Automatically Compute Information Flow Quantity via Probabilistic Semantics

Chunyan Mu and David Clark

King's College London
The Strand, London WC2R 2LS
{chunyan.mu,david.j.clark}@kcl.ac.uk

Abstract. Measuring information flow in software has recently become an active research topic in the security community. Information about confidential inputs may flow to public outputs in batch programs. It would be useful to quantify such flows in the computational world. In this paper, We present an automatic analyser for measuring information flow within software systems. We quantify leakage in terms of information theory and incorporate this computation into probabilistic semantics. Our semantic functions provide information flow measurement for programs given secure inputs under any probability distribution. The major contribution is a automatically quantitative analyser based on the leakage definition for such a language. While-loops are handled by applying *entropy of generalized distributions* and relative properties in order to provide a more precise analysis with observing time.

Key words: Language, Security, Non-interference, Semantics, Information Theory, Flow.

1 Introduction

Quantifying and measuring information flow in software has recently become an active research topic. Access control systems are designed to restrict access to information, but cannot control information propagation once accessed. The goal of information flow security is to ensure that the information propagates throughout the execution environment without security violations such that no secure information is leaked to public outputs. The traditional theory based reasoning and analysis of software systems has largely relied on logics, but they are not concerned with bit leakage, nor with the program execution observers. It would be good to have a quantitative study geared towards the tasks relevant for the computational environment in which we live. The quantitative information flow analysis tool can also be used as part of the testing and auditing process, and such research would be beneficent to the analysis of software applications in security related domains such as the military, banks etc.

Traditionally, the approach of information flow security is based on *interference* [12]. Consider *interference* between program variables: informally, the

capacity of variables to affect the values of other variables. Non-interference, *i.e.* absence of interference, is often used in proving that a system is well behaved, whereas interference can lead to mis-behaviors. However, mis-behaviors in the presence of interference will generally happen only when there is *enough* interference. A concrete example is a software system with *access control*. To enter such a system the user has to pass an identification stage; whether subsequently authorized or failed, some information has been leaked so these systems present interference. Otherwise, if the interference in such systems is *small* enough we can be confident in the security of the system. The security community hence requires determining *how much* information flows from *high* level to *low* level, which is known as *quantitative information flow*. Consider the following examples, which show secure information flow is violated during the execution of the programs:

1. `l:=h;`
2. `if (h==0) then l:=0 else l:=1;`
3. `l:=h; while (l<>0) l:=l*2;`

where l is a low security variable, and h is a high security variable. It is obvious that, the *assignment* command causes the entire information in h to flow to l, the *if statement* allows one bit of information in h to flow to l in the case that h and l are *Boolean*, and l learns some information (whether h equals to zero or not) about h via the termination behaviors of *while loop* command. Note that executing the program reduces the uncertainty about secure information and causes the information leakage. Quantifying and measuring information flow aims to compute how much information is leaked, and to suggest how secure the program is from a quantitative point of view.

Clark et al.'s system for a simple programming language[4] was the most complete static quantitative information flow analysis to our knowledge. The main weakness of this work is that the bounds for loops are over pessimistic. Malacaria [16] gave a more precise quantitative analysis of loop construct using partition property of entropy but its application is hard to automate and there is no formal treatment. All the work to date suffers one of two problems: either it is verified but does not give tight bounds or all the examples are given tight bounds but there is no general verified analysis. The *quality* of the analysis for the measurement of information flow needs to be improved. A system with both automatic and precise analysis is required.

In this paper, we consider the mutual information between a high security variable at the beginning of a batch program and a low one at the end of the program, conditioned on the mutual information between the initial values of *high* and *low*, as the measure of how much of the initial secret information is leaked by executing the program. Malacaria's [16] leakage calculation method for loops provides a way of calculating the exact leakage, given knowledge of whether the loop terminates and the maximum possible number of iterations when it does terminate. This calculation method has not been formalized with respect to semantics to date. Nor does it seem likely that an analysis based on abstraction could calculate exact leakage. We show that Kozen's Scott-domain

version of probabilistic state transformer semantics can be used to overcome some of the drawbacks in Malacaria's method. We devise an algorithm that implements Kozen's semantics. It takes as input a probability distribution on the initial store and calculates a probability distribution on the final store when the program sometimes terminates. In fact it calculates a probability distribution at each program point. These can then be used to calculate leakage. Specifically, while-loops are handled by applying the definition of *entropy of generalized distributions* and relative properties in order to provide a more precise analysis with observing time. We show that this algorithm calculates the same quantity as Malacaria's method, thereby providing correctness for his method relative to Kozen's semantics. Unlike Malacaria's method, there is *no need for any initial (human) analysis* of loop behavior and it is completely *automatic* while applying to general while language programs. The drawbacks of our approach are that it is not, in general, lacking abstraction, time complexity can become large in certain circumstances. The critical component in the time complexity is the conditional mutual information calculation. This is confirmed by tests of the implementation. Using the algorithm to generate plots of loop iterations vs. leakage for example can take hours. We are currently doing an abstract analysis for our approach.

The rest of the paper is organized as follows. In Section 2, we briefly review the relevant mathematical background. Section 3 introduces the syntax and the probabilistic semantics, presents a leakage analyser due to such semantics. An implementation is given in Section 4. Finally, we present related work and draw conclusions in Section 5,6.

2 Mathematical Background

In this section we review some definitions in the relevant mathematical background including information theory, measures, random variables and programs.

2.1 Measures, Random Variables and Programs

There is a clear connection between the notion of probability distribution, information theory, and information leakage in a program. Measures assign *weight* on the domain, and probabilities are a particular case of measures. Hence we could apply it to the semantics to drive the distributions' transform. A measure on a space Ω assigns a "weight" to subsets of the set Ω. A set of useful definitions are reviewed as follows referenced in [26]. A *measure space* is a triple $(\Omega, \mathcal{B}, \mu)$, where Ω is a set, $\mathcal{B}$ is a σ*-algebra* of subsets of Ω, μ is a nonnegative, countable additive set function on $\mathcal{B}$. A σ*-algebra* is a set of subsets of a set M that contains $\emptyset$, and is stable by countable union and complementation. A set M with a σ*-algebra* σ_M defined on it is called a *measurable space* and the elements of the σ*-algebra* are the measurable subsets. If M and N are measurable spaces, $f : M \to N$ is a *measurable function* if for all W measurable in N, $f^{-1}(W)$ is measurable in M. A *positive measure* is a function μ defined on a σ*-algebra* σ_M, which is countable

additive, and has a range in $[0, \infty]$. μ is countably additive, if taking $(E_n)_{n\in N}$ a disjoint collection of elements of σ_M, then $\mu(\bigcup_{n=0}^{\infty} E_n) = \sum_{n=0}^{\infty} \mu(E_n)$. A *probability measure* is a positive measure of total weight 1. A *sub-probability measure* has total weight less than or equal to 1. Note that $\mathcal{P}_{\leq 1}(M)$ is the sub-probability measures on M.

We consider denotational semantics for programs. Assume that the vector of all program variables $\boldsymbol{V}$ range over the same state space Ω. The denotational semantics of a command is a mapping from the set M of possible environments before a command into the set N of possible environments after the command. These spaces updated by semantic transformation functions can be used to calculate leakage at each program point.

2.2 Shannon's Measure of Entropy

In order to measure the information flow, we treat the program as a communication channel. Information theory introduced the definition of *entropy*, $\mathcal{H}$, to measure the average uncertainty in random variables. Shannon's measures were based on a logarithmic measure of the unexpectedness of a probabilistic event (random variable). The unexpectedness of an event which occurred with some non-zero probability p was $\log_2 \frac{1}{p}$. Hence the total information carried by a set of *events* was computed as the weighted sum of their unexpectedness: $\mathcal{H} = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$.

Considering a program as a state transformer, random variable X is a mapping between two states which are equipped with distributions, let $p(x)$ denote the probability that X takes the value x, the *entropy* $\mathcal{H}(X)$ of discrete random variable X was defined as: $\mathcal{H}(X) = \sum_x p(x) \log_2 \frac{1}{p(x)}$. Intuitively, *entropy* is a measure of the uncertainty of a random variable, which can never be negative. Furthermore, given two random variables X and Y, the notion of conditional entropy $\mathcal{H}(X|Y) = \sum_y p(y)\mathcal{H}(X|Y = y)$ suggests possible dependencies between random variables, *i.e.* knowledge of one may change the information associated with another. Let $p(x, y)$ denote the joint distribution of $x \in X$ and $y \in Y$, the notion of mutual information between X and Y, $\mathcal{I}(X;Y)$, is given by: $\mathcal{I}(X;Y) = \sum_x \sum_y p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)}$. Conditional versions of mutual information $\mathcal{I}(X;Y|Z)$ denotes the mutual information between X and Y given the knowledge of Z, and is defined as follows: $\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z)$.

2.3 Entropy of Generalized Probability Distributions

From a measure space transformer point of view, the loop command is going to create a set of sub-probability measures(see Section 3). In order to give a more precise leakage analysis, we need an observation of state at any semantic computation point in an abstract way, e.g, we may consider the attacker can observe the iteration of loops. Loop semantics uses sub-measures for loop approximations, and we need to calculate the entropy of sub-measures. However,

Shannon's entropy definition does not work for the sub-probability measures, therefore we need the more general entropy definition. We now consider some properties and definitions of measures of entropy on the set of *generalized probability distributions* by Renyi referenced in [23]. Let $(\Omega, \mathcal{B}, \mu)$ be a probability space, consider a function X defined for $\omega \in \Omega$ and is measurable with respect to $\mathcal{B}$, where X is called a *generalized random variable*. Furthermore, if $\mu(\Omega) = 1$, X is called a *complete random variable*; if $0 < \mu(\Omega) < 1$, X is called a *incomplete random variable*.

The distribution of a generalized random variable is called a *generalized probability distribution*. Specifically, consider a generalized random variable $X = \{x_1, x_2, \ldots, x_n\}$, with probability $p_k = \mu\{X = x_k\}$ for $k = 1, 2, \ldots, n$, such that the generalized probability distribution is written as $\mathcal{P} = (p_1, p_2, \ldots, p_n)$, the *weight* of the distribution $\mathcal{P}$ is defined by: $W(\mathcal{P}) = \sum_{k=1}^{n} p_k$, *and* $0 < W(\mathcal{P}) \leq 1$. It is easy to see that the weight of a complete distribution is equal to 1, and the weight of an incomplete distribution is less than 1.

Let Δ denote the set of all finite discrete generalized probability distributions, *i.e.* the set of all sequences $\mathcal{P} = (p_1, p_2, \ldots, p_n)$, where $0 < \sum_{k=1}^{n} p_k \leq 1$. $\forall \mathcal{P} \in \Delta$, the entropy of a generalized probability distribution $\widetilde{\mathcal{H}}(\mathcal{P})$ is defined as:

$$\widetilde{\mathcal{H}}(\mathcal{P}) = \frac{\sum_{k=1}^{n} p_k \log_2 \frac{1}{p_k}}{\sum_{k=1}^{n} p_k}$$

Entropy on Partitions The definition of entropy of partitions presented by Rokhlin [25] and the *partition property* of entropy given by Renyi [24] suggests that the entropy of a space with a partition can be computed by summing the entropy of each weighted partition. Formally, given a generalized distribution μ over a set of events $E = \{e_{1,1}, \ldots, e_{n,m}\}$: $\mu(E_i) = \sum_{1 \leq j \leq m} \mu(e_{i,j})$, and a partition of E in sets $(E_i)_{1 \leq i \geq n}$: $E_i = \{e_{i,1}, \ldots, e_{i,m}\}$, the entropy of E can be computed by:

$$\widetilde{\mathcal{H}}(\mu(e_{1,1}), \ldots, \mu(e_{n,m})) = \widetilde{\mathcal{H}}(\mu(E_1), \ldots, \mu(E_n)) + \sum_{i=1}^{n} \mu(E_i) \widetilde{\mathcal{H}}(\frac{\mu(e_{i,1})}{\mu(E_i)}, \ldots, \frac{\mu(e_{i,m})}{\mu(E_i)})$$

where, $\mu(E_i) = \sum_{j=1}^{m} \mu(e_{ij})$ $(i = 1, 2, \ldots, n)$, and by assumption, $\sum_{i=1}^{n} E_i = \sum_{i=1}^{n} \sum_{j=1}^{m} e_{ij} \leq 1$.

This formula can be considered as a theorem about the information associated with a mixture of distributions, see Renyi [24]. Indeed, the entropy of set E is the information associated with the mixture of the subset (a partition of E) distributions $\frac{\mu(e_{i,1})}{\mu(E_i)}, \ldots, \frac{\mu(e_{i,m})}{\mu(E_i)}$ with weights $\mu(E_i)$. The above formula suggests that this information is equal to the sum of the average of the information $\frac{\mu(e_{i,1})}{\mu(E_i)}, \ldots, \frac{\mu(e_{i,m})}{\mu(E_i)}$ with weights $\mu(E_i)$ and the information associated with the mixing distribution $(\mu(E_1), \ldots, \mu(E_n))$. Furthermore, if we denote the set of events in E by ξ, denote a partition of E in sets E_i by η, and denote the elements $e_{i,j}$ of the partition E_i as ζ, then we have: $\widetilde{\mathcal{H}}(\xi) = \widetilde{\mathcal{H}}(\eta) + \widetilde{\mathcal{H}}(\zeta|\eta)$.

3 The Leakage Analyzer

The semantics concerned with probability distribution behavior in program analysis and leakage computation will be presented in this section. Measures express the intuitive idea of a "repartition of weight" on the domain, probabilities are a particular case of measures. Hence we would expect this to apply to the semantics. We apply Kozen's [14] probabilistic semantics as the framework to build the relationship between the probability distribution functions and the variables, expressions, and the commands in a program language to present the probability distribution transformations during the program executions. This provides a basis to develop an automatic analysis of leakage measurement since there is an intimate connection between probability distribution and measurement of information flow.

The property of program executions considered here is the notion of probability distribution on the set of states which induces the computation transformations of probability distributions. The intuition behind a probabilistic transition is that it maps an *input distribution* to an *output distribution* so that we can get a series of computation traces of probability distributions. Next we present a set of measurable functions on the set of traces of execution. The transition probability functions map each state of the first state space to a probability distribution on the second state space. This will help us to explore the property of quantitative information flow on the transformations, in which the sequence reaches in a certain state.

3.1 The Language and its Semantics

The language we considered is standard, presented in Table 1.

$c \in \texttt{Cmd}$ $\quad \texttt{x} \in \texttt{Var}$ $\quad \texttt{e} \in \texttt{Exp}$ $\quad \texttt{b} \in \texttt{BExp}$ $\quad \texttt{n} \in \texttt{Num}$
$c ::= \texttt{skip} \mid \texttt{x} := \texttt{e} \mid \texttt{c}_1; \texttt{c}_2 \mid \texttt{if b then c}_1 \texttt{ else c}_2 \mid \texttt{while b do c}$
$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2$
$b ::= \neg b \mid e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 = e_2$

Table 1. The language

The denotational semantics for measure space transformations are in the following forms:

$$\begin{array}{ll} Val \triangleq \langle \Omega, \mathcal{B}, \mu \rangle & \Sigma \triangleq \boldsymbol{X} \to Val \\ C[\![\cdot]\!] : \texttt{Cmd} \to (\Sigma \to \Sigma) & E[\![\cdot]\!] : \texttt{Exp} \to (\Sigma \to \texttt{Val}) \\ B[\![\cdot]\!] : \texttt{BExp} \to (\Sigma \to \Sigma) & \end{array}$$

Kozen [14] presents two equivalent semantics for probabilistic programs. One interprets programs as partial measurable functions on a measurable space, the

other interprets programs as continuous linear operators on a Banach space of measures. We are more interested in the first one which expresses properties of probabilistic behavior of programs at a more appropriate level of abstraction.

The denotational semantics for measure space transformations are in the following forms:

$$\begin{array}{ll} Val \triangleq \langle \Omega, \mathcal{B}, \mu \rangle & \Sigma \triangleq \boldsymbol{X} \to Val \\ C[\![\cdot]\!] : \mathtt{Cmd} \to (\Sigma \to \Sigma) & E[\![\cdot]\!] : \mathtt{Exp} \to (\Sigma \to \mathtt{Val}) \\ B[\![\cdot]\!] : \mathtt{BExp} \to (\Sigma \to \Sigma) & \end{array}$$

We concentrate on the distributions and present the semantic functions by using Lambda Calculus and the notation of inverse function following [21], see Table 2.

$$\begin{array}{rl} f_{[\![x:=e]\!]}(\mu) & \triangleq \lambda W.\mu(f^{-1}_{[\![x:=e]\!]}(W)) \\ f_{[\![c_1;c_2]\!]}(\mu) & \triangleq f_{[\![c_2]\!]} \circ f_{[\![c_1]\!]}(\mu) \\ f_{[\![\mathtt{if\ b\ then\ c_1\ else\ c_2}]\!]}(\mu) & \triangleq f_{[\![c_1]\!]} \circ f_{[\![b]\!]}(\mu) \ + \ f_{[\![c_2]\!]} \circ f_{[\![\neg b]\!]}(\mu) \\ f_{[\![\mathtt{while\ b\ do\ c}]\!]}(\mu) & \triangleq f_{[\![\neg b]\!]}(\lim_{n\to\infty}(\lambda\mu'.\mu + f_{[\![c]\!]} \circ f_{[\![b]\!]}(\mu'))^n)(\lambda X.\bot)) \\ & \text{where, } f_{[\![B]\!]}(\mu) = \lambda X.\mu(X \cap B) \end{array}$$

Table 2. Probabilistic Denotational Semantics of Programs

This semantics can be considered as a distribution transformer. The vector of the program variables $\boldsymbol{V} = \{x | x \in \boldsymbol{V}\}$ satisfy some *joint distribution* μ on program input, a program $[\![C]\!]$ maps distribution μ over a $\boldsymbol{V}$ to distributions $f_{[\![C]\!]}(\mu)$ over $\boldsymbol{V}$: $f_{[\![C]\!]} : \mu \to \mu$, and $X : \{\mu : \sigma \mapsto [0,1]\}$, where σ denotes the *store*. Due to the measurability of the semantic functions, for all measurable $W \in X'$, $f_{[\![x:=e]\!]}(W)$ is measurable in X. The function $f_{[\![B]\!]}$ for boolean test B defines the set of environments matched by the condition B, which causes the space to split apart. *Conditional statement* is executed on the conditional probability distributions for either the *true* branch or *false* branch: $f_{[\![c_1]\!]} \circ f_{[\![b]\!]}(\mu) \ + \ f_{[\![c_2]\!]} \circ f_{[\![\neg b]\!]}(\mu)$. In the *while loop*, the measure space with distribution μ goes around the loop, and at each iteration, the part that makes test b *false* breaks off and exits the loop, while the rest of the space goes around again. The output distribution $f_{[\![\mathtt{while\ b\ do\ c}]\!]}(\mu)$ is thus the sum of all the partitions that finally find their way out. Note that these partitions are part of the space when the loop paritally terminated, which implies the outputs are partially observable and hence produce an incomplete distributions. For the case that the loop is completely non-terminated, $\bot$ is returned and leakage is 0 when no new paritition is produced but the test is still satisfied. Further details can be found in the following section.

3.2 Automatic Leakage Analysis for Programs

Assume we have two types of input variables: H(confidential) and L(public), and the inputs are equipped with probability distributions, so the inputs can be

viewed as a joint random variable (H, L). From a state transformation point of view, the semantic function of programs maps the state of inputs to the state of outputs. We present the basic leakage definition due to [4] for programs as follows.

Definition 1 (Leakage). *Let H be a random variable in high security inputs, L be one in low security inputs, and let L' be a random variable in the output observation, the secure information flow (or interference) is defined by $\mathcal{I}(L'; H|L)$,* i.e. *the conditional mutual information between the output and the high input given knowledge of the low output. Note that for deterministic programs, we have $\mathcal{I}(L'; H|L) = \mathcal{H}(L'|L)$,* i.e. *interference between the uncertainty in the output given knowledge of the low input.*

Arithmetic Expressions We denote the *probability function* of x as μ_x. Let $\mu_x(v)$ be the probability that the value of x is v, the domain of μ_x will be the set of all possible values that x could be. The computation function $f_e(\mu)$ of an arithmetic expression e defines the measure space that the arithmetic expression can evaluate to in a given set of environments. Specifically, consider the joint probability distribution over the discrete random variable X, Y, by the definition of conditional probability, the distribution function for arithmetic expression $e(x, y)$ is given by:

$$\mu_{\mathtt{e(x,y)}}(z) = \sum_{\mathtt{domain_y}} \mu_{\mathtt{x}}(e^{-1}(z, y))\mu_{\mathtt{y}}(y) = \sum_{\mathtt{domain_x}} \mu_{\mathtt{y}}(e^{-1}(z, x))\mu_{\mathtt{x}}(x)$$

where z stands for a possible value of $e(x, y)$ in the current environment. For instance, the probability density function for *addition* is: $\mu_{x+y}(z) = \sum_{\mathtt{domain_y}} \mu_{\mathtt{x}}(z - y)\mu_{\mathtt{y}}(y)$.

The entropy of expressions is considered as the entropy of its distribution $\mathcal{H}(\mu_e)$, which also can be calculated by using the functional relationship between inputs and outputs [4]: Let $Z = e(X, Y)$, then $\mathcal{H}(Z) = \mathcal{H}(X, Y) - \mathcal{H}(X, Y|Z)$, *i.e.* the entropy of the output space is the entropy of the input space minus the entropy of the input space given knowledge of the output space. Note that, for any constant value $c : \mu_{\mathtt{c}}(c) = 1$, $\mathcal{H}_c = 0$.

Assignment and random input Suppose $[\![c]\!] : X \to Y$, X and Y being metric spaces, W denotes all measurable space in Y is measurable, consider the following linear operator $f_{[\![c]\!]}$ in general, which is presented by inverse image:

$$f_{[\![c]\!]} : \begin{vmatrix} \mathcal{M}_{\leq 1}(X) \to \mathcal{M}_{\leq 1}(Y) \\ \mu \mapsto \lambda W.\mu(f^{-1}_{[\![c]\!]}(W)) \end{vmatrix}$$

Specifically, for the command of assignment, the transformation function for *assignment* updates the state such that the measure space of assigned variable x is mapped to the domain of expression e:

$$f_{x:=e}(\mu) = \lambda X.\mu(f^{-1}_{[\![x:=e]\!]}(X))$$

For example, if there is no low input, variable x is a low security variable with public output, the information leaked to x after command $[\![x := e]\!]$ is given by $\mathcal{L}_{[\![x:=e]\!]} = \mathcal{H}(\mu_e)$.

The probabilistic semantics also gave an interesting input semantics to assignment command such as $x :=$ `input()`. Function `input()` returns a probability distribution. The probabilistic choices behind the random input we consider is *internal*, *i.e.* made neither by the environment nor by the process but according to a given probability distribution. The random input function is therefore transformed to a simple assignment operator: the assigned variable is assigned a given probability distribution.

Sequential composition operator The distribution transformation function for *sequential* is obtained by composition:

$$f_{[\![c_1;c_2]\!]}(\mu) = f_{[\![c_2]\!]} \circ f_{[\![c_1]\!]}(\mu)$$

Let $\mu' = f_{[\![c_1;c_2]\!]}(\mu)$, the low security variable v is public observable, according to the leakage definition, the leakage due to sequential command $[\![c_1; c_2]\!]$ is the entropy $\mathcal{H}(\mu'_v|\mu_v)$, where μ'_v and μ_v are the distribution of variable v after and before the sequential command.

Consider a piece of program `l:=2; l:=h+3; print(l);`, assume h is high security variable, l is low security variable, and initially the variables satisfy with the joint distribution $\mu_{\langle h,l\rangle} = \begin{bmatrix} \langle 0,0\rangle \ w.p.\ 0.3 \\ \langle 1,0\rangle \ w.p.\ 0.5 \\ \langle 2,0\rangle \ w.p.\ 0.2 \end{bmatrix}$, then $f_{[\![l:=2;l:=h+3;]\!]}(\mu_{\langle h,l\rangle}) = \begin{bmatrix} \langle 0,3\rangle \ w.p.\ 0.3 \\ \langle 1,4\rangle \ w.p.\ 0.5 \\ \langle 2,5\rangle \ w.p.\ 0.2 \end{bmatrix}$. The distribution of l is $\mu_l = \begin{bmatrix} 3 \ w.p.\ 0.3 \\ 4 \ w.p.\ 0.5 \\ 5 \ w.p.\ 0.2 \end{bmatrix}$. The leakage due to this piece of program is the conditional entropy $\mathcal{H}(\mu'_l|\mu_l) = 1.485$, which means the information contained in h (uncertainty due to the initial distribution of h) totally flows to output.

Conditional A conditional creates multiple paths conditionalized on some tests, which constructs two branches based on the boolean test, and also makes the partitions a complete distribution. We define that $\mu_b(\texttt{tt})$ is the probability of b to be true, and $\mu_b(\texttt{ff})$ is the probability of b to be false. Let $\mathcal{P}_0 = \{p_0 = \mu_b(\texttt{tt})\}$, $\mathcal{P}_1 = \{p_1 = \mu_b(\texttt{ff})\}$ denote the partition due to the test b, $\mathcal{Q}^l_0 = \{q_{00}, \ldots, q_{0m}\}$, $\mathcal{Q}^l_1 = \{q_{10}, \ldots, q_{1m}\}$ denote the probability distribution of low security variable l in the two branches under the condition that event b is *true/false*,

$$f_{[\![\texttt{if b then c}_1\texttt{ else c}_2]\!]}(\mu) = f_{[\![c_1]\!]} \circ f_{[\![b]\!]}(\mu) + f_{[\![c_2]\!]} \circ f_{[\![\neg b]\!]}(\mu)$$

Assume l is low security variable and output to public, the leakage due to `if` statement is defined as: $\mathcal{L}_{[\![\texttt{if}]\!]} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) + \widetilde{\mathcal{H}}(\mathcal{Q}^l_0 \cup \mathcal{Q}^l_1|\mathcal{P}_0 \cup \mathcal{P}_1)$.

Example 1. Consider the following program. We only consider a k-bit variable with possible values $0, \ldots, s^k - 1$, *i.e.* non-negative numbers for simplification.

```
if (h==0) then l=0 else l=1; print(l);
```

Assume h is a 32-bit *high* security variable with uniform distribution, l is a *low* security variable. It is easy to get, $\mathcal{P}_0 = \{\frac{1}{2^{32}}\}$, $\mathcal{P}_1 = \{1 - \frac{1}{2^{32}}\}$, and $\mathcal{Q}_0^l = \{\mu_l(0)\} = \{\frac{1}{2^{32}}\}$, $\mathcal{Q}_1^l = \{\mu_l(1)\} = \{1 - \frac{1}{2^{32}}\}$. The resulting set of probability distribution transformation is obtained as:

$$f_{[\![\texttt{if}]\!]}\left(\langle h, l\rangle \mapsto \begin{bmatrix} \langle 0, \bot\rangle & w.p.\ 1/2^{32} \\ \ldots & \ldots \\ \langle 2^{32}-1, \bot\rangle & w.p. 1/2^{32} \end{bmatrix}\right) = \left(l \mapsto \begin{bmatrix} 0\ w.p.\ 1/2^{32} \\ 1\ w.p.\ 1 - 1/2^{32} \end{bmatrix}\right)$$

The distribution of low security variable l after the if statement can be described as $l \mapsto \begin{bmatrix} 0\ w.p.\ 1/2^{32} \\ 1\ w.p.\ 1 - 1/2^{32} \end{bmatrix}$, and the information flowed to low component under this example can be computed by:

$$\begin{aligned} \mathcal{H}_{[\![\texttt{if}]\!]} &= \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) + \widetilde{\mathcal{H}}(\mathcal{Q}_0^l \cup \mathcal{Q}_1^l | \mathcal{P}_0 \cup \mathcal{P}_1) \\ &= \widetilde{\mathcal{H}}(\{\frac{1}{2^{32}}\} \cup \{1 - \frac{1}{2^{32}}\}) + 0 = 7.8 \times 10^{-9} \end{aligned}$$

The result implies that this example just releases few information to the public, which agrees with our intuition: the possibility of $h = 0$ is quite low and the uncertainty of h under condition $h \neq 0$ is still big, *i.e.* just few information released.

While Loop The semantic function for loop is given by:

$$f_{[\![\texttt{while b do c}]\!]}(\mu) = f_{[\![\neg b]\!]}(\lim_{n\to\infty}(\lambda\mu'.\mu + f_{[\![c]\!]} \circ f_{[\![b]\!]}(\mu'))^n(\lambda X.\bot))$$

Such function produces the union of all the *sub-measures* that have already quit the loop so far. Intuitively, the initial measure space goes around the loop. At each iteration, the part of the space which leads the boolean test to be *false* exits the loop, while the rest of the space goes around again. This accumulates a set of partitions that have come to occupy the same parts of the program through different paths. At some certain points, e.g. the k^{th} iteration, the output distribution $\mathcal{P}_k$ is the sum of all the pieces that eventually find their way out, *i.e.* the union of sub-measures which have left the loop finally at that point. Consider a terminating loop `while b C` as a sub-measure transformer which builds a set of accumulated incomplete probability distributions, *i.e.* due to the k^{th} iteration,

$$\mathcal{P}([\![\texttt{while b C}]\!]) = \bigcup_{0 \leq i \leq k} \mathcal{P}_i([\![\texttt{while b C}]\!])$$

where $k \leq n$, and n is the maximum number of iteration of the loop. Let,

$$\mathcal{P}_i = \{p_i\} = \{\mu(e^i)\},\ \textit{where}\ e^i = \begin{cases} b^0 = \texttt{ff}, & i = 0 \\ b^0 = \texttt{tt} \wedge b^1 = \texttt{ff}, & i = 1 \\ b^0 = \texttt{tt}, \ldots, b^{i-1} = \texttt{tt} \wedge b^i = \texttt{ff}, & i > 1 \end{cases}$$

where e^i is the event that the loop test b is true until the i^{th} iteration, b^i denotes the value of the boolean test b at the i^{th} iteration. Consider the union of the decompositions

$$\mathcal{P} = (\mathcal{P}_0 \cup \mathcal{P}_1 \cup \ldots \cup \mathcal{P}_k)_{0 \leq k \leq n} = (\{p_0\} \cup \{p_1\} \cup \ldots \cup \{p_k\})_{0 \leq k \leq n}$$

the events $\mathcal{P}_0, \ldots \mathcal{P}_k$ build the partition of the states for a `while` loop. The sum of the probability of $\mathcal{P}_i$ is *less than or equal* to 1, and *equal to* 1 only if $k = n$, *i.e.* the loop terminates at the n^{th} iteration, every part of the space has found its way out or we get the fixed point (no new partition finds way out of the loops but the boolean test is still satisfied) which means the loop is non-terminate with regard to this part of the space.

Next we propose to produce the leakage definition for loop command due to the semantic function. First consider the entropy of the union of the decompositions $\bigcup_{0 \leq i \leq k} \mathcal{P}_i$. According to the mean-value property of entropy [23], let Δ denote the set of all finite discrete generalized probability distributions, if $\mathcal{P}_1 \in \Delta, \mathcal{P}_2 \in \Delta, \ldots, \mathcal{P}_n \in \Delta$, such that $\sum_{k=1}^{n} W(\mathcal{P}_k) \leq 1$, we have:

$$\widetilde{\mathcal{H}}(\mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots \cup \mathcal{P}_n) = \frac{W(\mathcal{P}_1)\widetilde{\mathcal{H}}(\mathcal{P}_1) + \ldots + W(\mathcal{P}_n)\widetilde{\mathcal{H}}(\mathcal{P}_n)}{W(\mathcal{P}_1) + \ldots + W(\mathcal{P}_n)}$$

The entropy of the union of set of incomplete distributions is the weighted mean value of the entropies of the set of distributions, where the entropy of each component is weighted with its own weight $(W(\mathcal{P}_i)_{0 \leq i \leq n})$.

We next consider the amount of information contained in the loop body under the observation of the set of events $E = \{e^i\}_{0 \leq i \leq n}$. We denote the set of events in E by ξ, let $\mathcal{P}$ denote the original distribution of the random variable ξ and $\mathcal{Q}$ denote the conditional distribution of random variable ξ under the condition that event E has taken place. We shall denote a measure of the amount of information concerning the random variable ξ contained in the observation of the event E by $\widetilde{\mathcal{H}}(\mathcal{Q}|\mathcal{P})$. Let $\mathcal{Q}_i^L$ denote the distribution of the low component at the end of the execution of the loop body due to the i^{th} iteration under the condition that event $e^i_{0 \leq i \leq k}$ has taken place, we have:

$$\mathcal{Q}^L = \left(\mathcal{Q}_0^L \cup \mathcal{Q}_1^L \cup \ldots \cup \mathcal{Q}_k^L\right)_{0 \leq k \leq n} = (\{q_{00}, \ldots, q_{0j}\} \cup \ldots \cup \{q_{k0}, \ldots, q_{kj}\})_{0 \leq k \leq n}$$

where k implies up to the k^{th} iteration. By applying the formula of conditional entropy for *generalized* probability distributions [23], and let W denotes *weight*, we have:

$$\widetilde{\mathcal{H}}(\mathcal{Q}^L|\mathcal{P}) = \frac{W(\mathcal{Q}_0^L)\widetilde{\mathcal{H}}(\mathcal{Q}_0^L|\mathcal{P}_0) + \ldots + W(\mathcal{Q}_k^L)\widetilde{\mathcal{H}}(\mathcal{Q}_k^L|\mathcal{P}_k)}{W(\mathcal{Q}_0^L) + \ldots + W(\mathcal{Q}_k^L)}$$

As we discussed above, the loop command creates a set of sub-measures (incomplete distribution). To give a more precise leakage analysis, we need an observation of state at any semantic computation point in an abstract way, e.g, we may consider the attacker can observe the iteration of loops. Loop semantics

uses sub-measures for loop approximations, and we need to calculate the entropy of sub-measures by applying the more general Renyi's *entropy of generalised distributions*. Definition 2 is therefore defined for computing leakage with observing time (iteration) for loops incorporated with our semantic function.

Definition 2. *We define the leakage into low component with regard to the while loop up to the k^{th} iteration by addition of the entropy of the union of the boolean test for each iteration and the sum of the entropy of the loop body for each weighted sub-probability measures:*

$$\begin{aligned}\mathcal{L}_{\texttt{while}}(k) &= \widetilde{\mathcal{H}}(\mathcal{P}) + \widetilde{\mathcal{H}}(\mathcal{Q}^L|\mathcal{P}) \\ &= \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_k) + \widetilde{\mathcal{H}}(\mathcal{Q}_0^L \cup \ldots \cup \mathcal{Q}_k^L|\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_k) \\ &= \frac{\sum_{i=0}^{n}(p_i \log_2 \frac{1}{p_i})}{\sum_{i=0}^{n} p_i} + \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} q_{ij} \log_2 \frac{q_{ij}}{p_i}}{\sum_{i=0}^{n}\sum_{j=0}^{m} q_{ij}} \\ &= \widetilde{\mathcal{H}}(p_0, \ldots, p_n) + \sum_{i=0}^{n} p_i \widetilde{\mathcal{H}}\left(\frac{q_{i0}}{p_i}, \frac{q_{i1}}{pi}, \ldots, \frac{q_{im}}{p_i}\right)\end{aligned}$$

where $\mathcal{P}_i = \{p_i\}$, thus $\widetilde{\mathcal{H}}(\mathcal{P}_i) = \log_2 \frac{1}{p_i}$.

- case $0 \leq k < n$: $\mathcal{P} = \mathcal{P}_0 \cup \ldots \cup \mathcal{P}_k$ and $\mathcal{Q}^L = \mathcal{Q}_0^L \cup \ldots \cup \mathcal{Q}_k^L$ are two *incomplete* distributions, such that, $W(\mathcal{P}_0) + \ldots + W(\mathcal{P}_k) < 1$, and $W(\mathcal{Q}_0^L) + \ldots + W(\mathcal{Q}_k^L) < 1$. With regard to this case, part of the space has exit the loop but not all, we therefore can compute the leakage by observing each iteration before the loop terminates. An intuition behind of the introduction of the notion of entropy of incompleted loops is that the term $\log_2(1/p_k)$ in Shannon's formular is interpreted as the entropy of the generalized distribution consisting of the single probability p_k and thus it implies that Shannon's entropy definition $\mathcal{H}(p_1, \ldots, p_n) = \sum_{k=1}^{n} p_k \log_2(\frac{1}{p_k})$ is actually a mean value.
- case $k = n$: $\mathcal{P} = \mathcal{P}_0 \cup \ldots \cup \mathcal{P}_n$ and $\mathcal{Q}^L = \mathcal{Q}_0^L \cup \ldots \cup \mathcal{Q}_n^L$ are two *complete* distributions, such that, $W(\mathcal{P}_0) + \ldots + W(\mathcal{P}_k) = 1$, and $W(\mathcal{Q}_0^L) + \ldots + W(\mathcal{Q}_k^L) = 1$, *i.e.* $\sum_{i=0}^{n} p_i = \sum_{i=0}^{n}\sum_{j=0}^{m} q_{ij} = 1$, all part of the space exit the loop.
- case $k = \infty$: this is the case of nonterminating (partitially or totally) loops, our tool returns the partitions makes the loop terminated (case of partially termination) and also the left partition which cannot produce new partitions but still satisfy test b.

Proposition 1 presents a relationship between our leakage definition for loops with Malacaria's [16] definition. We show that our algorithm calculates the same *final* quantity (when the loop terminates) as Malacaria's method, hence providing correctness for his method relative to our semantics. Unlike Malacaria's method there is no need for any initial human analysis of loop behavior and it is completely automatic while applying to general while language programs. Furthermore, our leakage rate is *variant* with observing time which is more

precise than Malacaria's *average* rate. Such analysis can provide a leakage profile of programs to show the behavior of information flow with observing time, *i.e.* how much information is leaked till any iteration of interest (depends on the observers). Importantly, our analysis can provide an *automatic* analysis for quantitative information flow and make the leakage computation due to each iteration.

Proposition 1. *Assume the loop is going to terminate at the n^{th} iteration, at the point of termination, the leakage computation for while loop due to our definition is equivalent to Malacaria's definition of leakage for loops [16].*

Proof. $\mathcal{L}_{\texttt{while}}(n)$ is the leakage into the low component with regard to the while loop. The pieces of the space partitioned by the semantic function of while loop are disjoint, hence there is no collisions which happened in Malacaria's definition. According to Definition 2, we have:

$$\begin{aligned}
\mathcal{L}_{\texttt{while}}(n) &= \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_n) + \widetilde{\mathcal{H}}(\mathcal{Q}_0^L \cup \ldots \cup \mathcal{Q}_n^L | \mathcal{P}_0 \cup \ldots \cup \mathcal{P}_n) \\
&= \frac{W(\mathcal{P}_0)\widetilde{\mathcal{H}}(\mathcal{P}_0) + \ldots + W(\mathcal{P}_n)\widetilde{\mathcal{H}}(\mathcal{P}_n)}{W(\mathcal{P}_0) + \ldots + W(\mathcal{P}_n)} + \\
&\quad \frac{W(\mathcal{Q}_0^L)\widetilde{\mathcal{H}}(\mathcal{Q}_0^L|\mathcal{P}_0) + \ldots + W(\mathcal{Q}_k^L)\widetilde{\mathcal{H}}(\mathcal{Q}_k^L|\mathcal{P}_k)}{W(\mathcal{Q}_0^L) + \ldots + W(\mathcal{Q}_k^L)} \\
&= \frac{p_0 \frac{p_0 \log_2 \frac{1}{p_0}}{p_0} + \ldots + p_n \frac{p_n \log_2 \frac{1}{p_n}}{p_n}}{p_0 + p_1 + \ldots + p_n} + \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} q_{ij} \log_2 \frac{q_{ij}}{p_i}}{\sum_{i=0}^{n}\sum_{j=0}^{m} q_{ij}} \\
&= \sum_{i=0}^{n}(p_i \log_2 \frac{1}{p_i}) + \sum_{i=0}^{n} p_i \sum_{j=0}^{m} \frac{q_{ij}}{p_i} \log_2 \frac{q_{ij}}{p_i} \\
&= \mathcal{H}(p_0, \ldots, p_n) + \sum_{i=0}^{n} p_i \mathcal{H}(\frac{q_{i0}}{p_i}, \frac{q_{i1}}{p_i}, \ldots, \frac{q_{im}}{p_i})
\end{aligned}$$

We have proved that, when $k = n$, our leakage definition is equivalent to Malacaria's leakage definition of collision free loops [16]. For the case of semantic function $f : X \to Y$ with collisions in Malacaria's definition, we consider to make an adjusting to Y to eliminate the collisions, drive the new partitions E_i of disjoint measurable subsets of X, and let $\mathcal{P}_i = \mu(E_i)$, where $0 \leq i \leq n$. The rest of the proof is then similar to previous arguments.

Example 2 (A terminating example).

```
l:=0; while(l<h) l:=l+1; print(l);
```

Assume h is a 3-bit *high* security variable with distribution:

$$\left[0 \text{ w.p. } \tfrac{7}{8} \quad 1 \text{ w.p. } \tfrac{1}{56} \quad \ldots \quad 7 \text{ w.p. } \tfrac{1}{56}\right]$$

l is a *low* security variable. The semantic function of while-loop presents the decompositions $\mathcal{P}_i$ with regard to the partitions due to the boolean test, and

the resulting set of probability distribution $\mathcal{Q}_i^L$ of low level variable l due to the loop body under the condition that event e^i has taken place are:

$$\begin{array}{ll} \mathcal{P}_0 = \{\mu(e^0)\} = \{\frac{7}{8}\} & \mathcal{Q}_0^L = \{\mu_l(0)\} = \{\frac{7}{8}\} \\ \mathcal{P}_1 = \{\mu(e^1)\} = \{\frac{1}{56}\} & \mathcal{Q}_1^L = \{\mu_l(1)\} = \{\frac{1}{56}\} \\ \dots & \dots \\ \mathcal{P}_7 = \{\mu(e^7)\} = \{\frac{1}{56}\} & \mathcal{Q}_7^L = \{\mu_l(7)\} = \{\frac{1}{56}\} \end{array}$$

i.e.

$$f_{[\![while]\!]}\left(\langle h,l\rangle \mapsto \begin{bmatrix} \langle 0,0\rangle \ w.p.\ 7/8 \\ \langle 1,0\rangle \ w.p.\ 1/56 \\ \dots \quad \dots \\ \langle 7,0\rangle \ w.p.\ 1/56 \end{bmatrix}\right) = \langle h,l\rangle \mapsto \begin{bmatrix} \langle 0,0\rangle \ w.p.\ 7/8 \\ \langle 1,1\rangle \ w.p.\ 1/56 \\ \dots \quad \dots \\ \langle 7,7\rangle \ w.p.\ 1/56 \end{bmatrix}$$

Note that $q_i = p_i$, hence $\widetilde{\mathcal{H}}(\mathcal{Q}^L|\mathcal{P}) = 0$, *i.e.* the leakage of the body is 0, we calculate the leakage due to each iteration $i_{0\leq i\leq 7}$ by:

$$\begin{array}{ll} \mathcal{L}_{\texttt{while}-0} = \widetilde{\mathcal{H}}(\mathcal{P}_0) = 0.19 & \mathcal{L}_{\texttt{while}-1} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) = 0.30 \\ \mathcal{L}_{\texttt{while}-2} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2) = 0.41 & \mathcal{L}_{\texttt{while}-3} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_3) = 0.52 \\ \mathcal{L}_{\texttt{while}-4} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_4) = 0.62 & \mathcal{L}_{\texttt{while}-5} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_5) = 0.71 \\ \mathcal{L}_{\texttt{while}-6} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_6) = 0.81 & \mathcal{L}_{\texttt{while}-7} = \widetilde{\mathcal{H}}(\mathcal{P}_0 \cup \ldots \cup \mathcal{P}_7) = 0.89 \end{array}$$

The result of this example also matches our expectation: the initial information contained in h is totally revealed to the low security output by the end of the program. Our analysis therefore provides the leakage profile of the loop showing the behavior of information flow by observing the output at each observable time point (in this case at the terminating point of the program).

Example 3 (A partially terminating example).

```
while(l>h) l:=l+1; print(l);
```

Assume h is a *high* security variable and l is a *low* security variable with joint distribution: $\mu_{\langle h,l\rangle} \mapsto \begin{bmatrix} \langle 0,2\rangle \ w.p.\ \frac{7}{8} & \\ \langle 1,2\rangle \ w.p.\ \frac{1}{56} & A \\ - - - - - - - - - - \\ \langle 2,2\rangle \ w.p.\ \frac{1}{56} & \\ \dots \ \dots & \\ \langle 7,2\rangle \ w.p.\ \frac{1}{56} & B \end{bmatrix}$

Note that this example is partially terminating: non-terminating on the top part (A) of the space while terminating on the bottom part (B) of the space. If the observation point is after iteration-1 the leakage is computed by: $\mathcal{L} = \widetilde{\mathcal{H}}(6/56) = 3.22$ otherwise the leakage is computed by: $\mathcal{L} = \widetilde{\mathcal{H}}(6/56, 50/56) = 0.491$. The first partition is part B of the space, which quits the loop immediately without entering the loop, and the second partition is part A which never finds its way out.

Example 4 (A non-terminating example).

```
while(l>h) {l:=l+1;} print(l);
```

Assume h is *high* security variable and l is *low* security variable with joint distribution:

$$\mu_{\langle h,l\rangle} \mapsto \left[\, \langle 0,2\rangle \ w.p.\ \tfrac{1}{3} \quad \langle 1,2\rangle \ w.p.\ \tfrac{2}{3} \quad \right]$$

The whole space will never quit the loop in this example, our tool outputs the result since no going out partition is produced but the boolean test is still satisfied. The leakage computation is given by: $\mathcal{L} = \widetilde{\mathcal{H}}(\bot) = 0$, this also meets our intuition: no information is leaked by a non-terminated loop (assuming non-termination is not observable).

4 An Implementation

For a feasibility study, we have implemented this method which can be used to calculate mutual information between (sets of) variables at any program point automatically. Fig. 1 describes the basic structure of the system. The initial configuration (joint distribution for variables $\boldsymbol{V}$) and the example program are fed into the analyser. The analyser will present the distribution transformation and the measurement of the secret information leaked to low component at each program point by executing the program.

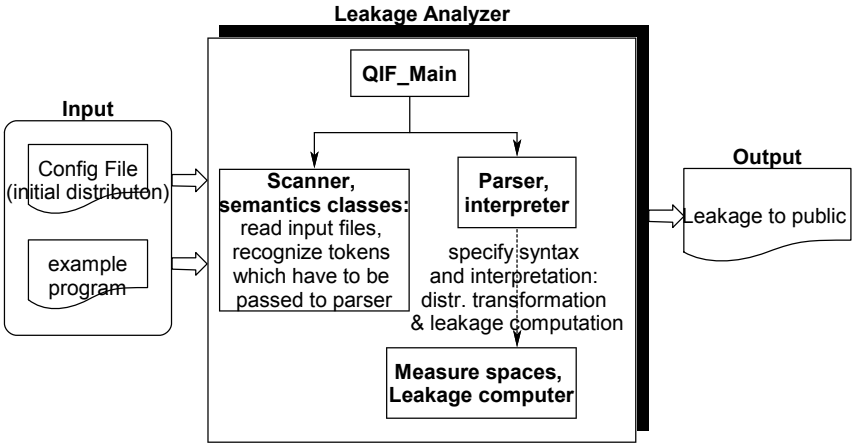

Fig. 1. Implementation: structure

Example 5. The following program computes the following sequence: if h is even then halve it else multiply by 3 and add 1, repeat this process until h is 1, then output l, *i.e.* output how many of this operations performed. Intuitively, this program just publishes some information of h to l, but not too much.

```
int l := 0;
while (h>1) {
  if (h%2 == 0) h := h/2
  else h := h*3+1;
  l++;
  print(l);
}
print(l);
```

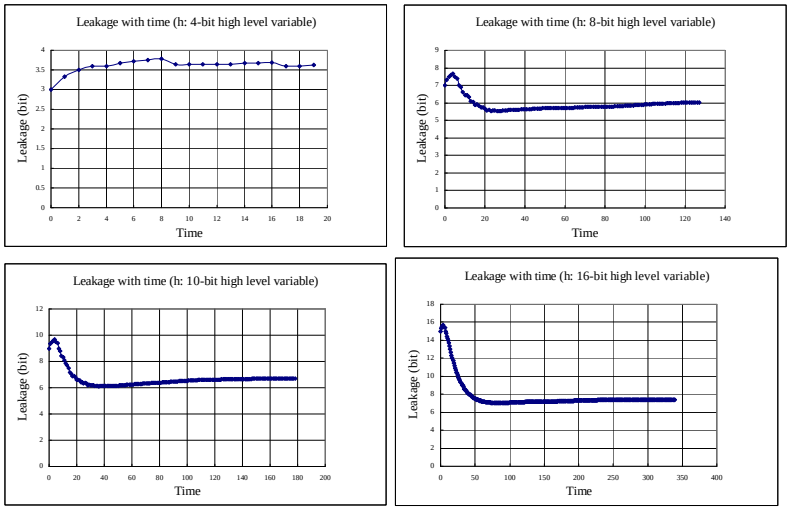

Fig. 2. Results of the experiment

By applying the analysis rules given in Clark et alia [4], the upper bound on leakage will be analyzed as the all the information in h. Malacaria's method can give a precise result but it cannot do it automatically, and the number of iterations is required in advance. Our approach however presents an analysis which is both precise and automatic, also shows the information flow behavior for the observed time (in iterations) elapsed. In order to check the time complexity and feasibility of the analysis, we performed experiments using different sizes of variables. Figure 5 shows the leakage analysis results in the cases of high input h being 4-bit, 8-bit, 10-bit, 16-bit variables under uniform distribution, as well as the time consumed for the final leakage computation (omitting the computations on each iteration) as 30, 82, 103, and 187 seconds respectively. The results for this example show how much secure information flowed from h to l as observed time (each iteration) elapsed.

Let us take h as a 16-bit variable as an example, the initial joint distribution for $\langle h, l\rangle$ is $\begin{bmatrix} \langle 0,0\rangle & w.p.\ \frac{1}{2^{16}} \\ \langle 1,0\rangle & w.p.\ \frac{1}{2^{16}} \\ \dots & \dots \\ \langle 2^{16}-1,0\rangle & w.p.\ \frac{1}{2^{16}} \end{bmatrix}$. Input such initial configuration and the example program into our analyser. The analyser transforms the joint distribution by analyzing the program. Note that at the end of each iteration, l is output to public. The analyser thus computes the leakage at each time(iteration) point(t) $\mathcal{L}_t$ by applying the definition of leakage presented in Section 3, and there are 339 iterations in total. Furthermore, as a test of tractability, using our leakage analysis system to compute the final result (without internal leakage computation) is a matter of minutes, and to generate plots of loop iterations and information flow with observing time takes hours. The critical component in time complexity lies in the conditional mutual information calculation according to the result of the experiments.

5 Related Work

There has been significant activities around the question of how to measure the amount of secure information flow caused by interference between variables by using information theoretic quantities. The precursor for this work was that of Denning in the early 1980's. Denning [8] presented that the data manipulated by a program can be typed with security levels, which naturally assume the structure of a partially ordered set. Moreover, this partially ordered set is a lattice under certain conditions [7]. Denning first explored the use of information theory as the basis for a quantitative analysis of information flow in programs. However, he did not suggest how to automate the analysis. In 1987, Millen [20] first built a formal correspondence between non-interference and mutual information, and established a connection between Shannon's information theory and state-machine models of information flow in computer systems. Later related work is that of McLean and Gray and McIver and Morgan in 1990's. McLean presented a very general Flow Model in [19], and also gave a probabilistic definition of security with respect to flows of a system based on this flow model. The main weakness of this model is that it is too strong to distinguish between statistical correlation of values and casual relationships between high and low object. It is also difficult to be applied to real systems. Gray presented a less general and more detailed elaboration of McLean's flow model in [13], making an explicit connection with information theory through his definition of the channel capacity of flows between confidential and non-confidential variables. Webber [27] defined a property of n-limited security, which took flow-security and specifically prevented downgrading unauthorized information flows. Wittbold and Johnson [28] gave an analysis of certain combinatorial theories of computer security from information-theoretic perspective and introduced non-deducibility on strategies due to feedback. Gavin Lowe [15] measured information flow in CSP by counting refusals. McIver and Morgan [17, 18] devised a new information theoretic

definition of information flow and channel capacity. They added demonic nondeterminism as well as probabilistic choice to *while* program thus deriving a non-probabilistic characterization of the security property for a simple imperative language. There are some other attempts in the 2000s: Di Pierro, Hankin and Wiklicky gave a definition of probabilistic measures on flows in a probabilistic concurrent constraint system where the interference came via probabilistic operators [9–11]. Clarkson etc. suggested a probabilistic beliefs-based approach to non-interference [5, 6]. This work might not works for most of situations, different attackers have different beliefs, the worst case leakage bound is required. Boreale [1] studied the quantitative models of information leakage in the process calculi. Clark, Hunt, and Malacaria [2–4] presented a more complete quantitative analysis but the bounds for loops are imprecise. Malacaria [16] gave a more precise quantitative analysis of loop construct but this analysis is hard to automate.

6 Conclusion and Future Work

Quantitative information flow for security analysis provides a potential way to examine the security properties in computer software. However, much work on quantitative information flow mechanisms lacks a satisfactory account of precise measurement of information released. This problem has not yet been successfully solved. This paper develops an automatic system for quantitative analysis of information flow in a probability distribution sensitive language. In order to give a more precise analysis for loops, we have given the leakage definition with observing time by using the measure of entropy of *generalized probability distributions*. We also have introduced probabilistic semantics to automate such analysis for leakage. The family of techniques for information flow measurement presented in this papaer can be applied to check whether a program meets an information flow security policy by measuring the amount of information revealed by the program. If the information revealed is *small* enough we can be confident in the security of the program, otherwise the analyser's result indicates the program points where the excessive flow occurs.

We are currently developing an approximation on our method. We propose to develop an algorithm to provide an abstract analysis of leakage and also to integrate with bounds on the number of small steps in the operational semantics. Such abstract analysis will enable us to produce more feasible and precise graphs of leakage behavior of a program over time. We also expect to improve our analysis based on the experimental results of the influence of certain parameters over the quality of approximation.

Secondly, we propose to develop a chopped information flow graph for representing information flow of program by capturing and modeling the information flow of data tokens on data slice. Since such graph can show the information flow on data slice explicitly, it can represent well the interesting elementary changes of the information flow of a program and can enhance efficience of the leakage analysis of programs.

Further possible plan is inspired by [21, 22], which suggested a scheme for the backwards abstract interpretation of nondeterministic, probabilistic programs. The idea is that we start from a description of an output event, compute back to the description of the input domains describing their probability distribution of making the behavior happen. This allows the effective computation of upper bounds on the probability of outcomes of the program.

The leakage analysis techniques we considered here concentrates on simple programming languages over semantics. There is still a long way to go before it can be actually used due to the need for formal semantics for full-blown languages and poor runtime performance of the leakage analyser implementation. It is also required to apply the measurement techniques to real programming languages and to operate on large programs without using too much time and computing resources.

Acknowledgments. This research is supported by the EPSRC grant EP/C009967/1 Quantitative Information Flow.

References

1. M. Boreale. Quantifying information leakage in process calculi. In *ICALP (2)*, pages 119–131, 2006.
2. D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *Electronic Notes in Theoretical Computer Science*, volume 59, Elsevier, 2002.
3. D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *J. Log. and Comput.*, 15(2):181–199, 2005.
4. D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15:321–371, 2007.
5. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, pages 31–45, Aix-en-Provence, France, June 2005.
6. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 2007.
7. D. E. R. Denning. A lattice model of secure informatin flow. *Communications of the ACM*, 19(5):236–243, May 1976.
8. D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
9. A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW*, pages 3–17, 2002.
10. A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. *Journal of Computer Security*, 12(1):37–82, 2004.
11. A. Di Pierro, C. Hankin, and H. Wiklicky. Quantitative static analysis of distributed systems. *J. Funct. Program.*, 15(5):703–749, 2005.
12. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, pages 11–20. IEEE Computer Society Press, 1982.
13. J. W. III Gray. Toward a mathematical foundation for informatin flow security. In *IEEE Security and Privacy*, pages 21–35, Oakland, California, 1991.

14. D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
15. G. Lowe. Defining information flow quantity. *Journal of Computer Security*, 12(3-4):619–653, 2004.
16. P. Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 225–235, Nice, France, 2007. ACM Press.
17. A. McIver and C. Morgan. A probabilistic approach to information hiding. In *Programming methodology*, pages 441–460, New York, NY, USA, 2003. Springer-Verlag New York.
18. A. McIver and C. Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004.
19. J. McLean. Security models and information flow. In *Proceeding of the 1990 IEEE Symposium on Security and Privacy*, Oakland, California, May 1990.
20. J. Millen. Covert channel capacity. In *Proceeding 1987 IEEE Symposium on Resarch in Security and Privacy*. IEEE Computer Society Press, 1987.
21. D. Monniaux. Abstract interpretation of probabilistic semantics. In *SAS'00: Proceedings of the 7th International Symposium on Static Analysis*, pages 322–339, London, UK, 2000. Springer-Verlag.
22. D. Monniaux. Backwards abstract interpretation of probabilistic programs. In *ESOP'01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 367–382, London, UK, 2001. Springer-Verlag.
23. A. Renyi. On measures of entropy and information. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability*, pages 547–561, 1961.
24. A. Renyi. *Probability theory*. North-Holland Publishing Company, Amsterdam, 1970.
25. V.A. Rokhlin. Lectures on the entropy theory of measure-preserving transformations. *Russian Mathematical Surveys*, 22(5):1–52, 1965.
26. W. Rudin. *Real and Complex Analysis*. McGraw-Hill, 1966.
27. D. G. Weber. Quantitative hook-up security for covert channel analysis. In *CSFW 1988*, Franconia, NH, 1988. IEEE.
28. J. Todd Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, pages 144–161, 1990.