

Quantitative analysis of secure information flow via Probabilistic Semantics

Chunyan Mu

Department of Computer Science
King's College London
The Strand, London WC2R 2LS
Email: Chunyan.Mu@kcl.ac.uk

David Clark

Department of Computer Science
King's College London
The Strand, London WC2R 2LS
Email: David.J.Clark@kcl.ac.uk

Abstract—We present an automatic analyzer for measuring information flow within software systems. In this paper, 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 an automatic quantitative analyzer based on the leakage definition for such a language. While-loops are handled by applying *entropy of generalized distributions* and related properties in order to provide the analysis with the ability to incorporate the observation of elapsed time.

Index Terms—Language, Security, Non-interference, Semantics, Information Theory, Flow.

I. 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 are seldom concerned with bit leakage or with the program execution observers. It would be good to have a quantitative study geared towards the tasks relevant to the computational environment in which we live. A 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 to information flow security is based on *interference* [10]. Consider *interference* between program variables: informally, this is 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) $\text{if}(h == 0) \text{ then } l := 0 \text{ else } l := 1;$
- 3) $l := h; \text{ while}(l <> 0) l := l * 2;$

where l is a low security variable, 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, Hunt and Malacaria's system for a simple programming language[5] was previously the most complete static quantitative information flow analysis to our knowledge. The main weakness of this work is that the bounds for loops are overly pessimistic. Malacaria [15] gave a precise quantitative analysis of loop constructs using the 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 processing 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 [15] 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 related properties in order to provide a more precise analysis when incorporating elapsed observed time into the analysis. 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 simple imperative (while) language programs. The drawbacks of our approach are that it is lacking abstraction and time complexity can become large in certain circumstances. Empirical testing shows that the critical component in the time complexity is the conditional mutual information calculation. Using the algorithm to generate plots of elapsed time during loop iterations vs. leakage for example can take hours. We are currently researching an abstract analysis for our approach.

The rest of the paper is organized as follows. In Section II, we briefly review the relevant mathematical background. Section III introduces the syntax and the probabilistic semantics, and presents a leakage analyzer based on the semantics. An implementation is given in Section IV. Finally, we present related work and draw conclusions in Section V, VI.

II. MATHEMATICAL BACKGROUND

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

A. Measures 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, probabilities are a particular case of measures. Hence we could apply it to the semantics to drive the distribution’s transformation. 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 [27]. 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 , and 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, $f : M \rightarrow N$ 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 $(E_n)_{n \in \mathbb{N}}$ 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 variables \vec{V} range over 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.

B. 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 sets of 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 is 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:

$$\begin{aligned} \mathcal{I}(X; Y|Z) &= \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X, Y|Z) \\ &= \mathcal{H}(X|Z) - \mathcal{H}(X|Y, Z) \\ &= \mathcal{H}(Y|Z) - \mathcal{H}(Y|X, Z) \end{aligned}$$

C. 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 III). In order to give a complete precise leakage analysis, we allow 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. The loop semantics uses sub-measures for loop approximations, and we need to calculate the entropy of sub-measures. However, Shannon’s entropy definition is not defined for sub-probability measures, therefore we need a more general entropy definition. We now consider some properties and definitions of measures of entropy on the set of *generalized probability distributions* by Renyi given in [24]. 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 an *incomplete random variable*.

The distribution of a generalized random variable is called a *generalized probability distribution*. Specifically, consider a generalized random variable $X = \{x_1, x_2, \dots, x_n\}$, with probability $p_k = \mu\{X = x_k\}$ for $k = 1, 2, \dots, n$, such that the generalized probability distribution is written as $\mathcal{P} = (p_1, p_2, \dots, p_n)$, 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 $\mathcal{P} = (p_1, p_2, \dots, p_n)$, where $0 < \sum_{k=1}^n p_k \leq 1$. $\forall \mathcal{P} \in \Delta$, the *entropy of a generalized probability distribution* $\tilde{\mathcal{H}}(\mathcal{P})$ is defined as:

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

Furthermore, the definition of entropy of partitions presented by Rokhlin [26] and the *partition property* of entropy given by Renyi [25] 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 $E = \{e_{1,1}, \dots, e_{n,m}\}$: $\mu(E_i) = \sum_{1 \leq j \leq m} \mu(e_{i,j})$, and a partition of E in sets $(E_i)_{1 \leq i \leq n}$: $E_i = \{e_{i,1}, \dots, e_{i,m}\}$, the entropy of E can be computed by:

$$\tilde{\mathcal{H}}(\mu(e_{1,1}), \dots, \mu(e_{n,m})) = \tilde{\mathcal{H}}(\mu(E_1), \dots, \mu(E_n)) + \sum_{i=1}^n \mu(E_i) \tilde{\mathcal{H}}\left(\frac{\mu(e_{i,1})}{\mu(E_i)}, \dots, \frac{\mu(e_{i,m})}{\mu(E_i)}\right)$$

where, $\mu(E_i) = \sum_{j=1}^m \mu(e_{i,j})$ ($i = 1, 2, \dots, n$), and by assumption, $\sum_{i=1}^n \mu(E_i) = \sum_{i=1}^n \sum_{j=1}^m \mu(e_{i,j}) \leq 1$. This formula can be considered as a theorem about the information associated with a mixture of distributions. Indeed, the entropy of set E is the information associated with the mixture of the subset (a partition of E) distributions $\frac{\mu(e_{i,1})}{\mu(E_i)}, \dots, \frac{\mu(e_{i,m})}{\mu(E_i)}$ with weights $\mu(E_i)$. The above formula suggests that this information is equal to the sum of the average of the information $\frac{\mu(e_{i,1})}{\mu(E_i)}, \dots, \frac{\mu(e_{i,m})}{\mu(E_i)}$ with weights $\mu(E_i)$ and the information associated with the mixing distribution $(\mu(E_1), \dots, \mu(E_n))$.

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: $\tilde{\mathcal{H}}(\xi) = \tilde{\mathcal{H}}(\eta) + \tilde{\mathcal{H}}(\zeta|\eta)$.

III. THE LEAKAGE ANALYZER

Kozen’s semantics of programs considered as probability distribution transformers and the use of this semantics 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 use Kozen’s probabilistic semantics [12] as the framework to build the relationship between the probability distribution functions and the variables, expressions and 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.

The property of program executions considered here is the notion of distribution on the set of states which induces the computational 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.

A. The Language and its Semantics

The language considered is standard, presented in Table I.

| | | | | |
|--|--------------------|--------------------|---------------------|--------------------|
| $c \in \text{Cmd}$ | $x \in \text{Var}$ | $e \in \text{Exp}$ | $b \in \text{BExp}$ | $n \in \text{Num}$ |
| $c ::= \text{skip} x := e c_1; c_2 \text{if } b \text{ then } c_1 \text{ else } c_2 \text{while } b \text{ do } c$ | | | | |
| $e ::= x n e_1 + e_2 e_1 - e_2 e_1 * e_2 e_1 / e_2$ | | | | |
| $b ::= \neg b e_1 < e_2 e_1 \leq e_2 e_1 = e_2$ | | | | |

TABLE I
THE LANGUAGE

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

$$\begin{aligned} Val &\triangleq \langle \Omega, \mathcal{B}, \mu \rangle & \Sigma &\triangleq \vec{X} \rightarrow Val \\ C[\cdot] &: \text{Cmd} \rightarrow (\Sigma \rightarrow \Sigma) & E[\cdot] &: \text{Exp} \rightarrow (\Sigma \rightarrow Val) \\ B[\cdot] &: \text{BExp} \rightarrow (\Sigma \rightarrow \Sigma) \end{aligned}$$

Kozen [12] 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. We concentrate on the distributions and

| | | |
|---|--------------|---|
| $f_{\llbracket x := e \rrbracket}(\mu)$ | \triangleq | $\lambda W. \mu(f_{\llbracket x := e \rrbracket}^{-1}(W))$ |
| $f_{\llbracket c_1; c_2 \rrbracket}(\mu)$ | \triangleq | $f_{\llbracket c_2 \rrbracket} \circ f_{\llbracket c_1 \rrbracket}(\mu)$ |
| $f_{\llbracket \text{if } b \text{ c}_1 \text{ c}_2 \rrbracket}(\mu)$ | \triangleq | $f_{\llbracket c_1 \rrbracket} \circ f_{\llbracket b \rrbracket}(\mu) + f_{\llbracket c_2 \rrbracket} \circ f_{\llbracket \neg b \rrbracket}(\mu)$ |
| $f_{\llbracket \text{while } b \text{ do } c \rrbracket}(\mu)$ | \triangleq | $f_{\llbracket \neg b \rrbracket}(\lim_{n \rightarrow \infty} (\lambda \mu'. \mu + f_{\llbracket c \rrbracket} \circ f_{\llbracket b \rrbracket}(\mu')^n))(\lambda X. \perp)$ where, $f_{\llbracket B \rrbracket}(\mu) = \lambda X. \mu(X \cap B)$ |

Fig. 1. Probabilistic Denotational Semantics

present the semantic functions by using Lambda Calculus and the notation of inverse function following [19], see Fig. 1.

The semantics can be considered as a distribution transformer. The vector of the program variables $\vec{V} = \{x | x \in \vec{V}\}$ satisfy some *joint distribution* μ on program input, a program $\llbracket C \rrbracket$ maps distribution μ over a \vec{V} to distributions $f_{\llbracket C \rrbracket}(\mu)$ over \vec{V} : $f_{\llbracket C \rrbracket} : \mu \rightarrow \mu$, and context $X : \{\mu : \sigma \mapsto [0, 1]\}$, where σ denotes the *store*. Due to the measurability of the semantic functions, for all measurable $W \in X'$, $f_{\llbracket x := e \rrbracket}(W)$ is measurable in X . The function $f_{\llbracket B \rrbracket}$ for boolean test B defines the set of environments matched by the condition B , which causes the space to split apart. The *conditional statement* is executed on the conditional probability distributions for either the *true* branch or the *false* branch: $f_{\llbracket c_1 \rrbracket} \circ f_{\llbracket b \rrbracket}(\mu) + f_{\llbracket c_2 \rrbracket} \circ f_{\llbracket \neg b \rrbracket}(\mu)$. 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 $f_{\llbracket \text{while } b \text{ do } c \rrbracket}(\mu)$ is thus the sum of all the partitions that finally find their way out. Note that these partitions form that part of the space where the loop partially terminated, which implies the outputs are partially observable. For the case that the loop never terminates, \perp is returned and the leakage is 0 when no new partition is produced but the test is still satisfied. Further details can be found in the following section.

B. 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) . The semantic function for programs maps the state of inputs to the state of outputs. We present the basic leakage definition due to [5] 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.

1) *Arithmetic Expressions:* We denote the *probability distribution function* for a program variable 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 can take. 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:

$$\begin{aligned} \mu_{e(x,y)}(z) &= \sum_{\text{domain}_y} \mu_x(e^{-1}(z, y)) \mu_y(y) \\ &= \sum_{\text{domain}_x} \mu_y(e^{-1}(z, x)) \mu_x(x) \end{aligned}$$

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

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 [5]: 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 $c : \mu_c(c) = 1, \mathcal{H}_c = 0$.

2) *Assignment and random input:* Suppose $\llbracket c \rrbracket : X \rightarrow Y$, X and Y being metric spaces, and W denotes any measurable space in Y , then consider the following linear operator $f_{\llbracket c \rrbracket}$, which is in general presented by inverse image:

$$f_{\llbracket c \rrbracket} : \begin{cases} \mathcal{M}_{\leq 1}(X) \rightarrow \mathcal{M}_{\leq 1}(Y) \\ \mu \mapsto \lambda W. \mu(f_{\llbracket c \rrbracket}^{-1}(W)) \end{cases}$$

Specifically, for the assignment command, 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 :

$$f_{x := e}(\mu) = \lambda X. \mu(f_{\llbracket x := e \rrbracket}^{-1}(X))$$

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

The probabilistic semantics also gives an interesting input semantics to assignment commands such as $x := \text{input}()$. Function $\text{input}()$ 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.

3) *Sequential composition operator:* The distribution transformation function for the *sequential composition operator* is obtained via functional composition:

$$f_{\llbracket c_1; c_2 \rrbracket}(\mu) = f_{\llbracket c_2 \rrbracket} \circ f_{\llbracket c_1 \rrbracket}(\mu)$$

Let $\mu' = f_{\llbracket c_1; c_2 \rrbracket}(\mu)$, the low security variable v is public observable, according to the leakage definition, the leakage

due to sequential command $\llbracket c_1; c_2 \rrbracket$ is the entropy $\mathcal{H}(\mu'_v | \mu_v)$, where μ'_v denotes the distribution of variable v after the sequential command and μ_v denotes the initial one.

Consider a program fragment $\llbracket l := 2; l := h + 3; \rrbracket$ 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

$f_{\llbracket l := 2; l := h + 3; \rrbracket}(\mu_{\langle h, l \rangle}) = \begin{bmatrix} \langle 0, 3 \rangle & w.p. & 0.3 \\ \langle 1, 4 \rangle & w.p. & 0.5 \\ \langle 2, 5 \rangle & w.p. & 0.2 \end{bmatrix}$. The distri-

bution 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 execution of this fragment is the conditional entropy $\mathcal{H}(\mu'_l | \mu_l) = 1.485$, which means all the information contained in h (uncertainty due to the initial distribution of h) flows to output.

4) *Conditional*: A conditional command constructs two branches based on its boolean test since the test partitions the complete distribution on states that existed before execution of the command. We define $\mu_b(\mathbf{tt})$ as the probability that b is true, and $\mu_b(\mathbf{ff})$ as the probability that b is false. Let $\mathcal{P}_0 = \{p_0 = \mu_b(\mathbf{tt})\}$, $\mathcal{P}_1 = \{p_1 = \mu_b(\mathbf{ff})\}$ denote the partitions due to the test b , while $\mathcal{Q}_0^l = \{q_{00}, \dots, q_{0m}\}$, $\mathcal{Q}_1^l = \{q_{10}, \dots, q_{1m}\}$ respectively denote the distribution of the low security variable l in the two branches under the condition that the test b is *true/false*,

$$f_{\llbracket \text{if } b \text{ then } c_1 \text{ else } c_2 \rrbracket}(\mu) = f_{\llbracket c_1 \rrbracket} \circ f_{\llbracket b \rrbracket}(\mu) + f_{\llbracket c_2 \rrbracket} \circ f_{\llbracket \neg b \rrbracket}(\mu)$$

Assuming l is a low security variable and output to public access, the leakage due to the `if` statement is defined as:

$$\mathcal{L}_{\llbracket \text{if} \rrbracket} = \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) + \tilde{\mathcal{H}}(\mathcal{Q}_0^l \cup \mathcal{Q}_1^l | \mathcal{P}_0 \cup \mathcal{P}_1)$$

Example Consider the following program. We only consider a k -bit variable with possible values $0, \dots, 2^k - 1$, *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 straightforward 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 probability distribution transformation is obtained as:

$$f_{\llbracket \text{if} \rrbracket} \left(\langle h, l \rangle \mapsto \begin{bmatrix} \langle 0, \perp \rangle & w.p. & 1/2^{32} \\ \dots & \dots & \\ \langle 2^{32} - 1, \perp \rangle & w.p. & 1/2^{32} \end{bmatrix} \right) \\ = \left(l \mapsto \begin{bmatrix} 0 & w.p. & 1/2^{32} \\ 1 & w.p. & 1 - 1/2^{32} \end{bmatrix} \right)$$

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 that flowed to the low component under this

example can be computed by:

$$\begin{aligned} \mathcal{H}_{\llbracket \text{if} \rrbracket} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) + \tilde{\mathcal{H}}(\mathcal{Q}_0^l \cup \mathcal{Q}_1^l | \mathcal{P}_0 \cup \mathcal{P}_1) \\ &= \tilde{\mathcal{H}}(\{\frac{1}{2^{32}}\} \cup \{1 - \frac{1}{2^{32}}\}) + 0 = 7.8 \times 10^{-9} \end{aligned}$$

The result shows that this example just releases little information to public access, which agrees with our intuition: the probability of $h = 0$ is quite low and the uncertainty in h under condition $h \neq 0$ is still great, *i.e.* little information is released.

5) *While Loop*: The semantic function for a loop is given by:

$$f_{\llbracket \text{while } b \text{ do } c \rrbracket}(\mu) = f_{\llbracket \neg b \rrbracket} \left(\lim_{n \rightarrow \infty} (\lambda \mu'. \mu + f_{\llbracket c \rrbracket} \circ f_{\llbracket b \rrbracket}(\mu'))^n (\lambda X. \perp) \right)$$

Such function produces the union of all of 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 by this iteration, *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,

$$\mathcal{P}(\llbracket \text{while } b \text{ c} \rrbracket) = \bigcup_{0 \leq i \leq k} \mathcal{P}_i(\llbracket \text{while } b \text{ c} \rrbracket)$$

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

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

i.e. 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

$$\begin{aligned} \mathcal{P} &= (\mathcal{P}_0 \cup \mathcal{P}_1 \cup \dots \cup \mathcal{P}_k)_{0 \leq k \leq n} \\ &= (\{p_0\} \cup \{p_1\} \cup \dots \cup \{p_k\})_{0 \leq k \leq n} \end{aligned}$$

the events $\mathcal{P}_0, \dots, \mathcal{P}_k$ 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 and 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-terminating with regard to this part of the space.

Next we produce the leakage definition for the loop command as induced by 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 [24], let Δ denote the set of all finite discrete generalized probability distributions, if $\mathcal{P}_1 \in \Delta, \mathcal{P}_2 \in \Delta, \dots, \mathcal{P}_n \in \Delta$, such that $\sum_{k=1}^n W(\mathcal{P}_k) \leq 1$, we have:

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

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 $\tilde{\mathcal{H}}(\mathcal{Q}|\mathcal{P})$. 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 $e_{0 \leq i \leq k}^i$ has taken place, we have:

$$\begin{aligned} \mathcal{Q}^L &= (\mathcal{Q}_0^L \cup \mathcal{Q}_1^L \cup \dots \cup \mathcal{Q}_k^L)_{0 \leq k \leq n} \\ &= (\{q_{00}, \dots, q_{0j}\} \cup \dots \cup \{q_{k0}, \dots, q_{kj}\})_{0 \leq k \leq n} \end{aligned}$$

where k is interpreted as ‘‘up to the k^{th} iteration’’. By applying the formula of conditional entropy for *generalized* probability distributions [24], and letting W denote *weight*, we have:

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

As we discussed above, the loop command creates a set of sub-measures (incomplete distributions). To give a more precise leakage analysis, we allow 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 or even intermediate states in the computation. The 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 elapsed time (e.g. iterations) for loops incorporated into 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:

$$\begin{aligned} \mathcal{L}_{\text{while}}(k) &= \tilde{\mathcal{H}}(\mathcal{P}) + \tilde{\mathcal{H}}(\mathcal{Q}^L|\mathcal{P}) \\ &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_k) + \tilde{\mathcal{H}}(\mathcal{Q}_0^L \cup \dots \cup \mathcal{Q}_k^L|\mathcal{P}_0 \cup \dots \cup \mathcal{P}_k) \\ &= \frac{\sum_{i=0}^n (p_i \log_2 \frac{1}{p_i})}{\sum_{i=0}^n p_i} + \frac{\sum_{i=0}^n \sum_{j=0}^m q_{ij} \log_2 \frac{q_{ij}}{p_i}}{\sum_{i=0}^n \sum_{j=0}^m q_{ij}} \end{aligned}$$

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

- case $0 \leq k < n$: $\mathcal{P} = \mathcal{P}_0 \cup \dots \cup \mathcal{P}_k$ and $\mathcal{Q}^L = \mathcal{Q}_0^L \cup \dots \cup \mathcal{Q}_k^L$ are two *incomplete* distributions, such that, $W(\mathcal{P}_0) + \dots + W(\mathcal{P}_k) < 1$, and $W(\mathcal{Q}_0^L) + \dots + W(\mathcal{Q}_k^L) < 1$. With regard to this case, part of the space has exited the loop but not all, we therefore can compute the leakage by observing each iteration before the loop terminates. An intuition behind the introduction of the notion of entropy of incompleting loops is that the term $\log_2(1/p_k)$ in Shannon’s formula 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 $\mathcal{H}(p_1, \dots, p_n) = \sum_{k=1}^n p_k \log_2(\frac{1}{p_k})$ is actually a mean value.
- case $k = n$: $\mathcal{P} = \mathcal{P}_0 \cup \dots \cup \mathcal{P}_n$ and $\mathcal{Q}^L = \mathcal{Q}_0^L \cup \dots \cup \mathcal{Q}_n^L$ are two *complete* distributions, such that, $W(\mathcal{P}_0) + \dots + W(\mathcal{P}_k) = 1$, and $W(\mathcal{Q}_0^L) + \dots + W(\mathcal{Q}_k^L) = 1$, i.e. $\sum_{i=0}^n p_i = \sum_{i=0}^n \sum_{j=0}^m q_{ij} = 1$, and all parts of the space exit the loop.
- case $k = \infty$: this is the case of nonterminating (partially or totally) loops, our tool returns the partitions which do make the loop terminate (case of partial termination) and also the remaining partition which cannot produce new partitions but still satisfies test b .

Proposition 1 presents the relationship between our leakage definition for loops with Malacaria’s [15] definition. We show that our algorithm calculates the same *final* quantity (when the loop terminates) as Malacaria’s method, providing correctness for his method relative to our presentation of 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 any while language program. Furthermore, our leakage is *variant* with observed elapsed 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 as it accumulates over elapsed time, i.e. how much information is leaked up until any iteration of interest (depending on the observational power of the observer).

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 [15].

Proof: $\mathcal{L}_{\text{while}}(n)$ 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:

$$\begin{aligned} \mathcal{L}_{\text{while}}(n) &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_n) + \tilde{\mathcal{H}}(\mathcal{Q}_0^L \cup \dots \cup \mathcal{Q}_n^L|\mathcal{P}_0 \cup \dots \cup \mathcal{P}_n) \\ &= \frac{W(\mathcal{P}_0)\tilde{\mathcal{H}}(\mathcal{P}_0) + \dots + W(\mathcal{P}_n)\tilde{\mathcal{H}}(\mathcal{P}_n)}{W(\mathcal{P}_0) + \dots + W(\mathcal{P}_n)} + \\ &= \frac{W(\mathcal{Q}_0^L)\tilde{\mathcal{H}}(\mathcal{Q}_0^L|\mathcal{P}_0) + \dots + W(\mathcal{Q}_k^L)\tilde{\mathcal{H}}(\mathcal{Q}_k^L|\mathcal{P}_k)}{W(\mathcal{Q}_0^L) + \dots + W(\mathcal{Q}_k^L)} \end{aligned}$$

$$\begin{aligned}
&= \frac{p_0 \frac{p_0 \log_2 \frac{1}{p_0}}{p_0} + \dots + p_n \frac{p_n \log_2 \frac{1}{p_n}}{p_n}}{p_0 + p_1 + \dots + p_n} + \\
&\quad \frac{\sum_{i=0}^n \sum_{j=0}^m q_{ij} \log_2 \frac{q_{ij}}{p_i}}{\sum_{i=0}^n \sum_{j=0}^m q_{ij}} \\
&= \frac{\sum_{i=0}^n (p_i \log_2 \frac{1}{p_i})}{\sum_{i=0}^n p_i} + \sum_{i=0}^n p_i \sum_{j=0}^m \frac{q_{ij}}{p_i} \log_2 \frac{q_{ij}}{p_i} \\
&= \mathcal{H}(p_0, \dots, p_n) + \sum_{i=0}^n p_i \mathcal{H}\left(\frac{q_{i0}}{p_i}, \frac{q_{i1}}{p_i}, \dots, \frac{q_{im}}{p_i}\right)
\end{aligned}$$

The above demonstrates that, when $k = n$, our leakage definition is equivalent to Malacaria's leakage definition for collision free loops [15]. For the case of semantic function $f : X \rightarrow Y$ with collisions in Malacaria's definition, we make an adjustment to Y to eliminate the collisions, deriving 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. ■

a) *A terminating example:*

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

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

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

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

$$\begin{aligned}
&f_{[\text{while}]} \left(\langle h, l \rangle \mapsto \left[\begin{array}{ll} \langle 0, 0 \rangle & \text{w.p. } 7/8 \\ \langle 1, 0 \rangle & \text{w.p. } 1/56 \\ \dots & \dots \\ \langle 7, 0 \rangle & \text{w.p. } 1/56 \end{array} \right] \right) \\
&= \langle h, l \rangle \mapsto \left[\begin{array}{ll} \langle 0, 0 \rangle & \text{w.p. } 7/8 \\ \langle 1, 1 \rangle & \text{w.p. } 1/56 \\ \dots & \dots \\ \langle 7, 7 \rangle & \text{w.p. } 1/56 \end{array} \right]
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{L}_{\text{while-0}} &= \tilde{\mathcal{H}}(\mathcal{P}_0) = 0.192645 \\
\mathcal{L}_{\text{while-1}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1) = 0.304939275 \\
\mathcal{L}_{\text{while-2}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2) = 0.412829778 \\
\mathcal{L}_{\text{while-3}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_3) = 0.516570646 \\
\mathcal{L}_{\text{while-4}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_4) = 0.616396764 \\
\mathcal{L}_{\text{while-5}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_5) = 0.71252562 \\
\mathcal{L}_{\text{while-6}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_6) = 0.805158879 \\
\mathcal{L}_{\text{while-7}} &= \tilde{\mathcal{H}}(\mathcal{P}_0 \cup \dots \cup \mathcal{P}_7) = 0.894483808
\end{aligned}$$

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).

b) *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:

$$\mu_{\langle h, l \rangle} \mapsto \left[\begin{array}{ll} \langle 0, 2 \rangle \text{ w.p. } \frac{7}{8} & A \\ \langle 1, 2 \rangle \text{ w.p. } \frac{1}{56} & \dots \\ \dots & \dots \\ \langle 7, 2 \rangle \text{ w.p. } \frac{1}{56} & B \end{array} \right]$$

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: $\mathcal{L} = \tilde{\mathcal{H}}(6/56) = 3.22$ otherwise the leakage is computed by: $\mathcal{L} = \tilde{\mathcal{H}}(6/56, 50/56) = 0.491$. 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.

c) *A non-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:

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

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: $\mathcal{L} = \tilde{\mathcal{H}}(\perp) = 0$, this also meets our intuition: no information is leaked by a non-terminated loop (assuming non-termination is not observable).

IV. AN IMPLEMENTATION

For a feasibility study, we have implemented the method described above which can be used to calculate mutual information between (sets of) variables at any program point

automatically. Fig. 2 describes the basic structure of the system. The initial configuration (joint distribution for variables \vec{V}) and the example program are fed into the analyzer, the analyzer will present the distribution transformation and the measurement of the secret information leaked to low component at each program point by executing the program.

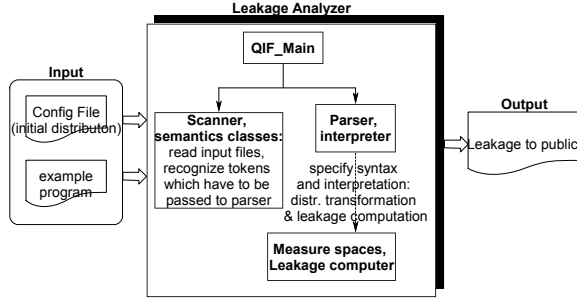


Fig. 2. Implementation: structure

Experiment 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);
```

By applying the analysis rules given in [5], 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 3 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 analyzer. The analyzer transforms the joint distribution

by analyzing the program. Note that at the end of each iteration, l is output to public. The analyzer thus computes the leakage at each time(iteration) point(t) \mathcal{L}_t by applying the definition of leakage presented in Section III, 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.

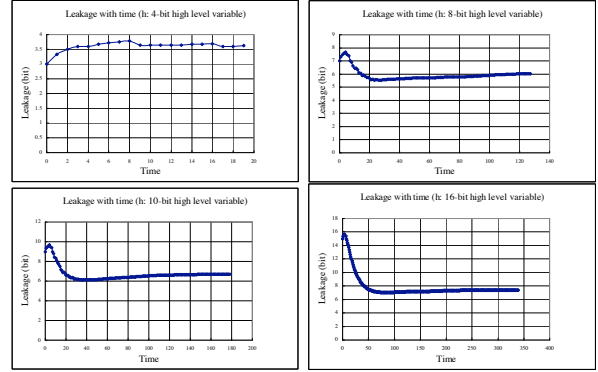


Fig. 3. Results of the experiment

V. RELATED WORK

There has been significant activity 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 [9] suggested that the data manipulated by a program can be typed with security levels, which naturally assumes the structure of a partially ordered set. Moreover, this partially ordered set is a lattice under certain conditions [8]. Denning first explored the use of information theory as the basis for a quantitative analysis of information flow in programs. However, she did not suggest how to automate the analysis or attempt to make the analysis formal and complete. In 1987, Millen [18] 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 [17], 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 causal relationships between high and low objects. It is also difficult to apply to real systems as it is quite abstract. Gray presented a more detailed and formal elaboration of McLean's flow model in [11], making an explicit connection with information theory through his definition of the channel capacity for flows between confidential and non-confidential

variables. Webber [28] defined a property of n -limited security, which took flow-security and specifically prevented downgrading unauthorized information flows. Wittbold and Johnson [29] 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 [13], [14] measured information flow in CSP by counting refusals. McIver and Morgan [16] devised a new information theoretic definition of information flow and channel capacity. They added demonic non-determinism 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 related researches in the 2000s: Pierro, Hankin and Wiklicky gave a definition of probabilistic measures on flows for a probabilistic concurrent constraint system where the interference came via probabilistic operators [21], [1], [22], [23]. Clarkson et alia suggested a probabilistic beliefs-based approach to non-interference [6], [7]. This work may not be useful in some situations, since different attackers have different beliefs and the worst case leakage bound is required. Boreale [2] studied quantitative models of information leakage in a process calculi using an approach based on Clark, Hunt and Malacaria [3], [4], [5] presented a more complete quantitative analysis but the bounds for loops are imprecise. Malacaria [15] gave a more precise quantitative analysis of loop construct but this analysis is hard to automate.

VI. CONCLUSION AND FUTURE WORK

Quantitative information flow for security analysis provides a powerful, flexible way to examine security properties based on information flow for computer software. However, much work on quantitative information flow mechanisms has lacked a satisfactory account of the precise measurement of any information released. This problem has hitherto not been successfully solved. This paper has developed 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 for observed elapsed time by using the measure of entropy *for generalized probability distributions*. We also have used a probabilistic semantics to automate such analysis for leakage. The family of techniques for information flow measurement presented in this paper 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 analyzer'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 abstraction interpretation based analysis of leakage and also to integrate this with bounds on the number of small steps in the operational semantics. Such analysis will enable us to produce graphs of leakage behavior of a program over time in

a computationally feasible way and allow control of the level of precision in the analysis. 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 efficiency of the leakage analysis of programs.

A further possible direction can be provided by [19], [20], 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 analyzer 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.

ACKNOWLEDGMENT

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

REFERENCES

- [1] Alessandro Aldini and Alessandra Di Pierro. A quantitative approach to noninterference for probabilistic systems, 2003.
- [2] Michele Boreale. Quantifying information leakage in process calculi. In *ICALP (2)*, pages 119–131, 2006.
- [3] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. In *Electronic Notes in Theoretical Computer Science*, volume 59, Elsevier, 2002.
- [4] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *J. Log. and Comput.*, 15(2):181–199, 2005.
- [5] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15:321–371, 2007.
- [6] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, pages 31–45, Aix-en-Provence, France, June 2005.
- [7] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 2007.
- [8] Dorothy Elizabeth Robling Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [9] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
- [10] 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.
- [11] James W. III Gray. Toward a mathematical foundation for information flow security. In *IEEE Security and Privacy*, pages 21–35, Oakland, California, 1991.
- [12] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

- [13] Gavin Lowe. Quantifying information flow. In *Proceedings IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.
- [14] Gavin Lowe. Defining information flow quantity. *Journal of Computer Security*, 12(3-4):619–653, 2004.
- [15] Pasquale 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.
- [16] Annabelle McIver and Carroll Morgan. A probabilistic approach to information hiding. In *Programming methodology*, pages 441–460, New York, NY, USA, 2003. Springer-Verlag New York.
- [17] John McLean. Security models and information flow. In *Proceeding of the 1990 IEEE Symposium on Security and Privacy*, Oakland, California, May 1990.
- [18] Jonathan Millen. Covert channel capacity. In *Proceeding 1987 IEEE Symposium on Resarch in Security and Privacy*. IEEE Computer Society Press, 1987.
- [19] David 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.
- [20] David 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.
- [21] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *CSFW*, pages 3–17, 2002.
- [22] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. *Journal of Computer Security*, 12(1):37–82, 2004.
- [23] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Quantitative static analysis of distributed systems. *J. Funct. Program.*, 15(5):703–749, 2005.
- [24] Alfred Renyi. On measures of entropy and information. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability*, pages 547–561, 1961.
- [25] Alfred Renyi. *Probability theory*. North-Holland Publishing Company, Amsterdam, 1970.
- [26] Vladimir Abramovich Rokhlin. Lectures on the entropy theory of measure-preserving transformations. *Russian Mathematical Surveys*, 22(5):1–52, 1965.
- [27] Walter Rudin. *Real and Complex Analysis*. McGraw-Hill, 1966.
- [28] Douglas G. Weber. Quantitative hook-up security for covert channel analysis. In *CSFW 1988*, Franconia, NH, 1988. IEEE.
- [29] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, pages 144–161, 1990.