to the system virtually verbatim. The system features an explicit proof object. To compute verification conditions, we provided the initial proof goal Ψ in program logic, and run the automated proof search strategy to exhaustion. The conjunction of the open program-free proof goals constituted $vc(\Psi)$.

We did not implement any glueing code between KeY and barvinok, but since both systems use virtually identical syntax for first-order formulas, it was easy to copy and paste between the two.

We only used the imperative fragment of Java. The verification conditions were generated with machine-faithful arithmetics.

We have also tried to run KeY in exhaustive mode on the "electronic purse" example (1) from [1]. This is the only example with non-negligible program analysis running time, which is reported by [1] at 24 seconds for $\mathtt{h} < 20$. Program analysis with KeY took around 3.5 seconds on a mobile system with a 1.60GHz Intel Core2 Duo CPU.

We conjecture that the theorem prover-based analysis is faster as it can compute the indistinguishability relation completely in one symbolic run. A model checker, in contrast, must explore the program repeatedly as the indistinguishability relation is refined with found leaks.

10 A Complete Example: Sum of Three 32-bit Integers

Theorem prover input The following input encodes (3) in KeY.

```
\programVariables{
  int h1a; int h2a; int h3a; int loa;
  int h1b; int h2b; int h3b; int lob;
}

\problem {
  inInt(h1a) & inInt(h2a) & inInt(h3a) &
  inInt(h1b) & inInt(h2b) & inInt(h3b) &
  \<{ loa=h1a+h2a+h3a; }\>\<{ lob=h1b+h2b+h3b; }\>  loa=lob
}
```

Computed indistinguishability relation The following verification condition is computed automatically for the problem above:

```
h1a >= -2147483648 & h1a <= 2147483647
h2a >= -2147483648 & h2a <= 2147483647 &
h3a >= -2147483648 & h3a <= 2147483647 &
h1b >= -2147483648 & h1b <= 2147483647 &
h2b >= -2147483648 & h2b <= 2147483647 &
h3b >= -2147483648 & h3b <= 2147483647 &
    (2147483648 + h1a + h2a + h3a) % 4294967296
  + (2147483648 + h1b + h2b + h3b) % 4294967296 * -1 = 0
```

This formula describes the indistinguishability relation and is abbreviated as `PHI` in the following.

Determining the number of classes We use the following barvinok query to compute n according to (5):

```
card ran (lexmin PHI);
```

The result is `4294967296`. Intuitively, every integer is realizable as output.

Determining individual class size We use the following barvinok query to compute $\{|C_i|\}$ according to (4):

```
card (lexmin PHI)^-1;
```

The result is initially (in the body of the term, after `->`, `[·]` is the floor function)

```
[h1b, h2b, h3b] ->
((18446744073709551616 +
   (6917529017977405443 - 12884901879/2 * h3b + 3/2 * h3b^2) *
   [(2147483649 + h3b)/4294967296]) +
 (-2305843005992468481 + 4294967293/2 * h3b - 1/2 * h3b^2) *
 [(2147483650 + h3b)/4294967296]
) :
 h1b = -2147483648 & h2b = -2147483648 & -2147483648 <= h3b <= 2147483647
```

but simplifies to `[h1b, h2b, h3b] -> 18446744073709551616`. The simplification is achieved by giving the tool the hint to split the term at the values of `h3b` where the floor terms change value (at $\texttt{2147483646} = \texttt{4294967296} - \texttt{2147483650}$, for instance). All classes have thus the same size. This would have been different with unbounded nonnegative integers.

Security metrics After observing the sum of three secret signed 32-bit integers, the Shannon entropy of the secret diminishes from $3 * 32 = 96$ bit to $H(H|L) = \log_2(18446744073709551616) * 18446744073709551616 * 2^{32}/2^{96} = 64$ bit. To corroborate the intuitive plausibility of this result consider the case of 1-bit integers.

11 Conclusion and Future Work

We have extend the scope of precise QIF by using symbolic model counting. Symbolic counting completely eliminates the bottleneck of leak enumeration and allows efficient treatment of variable domains and leaks of arbitrary size.

We have also demonstrated that a deductive verification system and a symbolic model counting system are a good platform for implementing a precise QIF analysis in a satisfyingly direct fashion. The soundness of the analysis results almost directly from the soundness of the tools. In exhaustive exploration mode, a deductive verification system is significantly faster for QIF than an implementation based on model checking.

So far, we have mainly concentrated on treating the programs given in [1] (while proving stronger properties). In the future, we would like to carry out more experiments with the technique presented. Addressing the bottleneck of exploring unbounded program path and state spaces is another goal of ours.

Acknowledgment The author would like to thank Bernhard Beckert for fruitful discussions.

References

1. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings, 30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 141–153. IEEE Computer Society, 2009.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.
3. A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19:769–779, November 1994.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
5. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings, Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.
6. D. Harel. *First-Order Dynamic Logic*. Springer, 1979.
7. J. Heusser and P. Malacaria. Applied quantitative information flow and statistical databases. In *Proceedings of the 6th international conference on Formal Aspects in Security and Trust*, FAST'09, pages 96–110, Berlin, Heidelberg, 2010. Springer-Verlag.
8. J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269. ACM, 2010.
9. The KeY tool. Website at `www.key-project.org`.
10. B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.
11. G. Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.
12. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Proceedings, Symposium on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
13. S. Verdoolaege. The Barvinok tool. Website at `www.kotnet.org/~skimo/barvinok/`.
14. S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.
15. Wolfram Research, Inc. Mathematica version 8.0 (computer algebra system). `http://www.wolfram.com/mathematica/`, 2010.
16. H. Yasuoka and T. Terauchi. Quantitative information flow – verification hardness and possibilities. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 15–27. IEEE Computer Society, 2010.