Approximation and Randomization for Quantitative Information-Flow Analysis

Boris Köpf
MPI-SWS
bkoepf@mpi-sws.org

Andrey Rybalchenko
TUM
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. Existing automated techniques for QIF fall short of providing full coverage of all program executions, especially in the presence of unbounded loops and data structures, which are notoriously difficult to analyze automatically. In this paper we propose a blend of approximation and randomization techniques to bear on the challenge of sufficiently precise, yet efficient computation of quantitative information flow properties. Our approach relies on a sampling method to enumerate large or unbounded secret spaces, and applies both static and dynamic program analysis techniques to deliver necessary over- and under-approximations of information-theoretic characteristics.**

I. Introduction

Quantitative information-flow analysis (QIF) is an emerging technique for establishing information-theoretic confidentiality properties. QIF has the potential to become an important practical tool for the security analysis of programs. QIF enjoys several key advantages that make it a compelling alternative technique to the established techniques for information flow analysis, e.g., non-interference analysis and qualitative declassification. For example, quantitative declassification policies can be easier to specify and interpret than qualitative policies, which are often difficult to get right for complex programs. Furthermore, the insights that QIF provides into a program's confidentiality properties go beyond the "yes or don't know" 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. Finally, because information theory forms the foundation of QIF, the quantities that QIF delivers can be directly associated with operational security guarantees. Examples of such guarantees are lower bounds for the expected effort of uncovering secrets by exhaustive search, or upper bounds on the probability of correctly guessing the secret in one attempt.

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 [36]. Existing automation methods focus on either dynamic or static techniques. Dynamic approaches deliver over-approximations of the information that is leaked in individual program executions [38], and can be efficiently computed by relying on program instrumentation techniques. Symbolic execution along the considered program runs can broaden the scope of analysis results, albeit at the cost of expensive solution counts for the resulting SAT formulas [41], which can be accelerated by heuristics. It is a major challenge to extrapolate analysis results to the remaining, unexplored executions. Static approaches target the computation of bounds that hold for the entire set of possible program executions. These approaches apply type systems- and model checking-based techniques to symbolically explore all program executions [1], [15]. In the presence of unbounded looping constructs these techniques resort to overly coarse abstractions, which may result in unsatisfactory precision.

Technically, successful automation of QIF must determine tight, yet efficiently computable bounds on the information-theoretic characteristics of the program. These characteristics can be expressed by a partition of the secret program inputs. In the partition, each block is defined by the preimage of some program output (if we assume deterministic programs). The computation of some information-theoretic characteristics, e.g. Shannon entropy, from this partition requires enumeration of all blocks and estimation of their respective sizes, whereas other measures, e.g. min-entropy, only depends 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 this paper we propose a blend of *approximation* and *randomization* techniques to tackle the challenge of automating QIF. The presented approach avoids the trap of block enumeration by a sampling method that uses the program itself to randomly choose blocks according to their relative sizes. Each sample amounts to a program execution and indexes a block by the corresponding program output. Symbolic execution delivers an under-approximation for each block, which is obtained by a symbolic backward propagation along the sequence of program statements tra-

versed during a sample run. Abstract interpretation [19] in combination with a program transformation that makes the program input explicitly available in the set of reachable program states by memorizing it in a auxiliary variable delivers an over-approximation of blocks indexed by program outputs. Through these indexes we put together under- and over-approximations for each sampled block and 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 our technique is that it ensures fast convergence of the sampling process and provides formal guarantees for the quality of the obtained bounds. Our proof builds upon a result by Batu et al. [8] 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. [8] 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.

Our technical contribution is a method for automatically deriving upper and lower bounds for the information leaked by a program that provides confidence guarantees on the derived bounds. Conceptually, our contribution is to identify a unique interplay between static, dynamic, and randomized analysis techniques and putting it to work for quantitative information flow analysis.

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

II. Illustration

We illustrate our method on two example programs whose quantitative information-flow analysis is currently out of reach for the existing automatic approaches. These examples 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 [18].

A. Dealing with loops

As a first example we consider the electronic purse program from [1] as shown below.

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

The program receives as input the balance `h` of a bank account and debits a fixed amount of 5 from this account until the balance is insufficient for this transaction, i.e., until $\texttt{h} < 5$.

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 [37], which corresponds to a formal, quantitative confidentiality guarantee.

One way for precisely determining the remaining uncertainty 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 [1]. Both enumeration problems severely limit the size of the systems that can be analyzed using this precise approach.

For the purse program, the required partition is given 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 [1] requires 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 paper, we show that the precise computation of each preimage can be replaced by under- and over-approximations. 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 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. The set of reachable final states $\overline{F}_{reach}$ of the augmented program 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 ρ_{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 [19]. 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. [18], [26]. We apply the bound generator BoundGen [18] and obtain

$$\overline{F}_{reach}{}^{\sharp} \;\equiv\; -5 + 5\,\mathtt{l} \le \mathtt{\_h} \le -1 + 5\,\mathtt{l}\ ,$$

The predicate $\overline{F}_{reach}{}^{\sharp}$ represents an over-approximation of the input-output relation ρ_{IO}. 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 [34] 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 and DART [13], [24], and at the binary level, e.g., BitScope [10].

For example, for an input of `h = 37`, this relation is determined to be

$$35 \le \mathtt{h} \le 39 \wedge \mathtt{l} = 8$$

Hence, the preimage size of `l = 8` is at least 5.

Randomization: A direct approximation of the leak in terms of Shannon entropy for uniformly distributed inputs 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 [8], 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

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

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 ith 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 $I = \{0, \ldots, 2^{32}-1\}$, we obtain bounds of

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

that hold with a probability of more than 0.99 when considering 10^7 samples.

B. Dealing with data structures

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 form.

```
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 [29].

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.

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 II-A, 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 [45] 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 [18], we obtain the following over-approximation ${F_{reach}}^\sharp$ of F_{reach},

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

which shows that $\#({F_{reach}}^\sharp) \leq \mathtt{len}+1$. We use this bound to infer that, after observing `time`, the remaining uncertainty about the exponent `k` is still larger than $\log_2 \frac{2^{\mathtt{len}}}{\mathtt{len}+1} = \mathtt{len} - \log_2(\mathtt{len}+1)$, given that `k` is drawn uniformly from $\{0,\ldots,2^{\mathtt{len}}-1\}$. An alternative interpretation is that the uncertainty about the exponent `k` is reduced by at most $\log_2(\mathtt{len}+1)$ bits.

III. Preliminaries

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

A. Programs and computations

A program $P = (S, I, F, \mathcal{T})$ consists of the following components.

- S - a set of *states*.
- $I \subseteq S$ - a set of *initial* states.
- $F \subseteq S$ - a set of *final* 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., $\rho_\tau \subseteq S \times S$.

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 $\tau \in \mathcal{T}$. A *path* is a sequence of transitions. We write ϵ 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 $X \circ Y \quad \equiv \quad \{(x,y) \mid \exists z : (x,z) \in X \wedge (z,y) \in Y\}$. 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 π is *feasible* if its path relation is not empty, i.e., $\rho_\pi \neq \emptyset$.

Let ρ be the *program transition relation* defined as follows.

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

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

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

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.

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.,

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

The preimage of an unreachable state is the empty set.

B. 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

$$ \begin{aligned} & B_1 \uplus \cdots \uplus B_r \sqsubseteq B'_1 \uplus \cdots \uplus B'_{r'} \\ & \quad \equiv \quad \forall i \in \{1,\ldots,r\} \ \exists j \in \{1,\ldots,r'\} : B_i \subseteq B'_j \ . \end{aligned} $$

Let $\sqsubset$ be the irreflexive part of $\sqsubseteq$.

We consider a 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 Π that consists of the preimages of reachable final states, i.e.,

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

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 Π captures that P leaks partial information about its input.

C. 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, e.g., by exhaustive search.

We assume a probability distribution p on I and we suppose that it is known to the attacker. For analyzing the program P, we define two random variables. 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* [44] $H(U)$ is a lower bound for the average number of bits required for representing the results of independent repetitions of the experiment associated with U. Thus, in terms of guessing, the entropy $H(U)$ is a lower bound for the average number of questions with binary outcome that need to be asked to determine U's value [12]. Given the random variable V, we write $H(U|V = s')$ for the entropy of U given the program output s'. The *conditional entropy* $H(U|V)$ of U given V is the expected value of $H(U|V = s')$ over all $s' \in F$. We can give lower bounds for the expected effort for determining a secret by exhaustive search in terms of entropy and conditional entropy, see [37] and [31], respectively.

Min-entropy: The *min-entropy* $H_\infty(U)$ captures the probability of correctly determining the value of U in a single guess. From this probability, it is straightforward to derive bounds on the probability for correctly determining the value of U in an arbitrary number of guesses. The *conditional min-entropy* (see, e.g. [45] for a definition) $H_\infty(U|V)$ captures the expected probability of correctly determining the value of U in one guess after having observed the output of V.

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 leaked information L is given by

$$L = H(U) - H(U|V) , \quad (2)$$

and it can be defined analogously using alternative measures of uncertainty. Equation (2) gives a direct correspondence between the leaked information L and the remaining uncertainty $H(U|V)$, if the initial uncertainty $H(U)$ is known. 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.

Uniform distributions: For the important case where p is the uniform distribution, we have

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

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 Π induced by the preimages of P, see [30], [45]. 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.*

$$H(U|V) \equiv \frac{1}{\#(I)} \sum_{B \in \Pi} \#(B) \log_2 \#(B) \quad (3)$$

$$H_\infty(U|V) \equiv \log_2 \frac{\#(I)}{\#(\Pi)} \quad (4)$$

In the remainder of the paper we will only consider logarithms with base two, and will omit the base from the formulas to simplify notation.

D. 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 Π and then determine their sizes.
2) For computing $H_\infty(U|V)$, determine the number of elements in Π.

There exist techniques for the automatic computation of Π and the enumeration of its blocks, see [1]. Unfortunately, the exact computation of Π can be prohibitively expensive.

IV. 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 Π. Moreover, we show how to compute over- and under-approximations for each block in Π. 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.

A. Bounding block count

Computation of min-entropy requires estimation of the number of blocks in the partition Π, which is equal to the cardinality of the set of reachable final states F_{reach}, see (1) and (4). The set F_{reach} can be over-approximated by applying abstract interpretation techniques [19] 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.

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

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 α. That is, instead of iterating $post_\rho$ we will iterate its composition with α, 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

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

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 α we can enforce termination of the iteration process after 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 [2], [20], [28], [40].

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 [10], [13], [23], [24].

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.

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

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

```
    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.
```

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

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 Π 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* Π*. 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 Π under the inclusion ${F_{reach}}^{\flat} \subseteq F_{reach} \subseteq {F_{reach}}^{\sharp}$. ■

B. Bounding block sizes

Computation of block sizes in Π requires identification of the blocks as sets of initial states. Our technique approaches the computation of Π through an intermediate step that relies on the input-output relation ρ_{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 ρ_{IO}, we compute Π following (1).

In order to compute the input-output relation ρ_{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

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

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

Our augmentation procedure is inspired by the use of ghost variables for program verification, see e.g. [4]. Note that when constructing $\overline{P}$ we *do not* apply the self-composition approach [6], 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 IV-A 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 α over-approximates sets of $\overline{S}$-states, and its restriction to the final states of $\overline{P}$, i.e.,

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

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* Π *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 Π.

C. Information-theoretic bounds

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

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 4. *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 [45] shows that $H_\infty(U|V) = \log(\#(I)/\#(\Pi))$. The assertion then follows from Theorem 1 and the monotonicity of the logarithm. ∎

Shannon entropy: We define upper and lower bounds $p^\flat(s')$ and $p^\sharp(s')$ for $p(V = s')$ on the basis of the under- and over-approximation of $P^{-1}(s')$ given in Theorem 3. Formally, we define

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

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

From Theorem 3 then follows that

$$p^\flat(s') \ \leq \ p(V = s') \ \leq \ p^\sharp(s') \tag{5}$$

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

$$\sum_{s' \in {F_{reach}}^\sharp} p^\sharp(s') \log p^\flat(s') + \log \#(I)$$
$$\leq H(U|V)$$
$$\leq \sum_{s' \in {F_{reach}}^\flat} p^\flat(s') \log p^\sharp(s') + \log \#(I) \ .$$

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

$$H(V) = \sum_{s' \in F_{reach}} P(V = s')(-\log P(V = s')) \ . \tag{6}$$

Observe that $-\log$ is nonnegative and decreasing on $(0, 1]$. Together with the bounds from (5), this monotonicity implies that replacing in (6) the occurrences of $-\log P(V = s')$ by $-\log p^\sharp(s')$, replacing the remaining occurrences of $P(V = s')$ 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. ∎

V. Randomized quantification

In Section IV 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 4), the approach for computing Shannon-entropy bounds requires *enumerating* this set (see Theorem 5). This enumeration constitutes the bottleneck of our approch 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 $O((\log \#(I))^2)$.

A. 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, it randomly picks an initial state $s \in I$, see line 4 in Figure 2. It runs the program P on input s to determine the final state s' and the corresponding execution path π, see line 5. It uses the technique described in Section IV-B for determining an over-approximation of the preimage of s', and it computes the size of this set, see line 6. It uses the techniques described in IV-B for determining an under-approximation of the preimage of s' and it computes the size

```
    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)δ²) do
4           s := choose from I randomly
5           (π, s′) := MKPATH(s)
6           H♯ := H♯ + log #({s″ | (s″, s′) ∈ F_reach♯})
7           H♭ := H♭ + log #({s″ | (s″, s′) ∈ ρ_π})
8           n := n + 1
9       done
10      return (H♯/n, H♭/n)
    end.
```

Figure 2. Randomized procedure RANT for computing an approximation of the remaining uncertainty about the input of a program.

of this set, see line 7. The variables $H^\sharp$ and $H^\flat$ aggregate the logarithms of these sizes. After

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

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

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 ϕ, this number corresponds to the number of models for ϕ. For example, if A is represented in linear arithmetic, this task can be performed efficiently using Barvinok's algorithm [7]. The Lattice Point Enumeration Tool (LATTE) [34] 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, δ, p) *outputs $(H^\sharp, H^\flat)$, then*

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

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 [8].

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*

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

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

Proof: We define the random variable Y by $Y(x) = -\log p(X = x)$. 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 [8] it is shown that the variance $Var[Z]$ of Z is bounded from above by

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

The Chebyshev inequality

$$p(|Q - E(Q)| \ge \delta) \le \frac{Var[Q]}{\delta^2} \tag{7}$$

gives upper bounds for the probability that the value of a random variable Q deviates from its expected value $E(Q)$. We apply (7) to Z with the expectation and variance bounds derived above and obtain

$$p(|Z - H(X)| \ge \delta) \le \frac{(\log m)^2}{n\delta^2}\ .$$

The assertion then follows by basic arithmetic. ∎

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

Proof of Theorem 6: Let s'_i be the final state of P that is computed in line 5 of the ith loop iteration of RANT, for $i \in \{1, \ldots, n\}$. For uniformly distributed U, we have $H(U) = \log \#(I)$ and $p(V = s'_i) = \#(P^{-1}(s'_i))/\#(I)$. 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

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

Lemma 1 now implies that

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

with a probability of more than

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

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. ■

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 [1].

B. 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. [8] 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 $p(X = x)$ 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 IV for approximating the probabilities of events on demand, allowing us to derive bounds on the information leakage of real programs.

VI. Related work

For an overview of language-based approaches to information-flow security, see the survey by Sabelfeld and Myers [42].

Denning is the first to quantify information flow in terms of the reduction in uncertainty about a program variable [21]. Millen [39] and Gray [25] use information theory to derive bounds on the transmission of information between processes in multi-user systems. Lowe [35] 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 [17] and has since then become standard, see e.g. [5], [6], [22], [43], [46].

Clark, Hunt, and Malacaria [14] 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 [15]. Their analysis is restricted to straight-line programs.

Malacaria [36] 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 II. In this way, our method can be used to automatically determine this information.

Köpf and Basin [30] 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 [1] 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 [38] 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 [41] 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.

Clarkson, Myers, and Schneider [16] 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 paper. One advantage of entropy-based measures is the direct connection to equiv-

alence relations, which makes them amenable to automated reasoning techniques.

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. [3], [9], [27], [32]. In particular, for bounding block count one could apply tools for discovery of all valid invariants captured by numeric abstract domains, e.g., octagons or (sub-)polyhedra [2], [20], [33], [40], or template based techniques [18], [26]. 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 [10], [11], [13], [23], [24].

Acknowledgments: We thank the anonymous reviewers for helpful feedback.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] T. Ball, T. D. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2), 2005.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *IEEE Trans. Dependable Sec. Comput.*, 5(4):224–241, 2008.

[12] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.

[13] 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.

[14] D. Clark, S. Hunt, and P. Malacaria. Quantitative Information Flow, Relations and Polymorphic Types. *J. Log. Comput.*, 18(2):181–199, 2005.

[15] 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.

[16] 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.

[17] E. Cohen. Information Transmission in Sequential Programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[18] 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.

[19] 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.

[20] 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.

[21] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[22] 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.

[23] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[24] 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.

[25] J. W. Gray. Toward a Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] G. Lalire, M. Argoud, and B. Jeannet. The interproc analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html.

[33] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, LNCS 5403, pages 229–244. Springer, 2009.

[34] 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].

[35] G. Lowe. Quantifying Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 18–31. IEEE, 2002.

[36] P. Malacaria. Assessing security threats of looping constructs. In *Proc. Symp. on Principles of Programming Languages (POPL '07)*, pages 225–235. ACM, 2007.

[37] J. L. Massey. Guessing and Entropy. In *Proc. IEEE Intl. Symp. on Information Theory (ISIT '94)*, page 204. IEEE Computer Society, 1994.

[38] 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.

[39] J. K. Millen. Covert Channel Capacity. In *Proc. IEEE Symp. on Security and Privacy (S&P '87)*, pages 60–66. IEEE, 1987.

[40] A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[41] 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.

[42] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE J. Selected Areas in Communication*, 21(1):5–19, 2003.

[43] 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.

[44] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[45] 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.

[46] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 15–23. IEEE, 2001.