# Automation of Quantitative Information-Flow Analysis

Boris Köpf[1] and Andrey Rybalchenko[2]

[1] IMDEA Software Institute, Spain
[2] Technische Universität München, Germany
boris.koepf@imdea.org
rybal@in.tum.de

**Abstract.** Quantitative information-flow analysis (QIF) is an emerging technique for establishing information-theoretic confidentiality properties. Automation of QIF is an important step towards ensuring its practical applicability, since manual reasoning about program security has been shown to be a tedious and expensive task. In this chapter we describe a approximation and randomization techniques to bear on the challenge of sufficiently precise, yet efficient computation of quantitative information flow properties.

## 1 Introduction

The goal of an information-flow analysis is to keep track of sensitive information during computation. If a program does not expose any information about its secret inputs to unauthorized parties, it has secure information flow, a property that is often formalized as noninterference. In many cases, achieving noninterference is expensive, impossible, or simply unnecessary: Many systems remain secure as long as the amount of exposed secret information is sufficiently small. Consider for example a password checker. A failed login attempt reveals some information about the secret password. However, for well-chosen passwords, the amount of leaked information is so small that a failed login-attempt will not compromise the security of the system.

Quantitative information-flow analysis (QIF) is a technique for establishing bounds on the information that is leaked by a program. The insights that QIF provides go beyond the binary output of Boolean approaches, such as non-interference analyzers. This makes QIF an attractive tool to support gradual development processes, even without explicitly specified policies. Furthermore, because information-theory forms the foundation of QIF, the quantities that QIF delivers can be directly associated with operational security guarantees, such as lower bounds for the expected effort of uncovering secrets by exhaustive search.

Automation of QIF is an important step towards ensuring its practical applicability, since manual reasoning about program security has been shown to be a tedious and expensive task [45]. Technically, successful automation of QIF must determine tight, yet efficiently computable bounds on the information-theoretic characteristics of the program. For deterministic programs with uniformly distributed inputs, these characteristics can be expressed by a partition of the secret program inputs. In this partition, each block is defined by the preimage of some program output. The computation of some information-theoretic characteristics, e.g., Shannon entropy, from this partition

requires the enumeration of all blocks and the estimation of their respective sizes. Other measures, e.g., min-entropy, only depend on the number of blocks in the partition. Exact computation for both kinds of characteristics is prohibitively hard, thus suggesting the exploration of approximation-based approaches. In the presence of approximation, characterizing the deviation from the exact result becomes an important question.

In this chapter, we describe approximation and randomization techniques to tackle the challenge of automating QIF for deterministic programs. The presented approach avoids the trap of block enumeration by a sampling method that uses the program itself to randomly choose blocks with probabilities corresponding to their relative sizes. Each sample amounts to a program execution and indexes a block by the corresponding program output. We obtain an under-approximation for each block using symbolic execution and symbolic backward propagation along the sequence of program statements traversed during a sample run. We obtain an over-approximation of each block in two steps. First, we transform the given program such that the input state becomes explicitly available in the set of reachable program states by memorizing it in an auxiliary variable. Second, an over-approximation of the reachable states of the transformed program that we obtain by applying abstract interpretation [24] delivers an over-approximation of blocks indexed by program outputs. Finally, we use the indexing by program outputs to put together under- and over-approximations for each sampled block. Thus, we obtain the necessary ingredients for the computation of information-theoretic guarantees, namely, lower and upper bounds for the remaining uncertainty about the program input.

The distinguishing feature of the presented technique is that it ensures fast convergence of the sampling process and provides formal guarantees for the quality of the obtained bounds. The proof builds upon a result by Batu et al. [9] stating that the entropy of a random variable can be estimated accurately and with a high confidence level using only a small number of random samples. Since Batu et al. [9] require an oracle revealing the probability of each sample of the random variable, in the first step towards the QIF setting we identify a correspondence between the sampling oracle and the preimage computation for the program whose QIF properties are analyzed. In the second step, we prove that the confidence levels and convergence speed of the exact setting, which relies on the actual preimages, can also be obtained in the approximative setting where only over- and under-approximations of preimages are known. This result allows our approach to replace the exhaustive analysis of all, say $n$-many, possible preimages with the treatment of a randomly chosen set of $O((\log n)^2)$ preimage samples.

This chapter extends and generalizes the results from [41]. In particular, the chapter contains a discussion of non-uniformly distributed secrets and adversarially chosen inputs.

**Outline** In the next section, we illustrate our approach on example programs. Then, we give basic definitions in Section 3. In Section 4, we present over- and under-approximation techniques. Section 5 describes how randomization combines over- and under-approximations to deliver a quantitative information flow analysis. Finally, we discuss related work in Section 6.

## 2 Illustration

We illustrate our method on two example programs whose quantitative information-flow analysis is currently out of reach for the existing automatic approaches. The examples present the computation of Shannon entropy and min-entropy, respectively, and require dealing with loops and data structures, which our method handles automatically using approximation and randomization techniques. We computed certain intermediate assertions for the following examples using BoundGen, an automatic tool for the discovery of resource usage bounds [23].

### 2.1 Estimating Shannon entropy

As a first example we consider the electronic purse program from [2] as shown below.

```
1    l = 0;
2    // assume(h < 20);
3    while(h>=5){
4      h = h-5;
5      l = l+1;
6    }
```

The program receives as input the nonnegative balance of a bank account in the integer variable h and debits a fixed amount of 5 from this account until the balance is insufficient for this transaction, i.e., until $h < 5$.

Our goal is to determine the remaining uncertainty about the initial value of h after learning the final value of l, where we use Shannon entropy as a measure of uncertainty. A high remaining uncertainty implies a large lower bound on the expected number of steps that are required for determining the initial value of h by brute-force search [47], which corresponds to a formal, quantitative confidentiality guarantee.

For uniformly distributed inputs, one can express the remaining Shannon entropy as a weighted sum of the logarithms of the sizes of the preimages of the program [38]. A large average preimage size corresponds to a large remaining uncertainty about the input and implies a large expected effort for determining the program's input after observing the output.

One way to precisely compute the remaining Shannon entropy is to compute the partition induced by the preimages of the program, which may require the enumeration of all pairs of program paths. Moreover, it requires the enumeration of all blocks in the computed partition [2]. Both enumeration problems severely limit the size of the systems that can be analyzed using this precise approach.

For the purse program and nonnegative h, the partition induced by the preimages of the program is

$$\{\{0,1,2,3,4\},\{5,6,7,8,9\},\dots\}$$

and is represented by the following formula that states that two initial values, say $h$ and $\overline{h}$, are in the same block.

$$\exists k \geq 0 : 5k \leq h \leq 5k+4 \wedge 5k \leq \overline{h} \leq 5k+4$$

The quantified variable $k$ corresponds to the number of loop iterations that the program executes when started on $h$ and $\overline{h}$. Such existentially quantified assertions are out of reach for the existing automatic tools for reasoning about programs. The current approaches simplify the problem by (implicitly) bounding the number of loop iterations and replacing the existential quantification with finite disjunction.

For example, the algorithm from [2] bounds the number of loop iterations by making finite the set of possible valuations of `h`, which is achieved by introducing the assumption in line 2. With this restriction, we obtain the following characterization of the partition.

$$
\begin{aligned}
&(0 \leq h \leq 4 \wedge 0 \leq \overline{h} \leq 4) \vee \\
&(5 \leq h \leq 9 \wedge 5 \leq \overline{h} \leq 9) \vee \\
&(10 \leq h \leq 14 \wedge 10 \leq \overline{h} \leq 14) \vee \\
&(15 \leq h \leq 19 \wedge 15 \leq \overline{h} \leq 19)
\end{aligned}
$$

As mentioned above, such solutions are only partial and overly restrictive, since the program properties derived for values below the loop bound do not necessarily carry over beyond this limit.

In this chapter, we show that the precise computation of each preimage can be replaced by the computation of the under- and over-approximation of the preimage. We also show that, by running the program on randomly chosen inputs and approximating only the preimages of the corresponding outputs, one can derive upper and lower bounds for the remaining uncertainty about the program's secret input. These bounds are valid with a provably high level of confidence, for a number of samples that is poly-logarithmic in the size of the input domain.

**Approximation** To compute *over-approximations* of preimages, we augment the program by declaring copies of input variables (so-called *ghost variables*) and adding an assignment from inputs into these copies as the first program statement. In our program, we declare a new variable `_h` and insert the initialization statement `_h = h;` before line 1. Let $\overline{F}_{reach}$ be the set of reachable final states of the augmented program. $\overline{F}_{reach}$ keeps track of the relation between the ghost variables and the output of the original program. As the ghost variables are not modified during computation, this set corresponds to the input-output relation $\rho_{IO}$ of the original program, i.e., $\overline{F}_{reach} = \rho_{IO}$. While the exact computation of $\overline{F}_{reach}$ is a difficult task, we can rely on abstract interpretation techniques for computing its over-approximation [24]. In particular, we can bias the over-approximation towards the discovery of the relation between the ghost variables and low outputs by using constraints and borrow existing implementations originally targeted for resource bound estimation, e.g. [23, 32]. We apply the bound generator BoundGen [23] and obtain

$$
\overline{F}_{reach}{}^{\sharp} \quad = \quad -5 + 5\,\mathtt{l} \leq \mathtt{\_h} \leq -1 + 5\,\mathtt{l}\ .
$$

The predicate $\overline{F}_{reach}{}^{\sharp}$ represents an over-approximation of the input-output relation $\rho_{IO}$. (Here, the outcome happens to be a precise description.) Hence for each low output value `l`, the set of ghost input values `_h`, such that `l` and `_h` are related by $\overline{F}_{reach}{}^{\sharp}$, over-approximates the preimage of `l` with respect to the original program. The size of these

approximated preimages can be determined using tools for counting models, e.g., we use LattE [43] for dealing with linear arithmetic assertions. In our example, we obtain 5 as an upper bound for the size of the preimage of each value of `l`.

To compute *under-approximations* of preimages, we symbolically execute the program on a randomly chosen input and determine the preimage with respect to the obtained path through the program. This computation relies on the relation between program inputs and outputs along the path that the execution takes. We establish this relation by combining the transition relations of individual steps. This technique can be efficiently automated using existing symbolic execution engines both at the source code level, e.g., KLEE [14] and DART [29], and at the binary level, e.g., BitScope [12].

For example, for an input of $\mathtt{h} = 37$, this relation is determined to be

$$35 \leq \mathtt{h} \leq 39 \wedge \mathtt{l} = 8 \,.$$

(Again, the result happens to be a precise description.) Hence, the preimage size of $\mathtt{l} = 8$ is at least 5.

**Randomization** The direct approximation of the leak as described above requires the computation of bounds for the size of each preimage. Note that the number of preimages can be as large as the input domain (e.g. when the program's output fully determines the input), which severely limits scalability. We overcome this limitation using a randomized approach, where we run the program on randomly chosen inputs and approximate only the preimages of the corresponding outputs. Leveraging a result from [9], we show how this set of preimages can be used for deriving bounds on the information-theoretic characteristics of the entire program. These bounds are valid with a provably high level of confidence, for a number of samples that is logarithmic in the size of the input domain.

Technically, we show that, for an arbitrary $\delta > 0$, the remaining uncertainty $H$ about the secret input is bounded by the following expression

$$\frac{1}{n}\sum_{i=1}^{n} \log_2 m_i^\flat - \delta \;\leq\; H \;\leq\; \frac{1}{n}\sum_{i=1}^{n} \log_2 m_i^\sharp + \delta \,,$$

where $n$ is the number of samples and $m_i^\flat$ and $m_i^\sharp$ are the upper and lower bounds for the size of the preimage corresponding to the $i$th sample. If the secret input is chosen from a set $I$, these bounds hold with a probability of more than $p$ for a number of samples $n$ such that

$$n = \frac{(\log_2 \#(I))^2}{(1-p)\delta^2} \,.$$

For our example and $I = \{0, \ldots, 2^{64} - 1\}$, our analysis delivers coinciding lower and upper bounds of 5 for the sizes of the preimages (except for the preimage containing $2^{64} - 1$, which is smaller). As a consequence, we obtain entropy bounds of

$$\log_2 5 - 0.1 \leq H \leq \log_2 5 + 0.1$$

that hold with a probability of more than 0.99 when considering $10^8$ samples.

### 2.2 Estimating min-entropy

The following program implements an algorithm for the bit-serial modular exponentiation of integers. More precisely, the program computes $\mathtt{x}^{\mathtt{k}} \mod \mathtt{n}$, where n is the constant modulus and k is maintained by the program in a binary array of constant length len.

```
1    int m = 1;
2    for (int i = 0; i<len; i++) {
3      m = m*m mod n;
4      if ( k[i] == 1 ) {
5        m = m*x mod n;
6      }
7    }
```

Due to the conditional execution of the multiplication operation m=m*x mod n in line 5, the running time of this program reveals information about the entries of the array k. Such variations have been exploited to recover secret keys from RSA decryption algorithms based on structurally similar modular exponentiation algorithms [37]. We analyze an abstract model of the timing behavior of this program, where we assume that each multiplication operation consumes one time unit. We make this model explicit in the program semantics by introducing a counter time that is initialized with 0 and is incremented each time a multiplication operation takes place.

Our goal is to quantify the remaining min-entropy about the content of k after observing the program's execution time, i.e., given the final value of time. The min-entropy captures the probability of correctly guessing a secret at the first attempt. In contrast to Shannon entropy, computation of min-entropy does not require the enumeration of blocks and estimation of their sizes. For min-entropy, we only need to estimate how many blocks (or alternatively how many possible outputs) the program can produce [59].

This simplification can be exploited when dealing with programs that manipulate data structures. As the example shows, applications often keep secret the content of data structures, while some of their properties, e.g., list length or even number of elements satisfying a Boolean query, are revealed as outputs. In such cases, our method can estimate the input's remaining min-entropy despite the difficulties of automatic reasoning about data structures. To succeed, our method applies the over- and under-approximation techniques presented in Section 2.1, however without the addition of ghost variables. No ghost variables are needed, since the actual block content given by the secret data structure elements is irrelevant for the min-entropy computation.

We extend a result from [59] to show that under-approximations of the remaining min-entropy can be computed from over-approximations of the size of the set of reachable states $F_{reach}$, which in our example is the set of possible values of the variable time. By applying the bound generator BoundGen [23], we obtain the following over-approximation ${F_{reach}}^{\sharp}$ of $F_{reach}$,

$$ {F_{reach}}^{\sharp} \;\equiv\; \mathtt{len} \leq \mathtt{time} \leq 2\mathtt{len}\,, $$

which shows that $\#(F_{reach}{}^{\sharp}) \leq \texttt{len} + 1$. We use this bound to infer that, after observing `time`, the remaining uncertainty about the exponent `k` is still larger than

$$\log_2 \frac{2^{\texttt{len}}}{\texttt{len}+1} = \texttt{len} - \log_2(\texttt{len}+1)\,,$$

given that `k` is drawn uniformly from $\{0, \ldots, 2^{\texttt{len}} - 1\}$. An alternative interpretation is that the expected reduction in uncertainty about the exponent `k` is at most $\log_2(\texttt{len}+1)$ bits.

## 3 Preliminaries

In this section, we give the necessary definitions for dealing with programs and information-flow analysis.

### 3.1 Programs and computations

Following [46] we treat programs as transition systems, and rely on existing translation procedures from programs written in particular programming languages to transition systems, e.g., [36]. A program $P = (S, I, \mathcal{T})$ consists of the following components.

- $S$ - a set of *states*.
- $I \subseteq S$ - a set of *initial* states.
- $\mathcal{T}$ - a finite set of *transitions* such that each transition $\tau \in \mathcal{T}$ is given a binary *transition relation* over states, i.e., $\rho_\tau \subseteq S \times S$.

For a program represented as source code, states are determined by the valuation of the declared program variables and the program counter, and transition correspond to program statements.

Let $F$ be the set of *final* program states that do not have any successors, i.e.,

$$F \;=\; \{s \in S \mid \forall s' \in S\ \forall \tau \in \mathcal{T} : (s, s') \notin \rho_\tau\}\,.$$

A *computation* of $P$ is a sequence of program states $s_1, \ldots, s_n$ such that $s_1$ is an initial state, i.e., $s_1 \in I$, $s_n$ is a final state, i.e., $s_n \in F$, and each pair $s$ and $s'$ of consecutive states follows a program transition, i.e., $(s, s') \in \rho_\tau$ for some $\tau \in \mathcal{T}$. We assume that final states do not have any successors, i.e., there is no pair of states $s$ and $s'$ such that $s \in F$ and $(s, s') \in \rho_\tau$ for some $\tau \in \mathcal{T}$.

A program is *deterministic* if for each state $s$ there is at most one transition that assigns a successor to $s$ and there is at most one such successor. Formally,

$$\forall s\ \forall s'\ \forall s''\ \forall \tau\ \forall \tau' : ((s, s') \in \rho_\tau \wedge (s, s'') \in \rho_{\tau'}) \rightarrow (s' = s'' \wedge \tau = \tau')\,.$$

In this chapter we only consider deterministic programs.

A *path* is a sequence of transitions. We write $\epsilon$ for the *empty* path, i.e., the path of length zero. Let $\circ$ be the *relational composition* function for binary relations, i.e., for binary relations $X$ and $Y$ we have $X \circ Y \;=\; \{(x, y) \mid \exists z : (x, z) \in X \wedge (z, y) \in Y\}$. Then,

a *path relation* is a relational composition of transition relations along the path, i.e., for $\pi = \tau_1, \ldots, \tau_n$ we have $\rho_\pi = \rho_{\tau_1} \circ \ldots \circ \rho_{\tau_n}$. A path $\pi$ is *feasible* if its path relation is not empty, i.e., $\rho_\pi \neq \emptyset$.

Let $\rho$ be the *program transition relation* defined as follows.

$$\rho \quad = \quad \bigcup_{\tau \in \mathcal{T}} \rho_\tau$$

We write $\rho^*$ for the transitive closure of $\rho$. The *input-output* relation $\rho_{IO}$ of the program relates each initial state $s$ with the corresponding final states, i.e.,

$$\rho_{IO} \quad = \quad \rho^* \cap (I \times F) \; .$$

A final state $s'$ is *reachable* from an initial state $s$ if $(s, s') \in \rho_{IO}$. We write $F_{reach}$ for the set of reachable final states, i.e.

$$F_{reach} \quad = \quad \{s' \in F \mid \exists s \in I : \; (s, s') \in \rho_{IO}\}$$

Given a final state $s' \in F$, we define its *preimage* $P^{-1}(s')$ to be the set of all initial states from which $s'$ is reachable, i.e.,

$$P^{-1}(s') \quad = \quad \{s \mid (s, s') \in \rho_{IO}\} \; .$$

The preimage of an unreachable state is the empty set.

### 3.2 Qualitative information flow: what leaks?

We characterize partial knowledge about the elements of a set $A$ in terms of *partitions* of $A$, i.e., in terms of a family $\{B_1, \ldots, B_r\}$ of pairwise disjoint *blocks* such that $B_1 \uplus \cdots \uplus B_r = A$. A partition of $A$ models that each $a \in A$ is known up to its enclosing block $B_i$ such that $a \in B_i$. We compare partitions using the imprecision order $\sqsubseteq$ defined by

$$\{B_1, \ldots, B_r\} \sqsubseteq \{B'_1, \ldots, B'_{r'}\} \quad = \quad \forall i \in \{1, \ldots, r\} \; \exists j \in \{1, \ldots, r'\} : B_i \subseteq B'_j \; .$$

With the imprecision order, larger elements correspond to less knowledge about the elements of $A$. Let $\sqsubset$ be the irreflexive part of $\sqsubseteq$.

**Knowledge about initial states** We consider a deterministic program $P$ that implements a total function, i.e., for each input state $s$ there is one final state $s'$ such that $(s, s') \in \rho_{IO}$. We assume that the initial state of each computation is secret. Furthermore, we assume an attacker that knows the program, in particular its transition relation, and the final state of each computation. In our model, the attacker does not know any intermediate states of the computation.

The knowledge gained by the attacker about initial states of computations of the program $P$ by observing their final states is given by the partition $\Pi$ that consists of the preimages of reachable final states, i.e.,

$$\Pi \quad = \quad \{P^{-1}(s') \mid s' \in F_{reach}\} \; . \tag{1}$$

There are two extreme cases. The partition $\Pi = \{I\}$ consisting of a single block captures that $P$ reveals no information about its input. In contrast, the partition $\Pi = \{\{s\} \mid s \in I\}$ where each block consists of a single element captures the case that $P$ fully discloses its input. For the remaining, intermediate cases such that $\{\{s\} \mid s \in I\} \sqsubset \Pi \sqsubset \{I\}$, the partition $\Pi$ captures that $P$ leaks partial information about its input.

**Knowledge refinement by interaction** More generally, a program $P$ may receive and produce both secret (*high*) and public (*low*) inputs, where the low inputs may be read or modified by an attacker. If the high input remains fixed over several runs of the program, the attacker can use the low inputs to influence the computation and thereby refine his knowledge about the high input. A restricted version of the above scenario can be reduced to the setting of programs with only high inputs and can be analyzed using the methods presented in this chapter.

Specifically, we consider the case where the attacker runs the program using a fixed finite set of low inputs $l_1, \ldots, l_n$. We model this scenario using a finite set of programs $P_1, \ldots, P_n$ where each $P_i$ corresponds to the program $P$ with the low input set to the value $l_i$. An attacker running the program $P$ with two low inputs $l_i$ and $l_j$ and observing the final states $s_i$ and $s_j$, respectively, can hence narrow down the set of possible secrets to $P_i^{-1}(s_i) \cap P_j^{-1}(s_j)$. More generally, the partition

$$\Pi \quad = \quad \{P_1^{-1}(s_1) \cap \cdots \cap P_n^{-1}(s_n) \mid s_1, \ldots, s_n \in F\} \tag{2}$$

characterizes what an attacker can learn about a fixed secret input after running $P$ with the low inputs $l_1, \ldots, l_n$ and observing the corresponding outputs, see e.g. [38].

Notice that the partition $\Pi$ corresponds to the set of preimages of the function $f\colon I \to F^n$ where each component $f_i$ is defined by the input-output behavior of program $P_i$. This function $f$ can be computed by the independent composition of the programs $P_1, \ldots, P_n$, see e.g. [7]. For example, for the case $n = 2$ we obtain an independent composition by creating a copy $P'$ of $P$ and replacing every program variable $x$ that occurs in $P$ by a fresh variable $x'$. An analysis of the following program with input $h'$ and output $(l, l')$ then corresponds to an analysis of $P$ with respect to two runs with high input $h$ and low inputs $l_1$ and $l_2$, respectively.

$l = l_1;\ l' = l_2;\ h' = h$
$P(h, l)$
$P'(h', l')$
**return** $(l, l')$

This construction easily generalizes to $n > 2$, however at the expense of an exponentially growing state-space. In this way, analyzing a deterministic program with respect to a fixed finite set $l_1, \ldots, l_n$ of low inputs can be reduced to the analysis of a program without low inputs. For simplicity of exposition we will focus on programs without low inputs in the remainder of the chapter.

### 3.3 Quantitative information flow: how much leaks?

In the following, we use information theory to characterize the information that $P$ reveals about its input. This characterization has the advantage of being compact and easy to compare. Moreover, it yields concise interpretations in terms of the effort needed to determine $P$'s input from the revealed information.

We assume that the program's initial state is drawn according to a probability distribution $p$ on $I$ and we suppose that $p$ is known to the attacker. For a random variable $X\colon I \to \mathcal{X}$ with range $\mathcal{X}$, we define $p_X\colon \mathcal{X} \to \mathbb{R}$ as $p_X(x) = \sum_{s\in X^{-1}(x)} p(s)$, which we will also denote by $\Pr(X = x)$. For analyzing the program $P$, there are two random variables of particular interest. The first random variable $U\colon I \to I$ models the random choice of an input in $I$, i.e., $U(s) = s$. The second random variable $V\colon I \to F$ captures the input-output behavior of $P$, i.e., $V(s) = s'$ where $(s, s') \in \rho_{IO}$.

**Shannon entropy** The *(Shannon) entropy* [58] of a random variable $X\colon I \to \mathcal{X}$ is defined as

$$H(X) = -\sum_{x\in\mathcal{X}} p_X(x) \log_2 p_X(x) \,.$$

The entropy is a lower bound for the average number of bits required for representing the results of independent repetitions of the experiment associated with $X$. Thus, in terms of guessing, $H(X)$ is a lower bound for the average number of questions with binary outcome that need to be asked to determine $X$'s value [13]. Given another random variable $Y\colon I \to \mathcal{Y}$, we write $H(X|Y = y)$ for the entropy of $X$ given that the value of $Y$ is $y$. The *conditional entropy* $H(X|Y)$ of $X$ given $Y$ is defined as the expected value of $H(X|Y = y)$ over all $y \in \mathcal{Y}$, namely

$$H(X|Y) = \sum_{y\in\mathcal{Y}} p_Y(y) H(X|Y = y) \,,$$

and it captures the remaining entropy about $X$ when $Y$ is observed.

Consider now a program $P$ and the corresponding random variables $U$ and $V$, as defined above. Then $H(U)$ is the observer's initial uncertainty about the secret input and $H(U|V)$ is the observer's expected uncertainty about the input after running the program. We will use $H(U|V)$ as a measure of information flow because it is associated with operational security guarantees: The expected effort for determining the secret input by exhaustive search is bounded from below by $2^{H(U|V)-2}$, see [47] and [11, 39].

**Min-entropy** The min-entropy of a random variable $X$ captures the probability of correctly determining the value of $X$ in a single guess using an optimal strategy, i.e., by choosing the most likely value. From this probability, it is straightforward to derive bounds on the probability for correctly determining the value of $X$ in an arbitrary number of guesses. Formally, the *min-entropy* $H_\infty$ is defined as

$$H_\infty(X) = -\log_2 \left( \max_{x\in X} p_X(x) \right) \,.$$

The *conditional min-entropy* $H_\infty(X|Y)$ quantifies the expected probability of correctly determining the secret in one guess after having observed the outcome of $Y$ [59] and is defined by

$$H_\infty(X|Y) = -\log_2 \left( \sum_{y \in Y} p_Y(y) \max_{x \in X} p_{X|Y=y}(x) \right) .$$

As before, $H_\infty(U|V)$ quantifies the expected probability of correctly determining the value of $U$ in one guess after observing the output $V$ of the program $P$.

Note that the success probability of a single guess can also be estimated using the conditional Shannon entropy, e.g. using Fano's inequality. However, as pointed out in [59], this estimation is not always accurate. Hence, it is preferable to use min-entropy to compute the success probability of single guesses.

**Leakage vs. security** The information *leaked* by $P$ is the reduction in uncertainty about the input $U$ when the output $V$ is observed. For the case of Shannon entropy, the leakage $L$ is given by

$$L = H(U) - H(U|V) , \tag{3}$$

and it can be defined analogously using alternative measures of uncertainty. Many approaches in the literature focus on computing the leakage rather than the remaining uncertainty. If the initial uncertainty $H(U)$ is known, Equation (3) gives a direct correspondence between the leakage $L$ and the remaining uncertainty $H(U|V)$. In the following, we focus on the remaining uncertainty rather than on the leakage, because the remaining uncertainty enjoys a more direct interpretation in terms of an attacker's difficulty for recovering secrets and, hence, security.

**(Non-)Uniform Input distributions** For the important case where $p$ is the uniform distribution, we have

$$\Pr(V = s') = \frac{\#(P^{-1}(s'))}{\#(I)} , \tag{4}$$

i.e., one can characterize the distribution of $V$ in terms of the sizes of the preimages of $P$. Moreover, one can give formulas for remaining uncertainty about the input of $P$ in terms of the number and the sizes of the partition $\Pi$ induced by the preimages of $P$, see [38, 59]. These formulas provide the interface between the qualitative and the quantitative viewpoints.

**Proposition 1.** *If the input of $P$ is uniformly distributed, we obtain the following expressions for the remaining uncertainty about the input after observing the output of $P$ in terms conditional Shannon and min-entropies, respectively.*

$$H(U|V) = \frac{1}{\#(I)} \sum_{B \in \Pi} \#(B) \log_2 \#(B) \tag{5}$$

$$H_\infty(U|V) = \log_2 \frac{\#(I)}{\#(\Pi)} \tag{6}$$

In scenarios where the input is distributed non-uniformly, Proposition 1 cannot be directly applied. However, a recent result shows that for Shannon entropy, the case of non-uniform input distributions can be reduced to the case of uniform input distributions [1]. The key idea behind the reduction is to consider the non-uniform input distribution $p$ as being generated by a deterministic program $D$ that receives uniformly distributed input.

The key requirement for this generator program is that, for each $s \in I$, the size of $D^{-1}(s)$ be proportional to $p(s)$. The following program satisfies this requirement by construction. Here, we assume that $I = \{s_1, \ldots, s_{\#(I)}\}$ and that the variable $r$ is initialized by values drawn uniformly from $[0, 1]$. For rational distributions, the program can be easily adapted to one that receives uniformly distributed input from an integer range.

$j := 1;\ c := p(s_1)$
**while** $j < \#(I) \wedge r > c$
  $j := j + 1$
  $c := c + p(s_j)$
**return** $s_j$

While the above construction is obviously not practical for large $I$, efficient generator programs often occur in practice. E.g., the Personal Identification Numbers (PINs) used in electronic banking are often not uniformly distributed, but derived from uniform bitstrings using decimalization techniques [21]. Another example are the keys of a public-key cryptosystem, which are typically not uniformly distributed bitstrings. However, they are produced by a key generation algorithm that operates on uniformly distributed input. More generally, a large number of randomized algorithms expect uniformly distributed randomness. For a language-based perspective on distribution generators, see [54].

Given a generator program $D$, the remaining uncertainty about the inputs of a program $P$ can be expressed as the difference in the remaining uncertainty about the inputs of $P\,;D$ and $D$. Modeling the uniformly distributed input by a random variable $U$ and interpreting program $D$ as a random variable, one obtains the following connection [1].

**Proposition 2.**

$$H(D|V) = H(U|V \circ D) - H(U|D)$$

Notice that (5) applies to both terms on the right hand side of Proposition 2 and completes the reduction to the uniform case. In the remainder of the chapter we hence focus on the uniform case. For a more detailed treatment of the non-uniform case, refer to [1]. Furthermore, we will only consider logarithms with base two, and will omit the base from the formulas to simplify notation.

### 3.4 Towards automation

Proposition 1 immediately suggests the following approach for automatically determining the remaining uncertainty about the inputs of a program $P$.

1. For computing $H(U|V)$, first enumerate the elements of $\Pi$ and then determine their sizes.
2. For computing $H_\infty(U|V)$, determine the number of elements in $\Pi$.

In [2] it was shown how the partition $\Pi$ can be obtained by computing relational weakest preconditions, how its blocks can be enumerated using SAT solvers, and how the sizes of the blocks can be determined using model counting [30]. Unfortunately, the exact computation of these properties can be prohibitively expensive (see also [15,60]).

## 4 Bounding information leaks

In this section, we present a method for the automatic derivation of upper and lower bounds on the remaining uncertainty about a program's inputs.

We consider bounds that over- and under-approximate the remaining uncertainty both qualitatively and quantitatively. On the qualitative side, we show how to compute over- and under-approximations of the set of blocks in $\Pi$. Moreover, we show how to compute over- and under-approximations for each block in $\Pi$. On the quantitative side, we show how these over- and under-approximations can be used for computing bounds on the remaining uncertainty in terms of min-entropy and Shannon entropy.

### 4.1 Bounding block count

Computation of min-entropy requires estimation of the number of blocks in the partition $\Pi$, which is equal to the cardinality of the set of reachable final states $F_{reach}$, see (1) and (6). The set $F_{reach}$ can be over-approximated by applying abstract interpretation techniques [24] on the program $P$. Abstract interpretation allows one to incrementally compute an approximation of the set of reachable program states by relying on the approximation of individual program transitions. The set $F_{reach}$ can be under-approximated by symbolic execution and backward propagation along the sequence of program statements traversed during the execution.

**Over-approximation of $F_{reach}$** For the computation of the over-approximation ${F_{reach}}^\sharp \supseteq F_{reach}$ we will use an *abstraction* function $\alpha : 2^S \to 2^S$ that over-approximates a given set of states, i.e., for each $X \subseteq S$ we have $X \subseteq \alpha(X)$. For simplicity of exposition we assume that the abstract values are sets of program states, which leads to the concretization function that is the identity function and hence is omitted. The presented approach can use more sophisticated abstract domains without any modifications.

In theory, the set of reachable final states $F_{reach}$ can be computed by iterating the one-step reachability function $post\colon 2^{S\times S} \times 2^S \to 2^S$ defined below. Note that we put the first parameter in the subscript position to simplify notation.

$$post_\rho(X) \quad = \quad \{s' \mid \exists s \in X : (s, s') \in \rho\}$$

The iteration process applies $post_\rho$ on the set $I$ zero, one, two, ..., many times, takes the union of the results and restricts it to the final states. The resulting set is the intersection

of the final states with the least fixpoint of $post_\rho$ containing $I$ by the Kleene fixpoint theorem. Formally, we have

$$\begin{aligned} F_{reach} &= F \cap (I \cup post_\rho(I) \cup post_\rho(post_\rho(I)) \cup \dots) \\ &= F \cap lfp(post_\rho, I) \ . \end{aligned}$$

Unfortunately, it is not practical to compute the result of iterating $post_\rho$ arbitrarily many times, i.e., the iteration may diverge for programs over unbounded (infinite) sets of states.

Abstract interpretation overcomes the above fundamental problem of computing $F_{reach}$ by resorting to an approximation of $post_\rho$ using the abstraction function $\alpha$. That is, instead of iterating $post_\rho$ we will iterate its composition with $\alpha$, i.e., we will compute the restriction of the abstract least fixpoint to the final states. Let $\bullet$ be a binary *function composition* operator such that $f \bullet g = \lambda x.f(g(x))$. Then

$$\begin{aligned} {F_{reach}}^\sharp &= F \cap (\alpha(I) \cup (\alpha \bullet post_\rho)(\alpha(I)) \cup (\alpha \bullet post_\rho)^2(\alpha(I)) \cup \dots) \\ &= F \cap lfp(\alpha \bullet post_\rho, \alpha(I)) \ . \end{aligned}$$

Instead of computing the exact set $F_{reach}$, we compute its over-approximation, i.e.,

$$F_{reach} \subseteq {F_{reach}}^\sharp \ .$$

By choosing an adequate abstraction function $\alpha$ we can enforce termination of the iteration process after a finite number of steps, as no new states will be discovered. In other words, after some $k \geq 0$ steps we obtain

$$(\alpha \bullet post_\rho)^{k+1}(\alpha(I))) \subseteq \bigcup_{i=0}^{k} (\alpha \bullet post_\rho)^k(\alpha(I)) \ ,$$

while maintaining the desired precision of the over-approximation. Here, we can rely on an extensive body of research in abstract interpretation and efficient implementations of abstract domains, including octagons and polyhedra [3, 25, 35, 50].

**Under-approximation of $F_{reach}$** When computing the under-approximation ${F_{reach}}^\flat \subseteq F_{reach}$ we follow an approach that is different from its over-approximating counterpart, since we cannot rely on abstract interpretation for computing an under-approximation. Despite the established theoretical foundation, abstract interpretation does not yet provide practical under-approximating abstractions.

Instead, we can resort to a practical approach for computing under-approximations by symbolic execution of the program along a selected set of program paths. This approach has been successful for efficient exploration of state spaces (for finding runtime safety violations), e.g., Verisoft, KLEE, DART, and BitScope [12, 14, 28, 29].

Let $\pi_1, \dots, \pi_n \in \mathcal{T}^+$ be a finite set of non-empty program paths, which can be chosen randomly or according to particular program coverage criteria. This set of paths determines a subset of reachable finite states in the following way.

$${F_{reach}}^\flat = \bigcup_{\pi \in \{\pi_1, \dots, \pi_n\}} \{s' \mid \exists s \in S : (s, s') \in \rho_\pi \cap (I \times F)\}$$

```
    procedure MkPath
    input
        s ∈ I - initial state
    begin
1       π := ϵ
2       while s ∉ F do
3           (τ, s′) := choose τ ∈ 𝒯 such that (s, s′) ∈ ρ_τ
4           s := s′
5           π := π · τ
6       done
7       return (π, s)
    end.
```

**Fig. 1.** Function MkPath computes the program path and the final state for a given initial state.

In Figure 1 we describe a possible implementation of a symbolic execution function MkPath that creates a program path for a given initial state. The termination of MkPath follows from the requirement that $P$ implements a total function.

Given the over- and under-approximations ${F_{reach}}^{\sharp}$ and ${F_{reach}}^{\flat}$, we can bound the number of blocks in the partition $\Pi$ as formalized in the following theorem.

**Theorem 1.** *The over- and under-approximations of the set of reachable final states yield over- and under-approximations of the number of blocks in the partition $\Pi$. Formally,*

$$\#({F_{reach}}^{\flat}) \leq \#(\Pi) \leq \#({F_{reach}}^{\sharp}) \ .$$

*Proof.* The theorem statement is a direct consequence of the bijection between $F_{reach}$ and $\Pi$ under the inclusion ${F_{reach}}^{\flat} \subseteq F_{reach} \subseteq {F_{reach}}^{\sharp}$.

### 4.2 Bounding block sizes

Computation of block sizes in $\Pi$ requires identification of the blocks as sets of initial states. Our technique approaches the computation of $\Pi$ through an intermediate step that relies on the input-output relation $\rho_{IO}$. We formulate the computation of the input-output relation as the problem of computing sets of reachable states, which immediately allows one to use tools and techniques of abstract interpretation and symbolic execution as presented above. Then, given $\rho_{IO}$, we compute $\Pi$ following (1).

In order to compute the input-output relation $\rho_{IO}$ we augment the program $P$ such that the set of reachable states keeps track of the initial states. We construct an augmented program $\overline{P} = (\overline{S}, \overline{I}, \overline{F}, \overline{\mathcal{T}})$ from the program $P$ as follows.

- $\overline{S} = S \times S$.
- $\overline{I} = \{(s, s) \mid s \in I\}$.
- $\overline{F} = S \times F$.

– $\overline{\mathcal{T}} = \{\overline{\tau} \mid \tau \in \mathcal{T}\}$, where for each transition $\overline{\tau} \in \overline{\mathcal{T}}$ we construct the transition relation $\rho_{\overline{\tau}}$ such that

$$\rho_{\overline{\tau}} \quad = \quad \{((s'', s), (s'', s')) \mid (s, s') \in \rho_\tau \wedge s'' \in S\} .$$

Similarly to $P$, we define $\overline{\rho} \quad = \quad \bigcup_{\overline{\tau} \in \overline{\mathcal{T}}} \rho_{\overline{\tau}}$.

Our augmentation procedure is inspired by the use of ghost variables for program verification, see e.g. [5]. Note that when constructing $\overline{P}$ we *do not* apply the self-composition approach [7], and hence we avoid the introduction of additional complexity to $P$. In fact, the construction of $\overline{P}$ from $P$ can be implemented as a source-to-source transformation by declaring copies of input variables and adding an assignment from inputs into these copies as the first program statement.

The set of reachable states of the augmented program $\overline{P}$ corresponds to the input-output relation of the program $P$, as stated by the following theorem.

**Theorem 2 (Augmentation).** *The input-output relation of $P$ is equal to the set of reachable final states of its augmented version $\overline{P}$, i.e.,*

$$\rho_{IO} = \overline{F}_{reach} .$$

*Proof.* The augmented program manipulates pairs of states of the original program. We observe that the augmented program does not modify the first component of its initial state, which stays equal to the initial value. Furthermore, the second component follows the transition relation of $P$. Thus, the theorem statement follows directly.

Now we apply abstract interpretation and symbolic execution techniques from Section 4.1 to the augmented program $\overline{P}$. We obtain the over-approximation $\overline{F}_{reach}^{\sharp}$ by abstract least fixpoint computation of $post_{\overline{\rho}}$, where $\alpha$ over-approximates sets of $\overline{S}$-states, and its restriction to the final states of $\overline{P}$, i.e.,

$$\overline{F}_{reach}^{\sharp} \quad = \quad \overline{F} \cap lfp(\alpha \bullet post_{\overline{\rho}}, \ \alpha(\overline{I})) .$$

The computation of the under-approximation $\overline{F}_{reach}^{\flat}$ requires a finite set of paths $\pi_1, \ldots, \pi_n$ through the augmented program, however we could also use a set of paths through $P$ and adjust accordingly. Again we use the paths for performing symbolic execution and applying existential quantification over the initial states to obtain $\overline{F}_{reach}^{\flat}$.

We finally put together over- and under-approximations of preimages of $P$, indexed by the corresponding final states.

**Theorem 3 (Augmented approximation).** *Projection of the over- and under-approximations of reachable final states of the augmented program on the initial component over- and under-approximates respective blocks in the partition $\Pi$ for the program $P$. Formally, for each $s' \in F$ we have*

$$\{s \mid (s, s') \in \overline{F}_{reach}^{\flat}\} \subseteq P^{-1}(s') \subseteq \{s \mid (s, s') \in \overline{F}_{reach}^{\sharp}\} .$$

Thus, given a reachable final state of $P$ we can apply Theorem 3 to compute an over- and under-approximation of the corresponding block in the partition $\Pi$.

### 4.3 Information-theoretic bounds

We now show how, for uniformly distributed input, bounds on the size and the number of the elements of $\Pi$ can be used for deriving bounds on the remaining uncertainty of a program in terms of Shannon entropy and min-entropy.

**Shannon entropy** For uniformly distributed inputs, one can express the probability $\Pr(V = s')$ of the program outputting $s'$ in terms of the size of $P^{-1}(s')$, see (4). We define upper and lower bounds $p^\flat(s')$ and $p^\sharp(s')$ for $\Pr(V = s')$ on the basis of the under- and over-approximation of $P^{-1}(s')$ given in Theorem 3.

Formally, we assume $s' \in F_{reach}$ and define

$$p^\flat(s') \;=\; \max\left\{\frac{\#(\{s \mid (s,s') \in \overline{F}_{reach}{}^\flat\})}{\#(I)}, \frac{1}{\#(I)}\right\}$$

$$p^\sharp(s') \;=\; \frac{\#(\{s \mid (s,s') \in \overline{F}_{reach}{}^\sharp\})}{\#(I)}\,.$$

From Theorem 3 then follows that

$$p^\flat(s') \;\le\; \Pr(V = s') \;\le\; p^\sharp(s') \tag{7}$$

for all $s' \in F_{reach}{}^\flat$. These bounds extend to all $s' \in F_{reach}$ because for $s' \in F_{reach} \setminus F_{reach}{}^\flat$, the value $p^\flat(s') = 1/\#(I)$ is an under-approximation of the probability $p(V = s')$.

The following theorem shows that we can bound $H(U|V)$ in terms of combinations of upper and lower bounds for the preimage sizes.

**Theorem 4.** *If $U$ is uniformly distributed, the remaining uncertainty $H(U|V)$ is bounded as follows*

$$\begin{aligned}\sum_{s' \in F_{reach}{}^\sharp} & p^\sharp(s') \log p^\flat(s') + \log \#(I) \\ &\le H(U|V) \\ &\le \sum_{s' \in F_{reach}{}^\flat} p^\flat(s') \log p^\sharp(s') + \log \#(I)\,.\end{aligned}$$

*Proof.* $P$ implements a total function. As a consequence, $V$ is determined by $U$. We obtain $H(U) = H(UV) = H(U|V) + H(V)$ and conclude $H(U|V) = H(U) - H(V)$. As $U$ is uniformly distributed, we have $H(U) = \log \#(I)$. By definition of Shannon entropy,

$$H(V) = \sum_{s' \in F_{reach}} \Pr(V = s')(-\log \Pr(V = s'))\,. \tag{8}$$

Observe that $-\log$ is nonnegative and decreasing on $(0, 1]$. Together with the bounds from (7), this monotonicity implies that replacing in (8) the occurrences of $-\log \Pr(V =$

$s'$) by $-\log p^\sharp(s')$, replacing the remaining occurrences of $\Pr(V = s')$ by $p^\flat(s')$, and dropping the summands corresponding to elements of $F_{reach} \setminus {F_{reach}}^\flat$ will only decrease the sum, which leads to the upper bound on $H(U|V)$. The lower bound follows along the same line.

**Min-entropy** The following theorem shows that it suffices to over- and under-approximate the size of the range of a program $P$ in order to obtain approximations for the remaining uncertainty about $P$'s input.

**Theorem 5.** *If $U$ is uniformly distributed, the remaining uncertainty $H_\infty(U|V)$ of a program $P$ is bounded as follows*

$$\log \frac{\#(I)}{\#({F_{reach}}^\sharp)} \leq H_\infty(U|V) \leq \log \frac{\#(I)}{\#({F_{reach}}^\flat)} \,.$$

*Proof.* Smith [59] shows that $H_\infty(U|V) = \log(\#(I)/\#(I\Pi))$. The assertion then follows from Theorem 1 and the monotonicity of the logarithm.

## 5 Randomized quantification

In Section 4 we showed how to obtain bounds on the remaining uncertainty about a program's input by computing over- and under-approximations of the set of reachable states and the corresponding preimages.

While the presented approach for computing min-entropy bounds requires determining the size of (an approximation of) the set of reachable states (see Theorem 5), the approach for computing Shannon-entropy bounds requires *enumerating* this set (see Theorem 4). This enumeration constitutes the bottleneck of our approach and inhibits scalability to large systems.

In this section, we show that the enumeration of the set of reachable states can be replaced by sampling the preimages with probabilities according to their relative sizes. To this end, we run the program on a randomly chosen input and approximate the preimage of the corresponding output. We combine the sizes of the approximations of the preimage and obtain upper and lower bounds for the remaining uncertainty. Moreover, we give confidence levels for these bounds. These confidence levels are already close to 1 for a number of samples of as small as $O((\log \#(I))^2)$.

On a technical level, our result makes use of the fact that the random variable given by the logarithm of the size of randomly chosen preimages has small variance [9]. The Chebyshev inequality implies that the estimations obtained from sampling this random variable are likely to be accurate.

### 5.1 RANT: A randomized algorithm for quantitative information flow analysis

Given a program $P$, our goal is to compute bounds $H^\sharp$ and $H^\flat$ with quality guarantees for the remaining uncertainty $H(U|V)$ about the program's input when the program's output is known. More precisely, given a confidence level $p \in [0, 1)$ and a desired degree of precision $\delta > 0$, we require that

$$H^\flat - \delta \;\leq\; H(U|V) \;\leq\; H^\sharp + \delta$$

with a probability of at least $p$.

**Algorithm** Our procedure RANT computes such bounds in an incremental fashion. After an initialization phase in lines 1 and 2, RANT randomly picks an initial state $s \in I$, see line 4 in Figure 2. Then, RANT runs the program $P$ on input $s$ to determine the final state $s'$ and the corresponding execution path $\pi$, see line 5. We use the technique described in Section 4.2 for determining an over-approximation of the preimage of $s'$ in line 6. Note that $\overline{F_{reach}}^\sharp$ only needs to be computed once and can be re-used for all iterations of the while loop. We use the techniques described in 4.2 for determining an under-approximation of the preimage of $s'$ in line 7. The variables $H^\sharp$ and $H^\flat$ aggregate the logarithms of the preimage sizes. After

$$n = \frac{(\log \#(I))^2}{(1-p)\delta^2}$$

many iterations of while loop, $H^\sharp$ and $H^\flat$ are normalized by $n$ and returned as upper and lower bounds for $H(U|V)$, respectively.

**Counting preimage sizes** The computations in lines 6 and 7 of RANT require an algorithm that, given a set $A$, returns the number of elements $\#(A)$ in $A$. If $A$ is represented as a formula $\phi$, this number corresponds to the number of models for $\phi$. For example, if $A$ is represented in linear arithmetic, this task can be performed efficiently using Barvinok's algorithm [8]. The Lattice Point Enumeration Tool (LATTE) [43] provides an implementation of this algorithm.

**Correctness** The following theorem states the correctness of the algorithm RANT.

**Theorem 6.** *Let $P$ be a program, $\delta > 0$, and $p \in [0, 1)$. Let $U$ be uniformly distributed. If* RANT$(P, \delta, p)$ *outputs $(H^\sharp, H^\flat)$, then*

$$H^\flat - \delta \leq H(U|V) \leq H^\sharp + \delta$$

*with a probability of more than $p$.*

We need the following lemma for the proof of Theorem 6. The proof of the lemma is based on a result from [9].

```
    function RANT
    input
        P : program
        δ > 0 : desired precision
        p ∈ [0, 1) : desired confidence level
    vars
        π: program path
    output
        H♯, H♭ : upper and lower bounds for H(U|V)
    begin
1       n := 0
2       H♭ := H♯ := 0
3       while n < log(#(I))^2/((1 − p)δ^2) do
4           s := choose from I randomly
5           (π, s′) := MKPATH(s)
6           H♯ := H♯ + log #({s″ | (s″, s′) ∈ F̄_reach♯})
7           H♭ := H♭ + log #({s″ | (s″, s′) ∈ ρ_π})
8           n := n + 1
9       done
10      return (H♯/n, H♭/n)
    end.
```

**Fig. 2.** Randomized procedure RANT for computing an approximation of the remaining uncertainty about the input of a program. The relation $\overline{F}_{reach}^{\sharp}$ is computed once and re-used for all iterations of the while loop.

**Lemma 1.** *Let $X$ be a random variable with range of size $m$ and let $\delta > 0$. Let $x_1, \ldots, x_n$ be the outcomes of $n$ independent repetitions of the experiment associated with $X$. Then*

$$-\frac{1}{n}\sum_{i=1}^{n} \log \Pr(X = x_i) - \delta \quad \leq \quad H(X) \quad \leq \quad -\frac{1}{n}\sum_{i=1}^{n} \log \Pr(X = x_i) + \delta$$

*with a probability of more than $1 - \frac{(\log m)^2}{n\delta^2}$.*

*Proof.* We define the random variable $Y$ by $Y(x) = -\log \Pr(X = x)$. Then we have $E(Y) = H(X)$ for the expected value $E(Y)$ of $Y$. By additivity of the expectation, we also have $E(Z) = H(X)$ for the sum $Z = \frac{1}{n}\sum_{i=1}^{n} Y_i$ of $n$ independent instances $Y_i$ of $Y$. In [9] it is shown that the variance $Var[Z]$ of $Z$ is bounded from above by

$$Var[Z] \leq \frac{(\log m)^2}{n} \ .$$

The Chebyshev inequality

$$\Pr(|Q - E(Q)| \geq \delta) \leq \frac{Var[Q]}{\delta^2} \tag{9}$$

gives upper bounds for the probability that the value of a random variable $Q$ deviates from its expected value $E(Q)$ by at least $\delta$. We apply (9) to $Z$ with the expectation and variance bounds derived above and obtain

$$\Pr\left(|Z - H(X)| \geq \delta\right) \leq \frac{(\log m)^2}{n\delta^2} \ .$$

Considering the complementary event and inserting the definition of $Z$ we obtain

$$\Pr\left(\left|-\frac{1}{n}\sum_{i=1}^{n}\log\Pr(X = x_i) - H(X)\right| \leq \delta\right) \geq 1 - \frac{(\log m)^2}{n\delta^2} \ ,$$

from which the assertion follows immediately.

We are now ready to give the proof of Theorem 6.

*Proof (Proof of Theorem 6).* Let $s'_i$ be the final state of $P$ that is computed in line 5 of the $i$th loop iteration of Rant, for $i \in \{1, \ldots, n\}$. For uniformly distributed $U$, we have $H(U) = \log \#(I)$ and $\Pr(V = s'_i) = \#(P^{-1}(s'_i))/\#(I)$. As $V$ is determined by $U$, $H(U|V) = H(U) - H(V)$. Replacing $H(V)$ by the approximation given by Lemma 1 we obtain

$$\begin{aligned} & H(U) + \frac{1}{n}\sum_{i=1}^{n}\log\Pr(V = s'_i) \\ = \ & \log \#(I) + \frac{1}{n}\sum_{i=1}^{n}\log\frac{\#(P^{-1}(s'_i))}{\#(I)} \\ = \ & \frac{1}{n}\sum_{i=1}^{n}\log \#(P^{-1}(s'_i)) \end{aligned}$$

Lemma 1 now implies that

$$\frac{1}{n}\sum_{i=1}^{n}\log \#(P^{-1}(s'_i)) - \delta \leq H(U|V) \leq \frac{1}{n}\sum_{i=1}^{n}\log \#(P^{-1}(s'_i)) + \delta$$

with a probability of more than

$$1 - \frac{(\log \#(F_{reach}))^2}{n\delta^2} \ ,$$

which is larger than

$$\min\left\{1 - \frac{(\log \#(I))^2}{n\delta^2}, 1 - \frac{(\log \#(F))^2}{n\delta^2}\right\} \ .$$

This statement remains valid if the preimage sizes on the left and right hand sides are replaced by under- and over-approximations, respectively. Finally, observe that the loop guard ensures that the returned bounds hold with probability of more than $p$, which concludes this proof.

Observe that the proof of Theorem 6 implies that in scenarios where $\#(F_{reach})$ is known and smaller than $I$, a smaller number of samples is already sufficient for obtaining a desired confidence level. For example, when analyzing a program with a single Boolean output, the bounds delivered by Rant are valid with a probability of more than

$$1 - \frac{1}{n\delta^2} \ .$$

In general, however, the computation of $\#(F_{reach})$ requires an additional analysis step. For simplicity, our presentation hence focusses on the weaker bounds in terms of $\#(I)$.

Note that, if the sizes of the preimages of $P$ can be determined precisely, we have $H^\sharp = H^\flat$. Then Theorem 6 gives tight bounds for the value of $H(U|V)$. In this way, Rant can be used to replace the algorithm Quant from [2].

### 5.2 Complexity of approximating entropy

The algorithm Rant relies on the approximation of the sizes of the preimages for given sampled outputs of the program. It is natural to ask whether bounds on the entropy can be estimated by sampling alone, i.e. without resorting to structural properties of the program.

A result by Batu et al. [9] suggests that this cannot be done. They show that there is no algorithm that, by sampling alone, can approximate the entropy of every random variable $X$ with a range of size $m$ within given multiplicative bounds. They also show that, for random variables with high entropy (more precisely $H(X) > \log(m/\gamma^2)$, for some $\gamma > 0$) any algorithm that delivers approximations $H$ with

$$\frac{1}{\gamma} H \leq H(X) \leq \gamma H$$

is required to take at least $\Omega(m^{1/\gamma^2})$ samples.

However, if in addition to the samples, the algorithm has access to an oracle that reveals the probability $\Pr(X = x)$ with which each sample $x$ occurs, the entropy can be estimated within multiplicative bounds using a number of samples that is proportional to $(\log m)/h$, where $h \leq H(X)$.

Lemma 1 extends this result to obtain additive bounds for $H(X)$. These bounds hold without any side-condition on $H(X)$, which allows us to determine the number of samples that are required for obtaining confidence levels that hold *for all* $X$ with $\#(ran(X)) \leq m$. The algorithm Rant builds on this result and employs the techniques presented in Section 4 for approximating the probabilities of events on demand, allowing us to derive bounds on the information leakage of real programs.

## 6 Related work

For an overview of language-based approaches to information-flow security, see the survey by Sabelfeld and Myers [56].

Denning is the first to quantify information flow in terms of the reduction in uncertainty about a program variable [26]. Millen [49] and Gray [31] use information theory to derive bounds on the transmission of information between processes in multi-user systems. Lowe [44] shows that the channel capacity of a program can be over-approximated by the number of possible behaviors.

The use of equivalence relations to characterize qualitative information flow was proposed by Cohen [22] and has since then become standard, see e.g. [6, 7, 27, 57, 61].

Clark, Hunt, and Malacaria [18] connect equivalence relations to quantitative information flow, and propose the first type system for statically deriving quantitative bounds on the information that a program leaks [19]. The analysis assumes as input upper and lower bounds on the entropy of the input variables and performs compositional reasoning on basis of those bounds. For loops with high guards, the analysis always reports the complete leakage of the guard.

Malacaria [45] characterizes the leakage of loops in terms of the loop's output and the number of iterations. In our model, the information that is revealed by the number of loop iterations can be captured by augmenting loops with observable counters, as shown in Section 2. In this way, our method can be used to automatically determine this information.

Mu and Clark [52] propose an automatic quantitative analysis based on probabilistic semantics. Their analysis delivers precise results, but is limited to programs with small state-spaces due to the explicit representation of distributions. An abstraction technique [51] addresses this problem by partitioning the (totally ordered) domain into intervals, on which a piecewise uniform distribution is assumed. Our approach is also based on partitioning the input domain, however, without the restriction to intervals. Furthermore, we avoid the enumeration of all blocks by the choice of a random subset.

Köpf and Basin [38] characterize the leaked information in terms of the number of program executions, where an attacker can adaptively provide inputs. The algorithms for computing this information for a concrete system rely on an enumeration of the entire input space and are difficult to scale to larger systems.

Backes, Köpf, and Rybalchenko [2] show how to synthesize equivalence relations that represent the information that a program leaks, and how to quantify them by determining the sizes of equivalence classes. Our approach shows that the exact computation of the sizes of the equivalence classes can be replaced by over- and under-approximations, and that the enumeration of equivalence classes can be replaced by sampling. This enables our approach to scale to larger programs, e.g., those with unbounded loops.

McCamant and Ernst [48] propose a dynamic taint analysis approach for quantifying information flow. Their method provides over-approximations of the leaked information along a particular path, but does not yield guarantees for all program paths, which is important for security analysis. For programs for which preimages can be approximated, our method can be used to derive upper and lower bounds for the leakage of *all* paths without the need for complete exploration.

Newsome, McCamant, and Song [53] use the feasible outputs along single program paths as lower bounds for channel capacity, and they apply a number of heuristics to approximate upper bounds on the number of reachable states of a program. They assume

a fixed upper bound on the number of loop unrollings. In contrast, our technique does not require an upper bound on the number of loop iterations, and it comes with formal quality guarantees for the estimated quantities.

Heusser and Malacaria [34] use model-checking to verify assertions about the number of feasible outputs of a program. A valid assertion translates to an upper bound on the channel capacity of the program. In contrast, we apply model counting techniques to immediately obtain upper and lower bounds on the number of feasible outputs.

Chatzikokolakis, Chothia, and Guha [16] use sampling to build up a statistical model of a probabilistic program, which is treated as a black box. Based on this model, they compute the maximum leakage w.r.t. all possible input distributions. In contrast, our approach is based on the actual semantics of deterministic programs, as given by the source code, and we use a randomized algorithm to compute the adversary's remaining uncertainty about the input.

A number of alternative information measures have been considered in the literature. Di Pierro, Hankin, and Wiklicky [55] measure information flow in terms of the number of statistical tests an attacker has to perform in order to distinguish two computations based on different secrets. Clarkson, Myers, and Schneider [20] propose to measure information flow in terms of the accuracy of an attacker's belief about a secret, which may also be wrong. Reasoning about beliefs is out of the scope of entropy-based measures, such as the ones used in this chapter. One advantage of entropy-based measures is the direct connection to equivalence relations, which makes them amenable to automated reasoning techniques. Finally, we mention that information-theoretic notions of leakage are also used for analyzing anonymity protocols, see e.g. [17].

Our approach relies on abstract interpretation and symbolic execution techniques for the approximation of the set of program outputs. There exist efficient implementations of abstract interpreters with abstraction functions covering a wide spectrum of efficiency/precision trade-offs, see e.g. [4, 10, 33, 42]. In particular, for bounding the block count one could apply tools for discovery of all valid invariants captured by numeric abstract domains, e.g., octagons or polyhedra [3, 50]. Similarly, we can rely on existing dynamic engines for symbolic execution that can deal with various logical representation of program states, including arithmetic theories combined with uninterpreted function symbols and propositional logic, e.g., VERISOFT, KLEE, DART, and BITSCOPE [12, 14, 28, 29].

## 7 Conclusions

The exact computation of the information-theoretic properties of programs can be prohibitively hard. In this chapter, we presented algorithms based on approximation and randomization that allow for a tractable yet sufficiently precise approximation of these properties. As ongoing work, we are putting these algorithms to work for the automatic analysis of microarchitectural side-channels [40].

## References


1. M. Backes, M. Berg, and B. Köpf. Non-Uniform Distributions in Quantitative Information-Flow. In *Proc. 6th ACM Conference on Information, Computer and Communications Security (ASIACCS '11)*, pages 367–374. ACM, 2011.
2. M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. IEEE Symp. on Security and Privacy (S&P '09)*, pages 141–153. IEEE, 2009.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2), 2008.
4. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '01)*, pages 203–213. ACM, 2001.
5. T. Ball, T. D. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2), 2005.
6. A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy (S&P '08)*, pages 339–353. IEEE, 2008.
7. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '04)*, pages 100–114. IEEE, 2004.
8. A. Barvinok. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed. *Mathematics of Operations Research*, 19:189–202, 1994.
9. T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld. The complexity of approximating entropy. In *Proc. ACM Symp. on Theory of Computing (STOC '02)*, pages 678–687. ACM, 2002.
10. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM, 2003.
11. M. Boreale. Quantifying information leakage in process calculi. *Information and Computation*, 207(6):699–725, 2009.
12. D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and D. Song. BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, 2007.
13. C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.
14. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224. USENIX, 2008.
15. P. Cerny, K. Chatterjee, and T. Henzinger. The complexity of quantitative information flow problems. In *Proc. IEEE Computer Security Foundations Symposium (CSF '11), to appear*. IEEE, 2011.
16. K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical Measurement of Information Leakage. In *Proc. 16th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, LNCS 6015, pages 390–404. Springer, 2010.

17. K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4):378 – 401, 2008.
18. D. Clark, S. Hunt, and P. Malacaria. Quantitative Information Flow, Relations and Polymorphic Types. *J. Log. Comput.*, 18(2):181–199, 2005.
19. D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.
20. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '05)*, pages 31–45. IEEE, 2005.
21. J. Clulow. The Design and Analysis of Cryptographic Application Programming Interfaces for Security Devices. Master's thesis, University of Natal, SA, 2003.
22. E. Cohen. Information Transmission in Sequential Programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
23. B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *Proc. Intl. Conf. on Formal Methods in Computer-Aided Design (FMCAD '09)*, pages 205–212. IEEE, 2009.
24. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, 1977.
25. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. ACM Symp. on Principles of Programming Languages (POPL '78)*, pages 84–96. ACM, 1978.
26. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
27. R. Giacobazzi and I. Mastroeni. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In *Proc. ACM Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM, 2004.
28. P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.
29. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '05)*, pages 213–223. ACM, 2005.
30. C. Gomez, A. Sabharwal, and B. Selman. Chapter 20: Model counting. In *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
31. J. W. Gray. Toward a Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 1(3-4):255–294, 1992.
32. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '09)*, pages 375–385. ACM, 2009.
33. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL '04)*, pages 232–244. ACM, 2004.
34. J. Heusser and P. Malacaria. Quantifying information leaks in software. In *26th Annual Computer Security Applications Conference, (ACSAC '10)*, pages 261–269. ACM, 2010.
35. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. Intl. Conf. on Computer Aided Verification (CAV '09)*, LNCS 5643, pages 661–667. Springer, 2009.

36. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, 2009.
37. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. Annual Intl. Cryptology Conference (CRYPTO '96)*, LNCS 1109, pages 104–113. Springer, 1996.
38. B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. ACM Conf. on Computer and Communications Security (CCS '07)*, pages 286–296. ACM, 2007.
39. B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *Proc. IEEE Computer Security Foundations Symposium (CSF '09)*, pages 324–335. IEEE, 2009.
40. B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-Channels. In *Proc. 24th International Conference on Computer Aided Verification (CAV '12)*, pages 564–580. Springer, 2012.
41. B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 3–14. IEEE, 2010.
42. G. Lalire, M. Argoud, and B. Jeannet. The interproc analyzer. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.
43. J. A. D. Loera, D. Haws, R. Hemmecke, P. Huggins, J. Tauzer, and R. Yoshida. LattE. http://www.math.ucdavis.edu/~latte/. [Online; accessed 08-Nov-2008].
44. G. Lowe. Quantifying Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 18–31. IEEE, 2002.
45. P. Malacaria. Risk assessment of security threats for looping constructs. *Journal of Computer Security*, 18(2):191–228, 2010.
46. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.
47. J. L. Massey. Guessing and Entropy. In *Proc. IEEE Intl. Symp. on Information Theory (ISIT '94)*, page 204. IEEE Computer Society, 1994.
48. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '08)*, pages 193–205. ACM, 2008.
49. J. K. Millen. Covert Channel Capacity. In *Proc. IEEE Symp. on Security and Privacy (S&P '87)*, pages 60–66. IEEE, 1987.
50. A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
51. C. Mu and D. Clark. An Interval-based Abstraction for Quantifying Information Flow. *ENTCS*, 253(3):119–141, 2009.
52. C. Mu and D. Clark. Quantitative Analysis of Secure Information Flow via Probabilistic Semantics. In *Proc. 4th International Conference on Availability, Reliability and Security (ARES '09)*, pages 49–57. IEEE Computer Society, 2009.
53. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85. ACM, 2009.
54. S. Park, F. Pfenning, and S. Thrun. A Probabilistic Language based upon Sampling Functions. In *Proc. ACM Symposium on Principles of Programming Languages (POPL '05)*, 2005.

55. A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate Non-Interference. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 3–17. IEEE, 2002.
56. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE J. Selected Areas in Communication*, 21(1):5–19, 2003.
57. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. Intl. Symp. on Software Security (ISSS '03)*, LNCS 3233, pages 174–191. Springer, 2004.
58. C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.
59. G. Smith. On the foundations of quantitative information flow. In *Proc. Intl. Conf. of Foundations of Software Science and Computation Structures (FoSSaCS '09)*, LNCS 5504, pages 288–302. Springer, 2009.
60. H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *Proc. IEEE Computer Security Foundations Symposium (CSF '10)*, pages 15–27. IEEE, 2010.
61. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 15–23. IEEE, 2001.