Computing Information Flow Using Symbolic Model-Checking

Rohit Chadha[1], Umang Mathur[2], and Stefan Schwoon[3]

1 **University of Missouri, USA**
2 **Indian Institute of Technology - Bombay, India**
3 **LSV, ENS Cachan & CNRS, INRIA Saclay, France**

Abstract

Several measures have been proposed in literature for quantifying the information leaked by the public outputs of a program with secret inputs. We consider the problem of *computing* information leaked by a deterministic or probabilistic program when the measure of information is based on (a) min-entropy and (b) Shannon entropy. The key challenge in computing these measures is that we need the total number of possible outputs and, for each possible output, the number of inputs that lead to it. A direct computation of these quantities is infeasible because of the state-explosion problem. We therefore propose symbolic algorithms based on binary decision diagrams (BDDs). The advantage of our approach is that these symbolic algorithms can be easily implemented in any BDD-based model-checking tool that checks for reachability in deterministic non-recursive programs by computing program summaries. We demonstrate the validity of our approach by implementing these algorithms in a tool `Moped-QLeak`, which is built upon `Moped`, a model checker for Boolean programs. Finally, we show how this symbolic approach extends to probabilistic programs.

1 Introduction

It is desirable for a program to never leak any information about its confidential inputs, a property usually formalized as *non-interference* [19, 28]. For example, when an adversary can make *low-security observations* of an execution, these should be independent of the confidential inputs. This property is often too strong in practice, mostly because it clashes with desired functionality. Therefore, many authors [17, 20, 27, 30] have proposed to evaluate security by the *amount* of leaked confidential information. This raises foundational questions of (a) how to measure that amount and (b) how to compute it. These challenges have received much attention recently.

A usual approach is to employ information-theoretic tools. There, a program is modeled as an information channel that transforms a random variable taking values from the set of confidential inputs into a random variable taking values from the set of public outputs (or equivalently, the adversary's observations). Based on this, one quantifies the adversary's uncertainty about the confidential inputs. The amount of information leaked by the program is then modeled as the difference between the initial uncertainty and the uncertainty remaining in the secret inputs after the adversary observes the execution (with lower values being more preferable to higher values). Commonly used measures of uncertainty are Shannon entropy [17, 20, 27] and min-entropy [30]. Intuitively, leakage based on min-entropy measures vulnerability of the secret inputs to a single guess of the adversary who observes the program execution, while, leakage based on Shannon entropy measures expected number of guesses required for the adversary to guess the secret input having observed the program execution. We refer to [30] for a detailed comparison between these two measures.

Though appealing from a conceptional viewpoint, these measures do not readily lend themselves to feasible computation. For example, it has been shown in [32] that when

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

using Shannon's entropy for measuring uncertainty, the problem of deciding whether the information leaked by a *loop-free* deterministic Boolean program is less than a rational number is harder than *counting* the number of satisfying assignments of a Boolean formula in Conjunctive Normal Form. The hardness of the problem comes from the fact that one has to compute (a) how many outputs are observable to the adversary and (b) for each possible output, how many inputs lead to that particular output.

When Boolean deterministic programs contain loops, computing information leakage becomes PSPACE-complete [33, 7, 31], for both min-entropy and Shannon entropy. Although this is same complexity as checking safety of Boolean programs (or equivalently, reachability), the decision procedures given in [33, 7, 31] are not feasible in practice. Instead researchers have developed heuristics to exploit reachability tools to compute the amount of information leaked. The reachability tools employed come from model checking [3, 23, 9, 10], static analysis [13, 14, 15, 3], SMT solvers [26, 25], and statistical analysis [23, 8].

Contributions. We first consider the problem of evaluating the amount of information leaked by the public outputs of Boolean *deterministic* programs with uniformly distributed secret inputs. We exploit symbolic model-checking techniques to achieve our goals. More precisely, we demonstrate how model checkers based on Binary Decision Diagrams (BDDs) can very easily be enhanced to compute information leakage. As we shall see shortly, our approach is informed by the model-checking algorithms used by these tools.

BDDs [24, 1, 6] are data structures used to store Boolean functions[1] with finite domain. Their efficiency has led to many applications in program verification. Broadly, in this approach, the program is viewed as a transition system in which a configuration contains the current line number and the values of the variables. Transitions are encoded as BDDs, and reachability is encoded as the least fixed-point solution to a set of Boolean equations. This solution is the result of a fixed-point iteration with efficient BDD operations (Please see [6] for a discussion of complexity of BDD operations). For certain BDD-based tools [4, 18], this fixed-point computation yields the relation between the values of global variables at the start of the program and the values of the global variables when the queried location is reached. By querying the exit point, we can thus compute the relation between the inputs and the outputs of the program, henceforth referred to as the *summary* of the program.

Our key observation is that this summary (which is given as a BDD) is indeed all the information we need to quantify information leakage. We give symbolic algorithms that extract information leakage from the summary according to either Shannon entropy or min-entropy. This approach is appealing because these algorithms can be easily plugged into existing BDD-based model-checking tools.

We validate this approach by implementing[2] our algorithms in `Moped` [18], a BDD-based symbolic model checker that checks for assertion errors in non-probabilistic programs modeled as reachability problems. Apart from providing support for Boolean data, `Moped` also supports integers of variable length, arrays, and C-like structures. Our experience with these implementations are promising, as the computation of information leakage (for both min-entropy and Shannon-entropy) comes with little overhead over the reachability computation. Although, `Moped` supports recursion also, we currently provide support only for non-recursive programs.

We then turn our attention to probabilistic programs. For probabilistic non-recursive

[1] Boolean functions are functions that take values in the set $\{0, 1\}$.
[2] The tool implementing the algorithms for entropy calculations is available for download at `http://people.cs.missouri.edu/~chadhar/mql/`

programs, we need to compute, for each possible input-output pair (i, o), the conditional probability that the program outputs o when the input is i. Usually, these quantities are stored as a matrix, also called the *channel matrix.* We compute the channel matrix as the least fixed-point solution to a system of *linear equations* [29]. Many probabilistic model checkers such as [21] do this computation using Algebraic Decision Diagrams (ADDs) [16], a generalization of BDDs. The summary for probabilistic programs now encodes (symbolically) the channel matrix, and we construct symbolic algorithms to extract the leaked information from the computed summary. We validate this approach by first extending the ability of `Moped` to compute the summary for probabilistic non-recursive programs and then implementing the symbolic algorithms for computing the information leakage. Once again, this computation comes with little overhead over the reachability computation.

Related work. In recent years, several automated approaches from model checking [3, 23, 9, 10], static analysis [13, 14, 15, 3], and statistical analysis [23, 8] have been employed to compute information leakage. We discuss the most closely related works.

The problem of automatically computing information flow was first tackled in [3]. This approach iteratively constructs equivalence classes on inputs: two inputs are said to be equivalent if they lead to the same output. One starts with a single equivalence class and progressively refines when these inputs lead to different outputs. At each step, the equivalence relation is characterized using logical formulas and refined using experimental runs of the program. Once a fixed point is reached, they use the size of the resulting equivalence classes to compute information leakage. This technique is optimized in [23], where statistical techniques are used to estimate the equivalence classes.

SMT solvers are used in [26, 25] to estimate min-entropy leakage in Boolean programs. Since min-entropy leakage of deterministic programs is determined by the number of different feasible outputs, they estimate min-entropy leakage by estimating an upper bound on the number of feasible outputs. Concretely, they compute, using SMT solvers, how many different values each pair of output bits can take. This technique only yields an upper bound, and it is easy to construct examples where this upper bound is far from the correct value. Our techniques in contrast yield exact values.

For probabilistic programs, the use of model-checking to compute information leakage has been explored in [9, 2, 12, 5]. Please note that the models considered in these papers are more general as they also allow for other observations than just the outputs at the end of the program execution. [9] uses [21] to get the channel matrix and then computes the information leakage by hand, [12] implements an explicit state model-checking algorithm, and [5] computes the information leakage using (forward) symbolic executions. [2] also proposes to compute the channel matrix using fix-point iterations. Once the channel matrix is computed *explicitly* then information leakage can be computed. Our approach is different in that we solve the fix-point iterations *symbolically* and use the *symbolic* representation of the computed matrix directly in the computation of information leakage.

2 Preliminaries

We recall some standard definitions and establish notations. Please note that our notations closely follow [7]. For a finite set A, $|A|$ shall denote the number of elements of A. For a function $f : A \to B$ and $b \in B$, $f^{-1}(b)$ denotes the set $\{a \mid f(a) = b\}$. We write 2^A to mean both the set of functions $A \to \{\texttt{true}, \texttt{false}\}$ as well as the set of subsets of A. All logarithms are to the base 2 and $0 \log 0 := 0$. The set of real numbers will be denoted by $\mathbb{R}$ and the non-negative reals by $\mathbb{R}^+$.

2.1 Boolean Programs

We first consider non-recursive Boolean deterministic programs in this paper. We shall demonstrate later how to extend our techniques to Boolean probabilistic programs. The programs that we consider have global and local variables. Usually, when talking about information leakage, it is assumed that there is a set of *high-security input variables* and a set of *low-security output variables.* However, in this paper we will assume that the secret inputs to the program are the *initial* values of the global variables and that the public outputs are the *final* values of the global variables. Thus, we do not explicitly separate high-security input-only variables and low-security output-only variables. This does not cause a loss of expressiveness: if we want to make sure that changes to a high-security input-only variable by a program are not visible to the adversary, we can set them to `false` upon exit. Similarly, if we want to explicitly designate some variables as low-security output-only variables, we can initialize all of them to `false` at the beginning of the program.

Assuming that local variables are always initialized to `false`, the semantics of a program P with global variables $\mathcal{G}$ can be seen as a function $F_P \colon S \to O$ where $S = 2^{\mathcal{G}}$ and $O = 2^{\mathcal{G}} \cup \{\bot\}$. $F_P(\bar{g_0}) = \bot$ iff P does not terminate on $\bar{g_o}$, and otherwise $F_P(\bar{g_0}) \in 2^{\mathcal{G}}$ is the valuation of the global variables when P stops executing. From now on, we will confuse P with the function F_P. Note that we are treating non-termination as an explicit observation of the attacker.

Assuming that the inputs are sampled from a distribution μ, let $\mathcal{S}$ be the random variable taking values in S according to μ. μ can be extended to a joint probability distribution on S and O by setting $\mu(\mathcal{O} = o \mid \mathcal{S} = s) = 1$ if $P(s) = o$ and 0 otherwise.

2.2 Information leakage in programs

Several measures of information leakage have been considered in literature. Of these, we consider Shannon entropy and min-entropy. We assume that the reader is familiar with information theory and introduce some abbreviations and results that we shall need. For this section, we fix a Boolean program P. As discussed above, the semantics of P is a function $P : S \to O$. If S is sampled from a distribution μ, then μ gives rise to a joint probability distribution on S and O.

Leakage based on Shannon entropy. In Shannon entropy, the *information leaked by the program P* is defined as $\mathrm{SE}_\mu(P) \coloneqq \mathrm{I}_\mu(\mathcal{S};\mathcal{O})$, where $\mathcal{S}$ and $\mathcal{O}$ are random variables taking values in S and O respectively according to the joint distribution μ, and $\mathrm{I}_\mu(\mathcal{S};\mathcal{O})$ is the mutual information of random variables $\mathcal{S}$ and $\mathcal{O}$. This definition holds even when we allow P to be probabilistic. When P is deterministic and μ is U, the uniform distribution on inputs, we have [3, 22]: $\mathrm{SE}_{\mathsf{U}}(P) = \log |S| - \frac{1}{|S|} \sum_{o \in O} |P^{-1}(o)| \log |P^{-1}(o)|$.

Leakage based on Min entropy. In min-entropy [30], the *information leaked by the program P* on uniformly distributed inputs is defined as $\mathrm{ME}_{\mathsf{U}}(P) \coloneqq \log \sum_{o \in \mathcal{O}} \max_{s \in \mathcal{S}} \mu(\mathcal{S} = s \mid \mathcal{O} = o)$. This holds even when we allow P to be probabilistic. When P is deterministic [30], we get that $\mathrm{ME}_\mu(P) \coloneqq \log |\mathcal{O}'|$, where $\mathcal{O}' = \{o \in \mathcal{O} \mid \exists s \in \mathcal{S} : P(s) = o\}$ are the outputs that can actually be realized.

2.3 Algebraic Decision Diagrams

We assume that the reader is familiar with Binary Decision Diagrams (BDDs) and merely recall some facts necessary for our presentation. Note that when speaking of BDDs, we always mean their reduced ordered form [6]. BDDs are data structures for storing elements

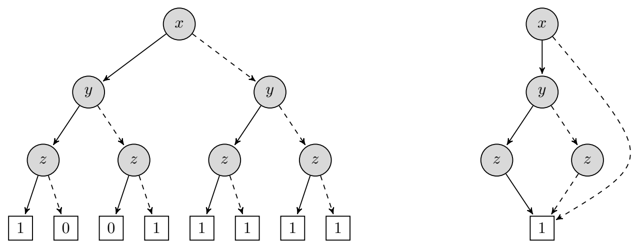

Figure 1 An unreduced decision diagram (left) and and corresponding BDD (right).

of $2^{\mathcal{V}} \to \{0,1\}$, where $\mathcal{V} = \{x_1,\ldots,x_n\}$ is a finite set of Boolean variables. They take the form of a rooted, directed acyclic labeled graph. Non-terminal nodes are labeled by an element of $\mathcal{V}$, and terminals are either 0 or 1. There are two edges out of a non-terminal node, one labeled *then* and the other labeled *else*. Assuming a fixed strict order $<$ on $\mathcal{V}$, an edge from a non-terminal labeled x to a non-terminal labeled y satisfies $x < y$. From now on we will often confuse a function $2^{\mathcal{V}} \to \{0,1\}$ with its BDD representation.

▶ **Example 1.** Figure 1 shows how a BDD over the set $\mathcal{V} = \{x, y, z\}$ with the order $x < y < z$ would store the Boolean assignments satisfying $x \to (y \leftrightarrow z)$. The figure on the left shows a (non-reduced) diagram exhaustively listing all assignments, and the right-hand side shows the resulting BDD, where for simplicity the terminal 0 and edges leading to it have been omitted. Note that the solid arrows are *then* branches and the dashed arrows are *else* branches.

ADDs are a generalization of BDDs that store elements of the set $2^{\mathcal{V}} \to M$, where $\mathcal{V} = \{x_1,\ldots,x_n\}$ and M is an arbitrary set. The main difference between BDDs and ADDs is that the terminal nodes contain elements of M and not just elements of $\{0,1\}$. For our purposes, M will be either $\mathbb{R}$ or $\mathbb{R}^+$. Analogous to BDDs, the value of a function f represented by an ADD T at $(z_1,\ldots,z_n) \in 2^{\mathcal{V}}$ is given by the label of the terminal node along the unique path from the root to a terminal node such that if a non-terminal node is labeled x_i along the path then the outgoing edge from x_i must be labeled *then* if and only if z_i is `true`.

Note that an BDD is an ADD where all the terminals are either 0 or 1. Henceforth, we will refer to BDDs as 0/1-ADDs. Many efficient operations can be performed on ADDs. We list the most relevant ones for our paper.

1. The function $\mathsf{isConst}(T)$ checks if T is a constant function. $\mathsf{val}(T)$ returns the value of T if $\mathsf{isConst}(T)$ is true.
2. If *op* is a commutative and associative binary operator on $\mathbb{R}$ and $\mathcal{V}_1$ a subset of variables of $\mathcal{V}$ then $\mathsf{abstract}(op, \mathcal{V}_1, T)$ returns the result of abstracting all the variables in $\mathcal{V}_1$ by applying the operator *op* over all possible values taken by variables in $\mathcal{V}_1$. $\mathsf{abstract}(op, \mathcal{V}_1, T)$, thus obtained, is a function with domain as the set $\mathcal{V} \setminus \mathcal{V}_1$ and range as $\mathbb{R}$.
   For example, if T represents the function f, then $\mathsf{abstract}(+, \{x_1, x_2\}, T)$ returns the ADD which represents the function
   $$f(\mathtt{true}, \mathtt{true}, x_3, \ldots, x_n) + f(\mathtt{true}, \mathtt{false}, x_3, \ldots, x_n) + \\ f(\mathtt{false}, \mathtt{true}, x_3, \ldots, x_n) + f(\mathtt{false}, \mathtt{false}, x_3, \ldots, x_n).$$
3. If T is a 0/1-ADD and $\mathcal{V}_1$ a subset of $\mathcal{V}$, then $\mathsf{orAbstract}(\mathcal{V}_1, T)$ returns the result of abstracting all the variables in $\mathcal{V}_1$ by applying disjunction over all possible values taken by variables in $\mathcal{V}_1$.

3 Leakage in non-probabilistic programs

In this section, we shall describe our ADD-based algorithms for computing the information leaked by deterministic programs when the leakage is measured using (a) min-entropy and (b) Shannon entropy. We fix some notation. Consider a set of variables $\mathcal{G} = \{x_1, \ldots, x_n\}$. Let $\mathcal{G}' = \{x'_1, \ldots, x'_n\}$ be a set of distinct variables disjoint from $\mathcal{G}$. Note that there is a one-to-one correspondence between elements of $2^{\mathcal{G}}$ and $2^{\mathcal{G}'}$ and every element $(z'_1, \ldots, z'_n)$ of $2^{\mathcal{G}'}$ can be identified with a unique element $(z_1, \ldots, z_n)$ of $2^{\mathcal{G}}$ and vice versa. $\mathcal{G}$ shall represent the initial values of the variables of a program and and $\mathcal{G}'$ shall represent their final values. In this section, we will assume that all possible valuations of $\mathcal{G}$ are valid inputs to the program (and hence our input domain shall always be a power of 2). We discuss how to restrict the domain in Appendix C.

▶ **Definition 2. (Summary of a Program)** Let P be a program with $\mathcal{G} = \{x_1, \ldots, x_n\}$ as the set of global variables. Let $\mathcal{G}' = \{x'_1, \ldots, x'_n\}$ be a set of distinct variables disjoint from $\mathcal{G}$. The *summary* of P, denoted T_P, is a function $T_P : 2^{(\mathcal{G} \cup \mathcal{G}')} \to \{0, 1\}$ such that for every $z_1, \ldots, z_n, z'_1, \ldots, z'_n \in \{\mathtt{true}, \mathtt{false}\}$, we have $T_P(z_1, \ldots, z_n, z'_1, \ldots, z'_n) = 1 \iff P(z_1, \ldots, z_n) = (z'_1, \ldots, z'_n)$.

Observe that thanks to the correspondence between OBDDs and Boolean functions, T_P can be considered as an OBDD on the set of variables $\mathcal{G} \cup \mathcal{G}'$. Now, T_P can be seen as the least fixed point of a system of Boolean equations, which can efficiently be constructed by iterative methods. For our purposes, it suffices to say that BDD-based model-checkers essentially construct this relation for us (and, if not, can be modified to carry out this construction). We assume for our paper that T_P is constructed by a BDD-based model-checker. It remains to show how to exploit T_P to compute the information leaked by P.

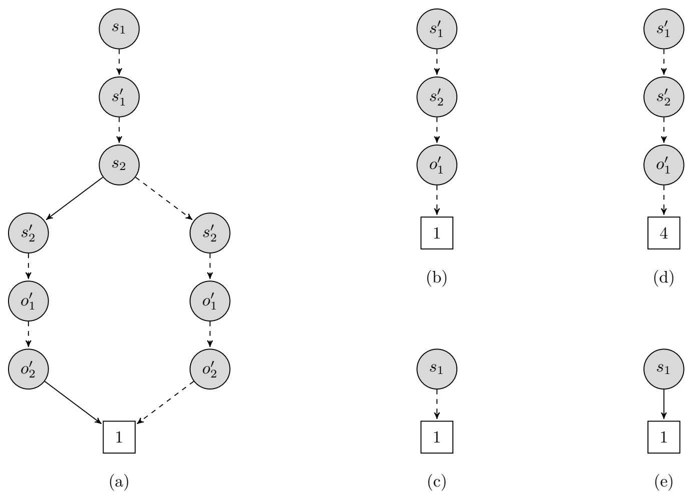

Figure 2 (a) Transition relation of the program P_{ex}. The ordering assumed is $s_1 < s'_1 < s_2 < s'_2 < o_1 < o'_1 < o_2 < o'_2$. (b) All the possible outputs of the program P_{ex} as an ADD. (c) All possible inputs on which P_{ex} terminates represented as an ADD. (d) $T_{\text{eq-size},P}$ for P_{ex} as an ADD. (e) $T_{\text{non-term},P}$ for the program P_{ex} as an ADD.

▶ **Example 3.** Consider the following program P_{ex} with 4 global Boolean variables: $s1, s2, o1$ and $o2$.

```
o1 = false; o2 = false;
```

```
while s1 {};
o1 = false; o2 = s2;
s1 = false; s2 = false;
```

Here, variables $s1$ and $s2$ are high-security input-only variables and $o1$, $o2$ low-security input variables. This is why we initialized o_1 and o_2 to be `false` and set s_1 and s_2 `false` before the end of the program. Observe also that the program does not terminate when $s1$ is `true` at the beginning of the program. Assuming the order $s_1 < s_1' < s_2 < s_2' < o_1 < o_1' < o_2 < o_2'$, the transition relation of P is shown as a 0/1-ADD in Figure 2 (a).

For the rest of the section, unless otherwise stated, we will fix the Boolean program P. We assume that $\mathcal{G} = \{x_1, \ldots, x_n\}$ is the set of global variables of P and that $\mathcal{G}' = \{x_1', \ldots, x_n'\}$ is a set of distinct variables disjoint from $\mathcal{G}$. The *summary* of P will be referred to as T_P.

Leakage measured using min-entropy. The amount of information leaked by the program P when using min-entropy is as follows (cf. Section 2):

- Let $\mathrm{post}(2^{\mathcal{G}}) = \{\bar{g'} \in 2^{\mathcal{G}} \mid \exists \bar{g} \in 2^{\mathcal{G}}.\ P(\bar{g}) = \bar{g'}\}$. If the program P terminates on all inputs then the min-entropy leakage is $\log|\mathrm{post}(2^{\mathcal{G}})|$.
- If there is some input $\bar{g} \in 2^{\mathcal{G}}$ such program P does not terminate on $\bar{g}$ then the min-entropy leakage is $\log(|\mathrm{post}(2^{\mathcal{G}})| + 1)$.

Thus, to compute the min-entropy leakage, we need to compute $|\mathrm{post}(2^{\mathcal{G}})|$ and check if there is an input on which the program P never terminates. The following lemma shows how these two tasks can be achieved using ADDs. (See Appendix A for the proof.)

▶ **Lemma 4.** *Let $T_{out,P} = \mathit{orAbstract}(\mathcal{G}, T_P)$ and $T_{term,P} = \mathit{orAbstract}(\mathcal{G}', T_P)$.*

1. $|\mathrm{post}(2^{\mathcal{G}})| = \mathit{val}(\mathit{abstract}(+, \mathcal{G}', T_{out,P}))$.

2. *P terminates on every input iff $\mathit{isConst}(T_{term,P})$ and $\mathit{val}(T_{term,P}) = 1$.*

▶ **Example 5.** Consider the program P_{ex} in Example 3 with $\mathcal{G} = \{s_1, s_2, o_1, o_2\}$. Observe that the program terminates only when s_1 is `false`, in which case the final value of s_1 is also `false`. The initial values of o_1 and o_2 do not effect the output. The final values of s_2 and o_1 are always `false`. The value of o_2 is exactly the value of s_2. Thus, there are two possible outputs (`false`, `false`, `false`, `true`) and (`false`, `false`, `false`, `false`), both of which happen for exactly 4 inputs. The ADD representing $T_{\mathsf{out},P}$, the set of all possible outputs of P is given in Figure 2 (b). Note that o_2' does not appear in the picture because the *then* and *else* branches of o_2' lead to isomorphic subtrees. Observe that $\mathsf{abstract}(+, \mathcal{G}', T_{\mathsf{out},P})$ is the constant ADD 2. The ADD $T_{\mathsf{term},P}$ representing all possible inputs on which P terminates is given in Figure 2 (c).

Algorithm 1: Symbolic computation of min-entropy leakage of a deterministic program

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{ME_U}(P)$
1 begin
2     $T_{\mathsf{out},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}, T_P)$
3     $\mathsf{num_{out}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}', T_{\mathsf{out},P}))$
4     $T_{\mathsf{term},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}', T_P)$
5     if $\mathit{isConst}(T_{term,P}) = \mathtt{false}$ or $\mathit{val}(T_{term,P}) = 0$ then
6         $\mathsf{num_{out}} \longleftarrow \mathsf{num_{out}} + 1$;
7     return $\log \mathit{num_{out}}$
```

Thanks to Lemma 4, we have the following theorem:

▶ **Theorem 6.** *For a program P with global variables $\mathcal{G} = \{x_1, \ldots, x_n\}$, let $\mathcal{G}' = \{x'_1, \ldots, x'_n\}$ be a set of distinct variables disjoint from $\mathcal{G}$. Let T_P be the summary of P represented as a 0/1-ADD on $\mathcal{G} \cup \mathcal{G}'$. The Algorithm 1 computes* $\mathrm{ME}_{\mathsf{U}}(P)$.

Leakage measured using Shannon entropy. We now consider information leaked by P when measured using Shannon entropy. We need to compute $\sum_{\bar{g'} \in 2^{\mathcal{G}'}} |P^{-1}(\bar{g'})| \log|P^{-1}(\bar{g'})| + |P^{-1}(\perp)| \log|P^{-1}(\perp)|$. In order to compute this sum, we need a new auxiliary definition:

▶ **Definition 7.** Let $\star : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$ be the binary operator defined as $r_1 \star r_2 = r_1 \log r_1 + r_2 \log r_2$.

Algorithm 2: Symbolic computation of Shannon entropy leakage of a deterministic program

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{SE}_{\mathsf{U}}(P)$
1 Let $n$ be the number of variables in $\mathcal{G}$.
2 begin
3     $T_{\text{eq-size},P} \longleftarrow \mathsf{abstract}(+, \mathcal{G}, T_P)$
4     $\mathsf{sum} \longleftarrow \mathsf{val}(\mathsf{abstract}(\star, \mathcal{G}', T_{\text{eq-size},P}))$
5     $T_{\text{term},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}', T_P)$
6     $T_{\text{non-term},P} \longleftarrow \mathsf{cmpl}(T_{\text{term},P})$
7     $\mathsf{num}_{\text{non-term}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\text{non-term},P}))$
8     $\mathsf{sum} \longleftarrow \mathsf{sum} + \mathsf{num}_{\text{non-term}} \log(\mathsf{num}_{\text{non-term}})$
9     return $(n - \frac{\mathsf{sum}}{2^n})$
```

We have the following. (See Appendix B for the proof).

▶ **Theorem 8.** *For a program P with global variables $\mathcal{G} = \{x_1, \ldots, x_n\}$, let $\mathcal{G}' = \{x'_1, \ldots, x'_n\}$ be a set of distinct variables disjoint from $\mathcal{G}$. Let T_P be the summary of P represented as a 0/1-ADD on $\mathcal{G} \cup \mathcal{G}'$. Algorithm 2 computes* $\mathrm{SE}_{\mathsf{U}}(P)$.

▶ **Example 9.** Consider the program P_{ex} in Example 3. Recall that there are two possible outputs (`false`, `false`, `false`, `true`) and (`false`, `false`, `false`, `false`), both of which happen for exactly 4 inputs. The ADD $T_{\text{eq-size},P}$ for the P_{ex} is depicted in Figure 2 (d). The program does not terminate whenever s_1 is `false`. The ADD $T_{\text{non-term},P}$ is depicted in Figure 2 (e).

4 Experimental evaluation

In this section, we present some results based on our experiments for calculating the different leakage values using the tool `Moped-QLeak`. It is based on the existing BDD-based symbolic model-checker `Moped` [18] that checks for assertion errors modelled as reachability problems. `Moped`, apart from providing support for basic Boolean data, also supports complex data types such as integers of variable length, arrays and C-like structures. `Moped` is based on the CUDD (Colorado University Decision Diagram) package.

`Moped-QLeak` performs basic reachability analysis and generates a summary of an input program written in *Remopla*. This summary is then used to calculate the information leakage. Currently, `Moped-QLeak` only supports non-recursive programs and is currently available for download at: `http://people.cs.missouri.edu/~chadhar/mql/`

`Moped` translates Remopla programs into BDDs. `Moped-QLeak` re-uses this as a frontend, but internally works with the more generic ADD structure to carry out the calculations presented in Section 3. Other than that, we made the following optimizations with respect to the standard behavior of `Moped`.

Example	Order	ME	SE	Time	Data types
Illustrative Example	I	3	2.03966e-05	0.215	bool
Electronic Purse	D	2	2	0.009	5 bit integers (Restricted)
Mix and Duplicate	S	16	16	0.041	bool
Binary Search	I	16	16	9.307	bool
Sanity Check	I	4	1.16797e-7	0.060	bool
Implicit Flow	D	2.80735	0.00301072	0.010	15 bit integers
Implicit Flow	D	2.80735	1.757e-07	0.016	30 bit integers
Implicit Flow	D	2.80735	4.67189e-08	0.190	bool
Masked Copy	I	16	16	0.038	bool
Sum Query	D	4.80735	4.35132	0.034	5 bit integers (Restricted)
Ten Random Outputs	D	3.32193	2.6355e-07	0.055	30 bit integers

Table 1 Examples used for evaluation

Algebraic operations: The input language of `Moped` understands expressions using algebraic and Boolean operations. However, `Moped` was not conceived with large integer operands in mind, and we detected some inefficiencies in these translations for integer operands having large number of bits. These often drastically affected the overall time taken for calculation of the summary, in which cases we improved the translation. Also, for the purpose of experimental evaluation of the efficiency of `Moped-QLeak`, we encode all the examples with variables having large range as Boolean programs and note a striking change in the running times. Furthermore, `Moped` does not support integers with bit length > 30. Hence, all examples with bit length > 30 were also coded as Boolean programs.

Size of ADDs and variable orderings: As usual with symbolic methods, their efficiency is highly sensitive to the size of the decision diagrams generated during the course of the reachability analysis, which, in turn, may depend on the variable ordering. (Finding the most efficient ordering is a NP-hard problem). `Moped` does not automatically determine the best variable ordering and gives the user the flexibility to choose the ordering. Hence, the examples for which the default ordering of variables (which entails the order of declaration of the variables in the source file) was the overhead, have been re-written with supposedly efficient variable orderings. The principal obstacle here is the computation of summary. The computation of leakage itself adds little overhead.

We illustrate our orderings using the variables O (for public outputs) and S (for private inputs). Let O_N (O_1) be the most (least) significant bit of O and likewise S_N (S_1) the most (least) significant bit of S. We primarily used two kinds of orderings:

- Contiguous ordering: This is the default ordering of the tool, where we set $O_1 < O_1' < O_2 \cdots O_N' < S_1 < S_1' < S_2 < \cdots < S_N'$.
- Interleaved ordering: In this ordering, we set $O_1 < O_1' < S_1 < S_1' < O_2 < O_2' < S_2 < S_2' < \cdots O_N < O_N' < S_N < S_N'$.

The choice of an ordering depends largely on the structure and semantics of the program. The ADDs produced are generally smaller if a variable v_1 is closer to a variable v_2 such that the value of v_1 depends on the value of v_2. Essentially, as long as variables are compared and assigned to constants in the program, the default ordering works very well and in that case we do not even attempt the interleaved ordering. For other examples, typically, we switch to interleaved ordering as contiguous ordering becomes inefficient very fast with the number of bits as the ADDs become very large. Going by this, we have also reordered variable declarations in an example (see Mix and Duplicate below) so that variables with a constant difference in the *indices* are closer. Table 1 presents some selected benchmark programs that we used to test `Moped-QLeak`. The examples have been derived from [26]. The experiments were conducted on a 64-bit Xeon-X5650 2.67GHz Linux machine. Unless otherwise stated, S

and O are 32-bit unsigned integers in all the programs. For each example, we give the name, the ordering, the Shannon entropy (SE) and min-entropy (ME) leakage values, the execution time of the tool in seconds, and the data types that occur in the example, which are either all Boolean or integers with a specified number of bits. If the example uses restricted domains then we mention it in the data types. The order is either the contiguous default order (D), the interleaved order (I), or another example-specific order (S). There is one example from [26], Population Count, for which the computation of summary never succeeds as there is no good variable ordering for that example. Note that we run the tool to compute the two leakage values separately and report the worse case. The time difference between the two values is almost always within a 3-4 microseconds. We discuss two interesting examples below and discuss Implicit Flow and Ten Random Outputs in Appendix D.

Mix and Duplicate. The following program copies the XOR of the i^{th} and the $(i+16)^{th}$ bit of S to both the i^{th} and the $(i+16)^{th}$ bit of O.

```
O = ((S >> 16) ^ S) & 0xffff;
O = O | O << 16;
```

It is thus that the i^{th} and the $(i+16)^{th}$ bits of S and O are closely related. In fact, the ADDs formed after reordering the variables to $O_{17} < O'_{17} < O_1 < O'_1 < S_{17} < S'_{17} < S_1 \cdots O'_{16} < S_{32} < S'_{32} < S_{16} < S'_{16}$ have drastic reduction in the number of distinct nodes. Note that intuitively, half the input bits are leaked in the example (namely the XOR of i^{th} and $(i+16)^{th}$ bits of S). This intuition is confirmed by the results. **Binary Search.** The following program scans the first b bits of the input S and puts a 1 at the i^{th} bit of O iff the i^{th} bit of S is 1.

```
O = 0;
for (i = 0; i < b; i++)
  {m = 2^(31-i);
  if (O + m <= S) O += m; }
```

For our experiments, we took $b = 16$. We converted this program to a Boolean program with an interleaved ordering. We also unrolled the loop for $b = 16$. Also note that since O is 0 to start with and m is a power of 2, the addition of O and m can be modelled as bitwise or of O and m for the purpose of efficiency. It can also be checked (using assertion-checking in `Moped`) that the $(31 - i)^{th}$ bit is false before the i^{th} iteration, and thus the carry-bit is always 0, justifying our simplification. Note that intuitively, half the input bits (the first 16 bits) are leaked by the program. This intuition is confirmed by the results.

5 Leakage in probabilistic programs

We also generalized our algorithms for computing information leakage in programs that allow probabilistic choices. The *summary* of a probabilistic program is the *channel matrix*. The channel matrix on inputs S and outputs O is the $S \times O$ matrix such that its (s, o) entry is the conditional probability of observing o given s. More precisely, for a probabilistic program P with $\mathcal{G} = \{x_1, \ldots, x_n\}$ as the set of global variables, and $\mathcal{G}' = \{x'_1, \ldots, x'_n\}$ a set of distinct variables disjoint from $\mathcal{G}$, the *summary* of program P, denoted by T_P is the function $T_P : 2^{(\mathcal{G} \cup \mathcal{G}')} \to \mathbb{R}^+$ such that for every $z_1, \ldots, z_n, z'_1, \ldots, z'_n \in$ {`true`, `false`}, $T_P(z_1, \ldots, z_n, z'_1, \ldots, z'_n)$ is the conditional probability that the programs outputs $(z'_1, \ldots, z'_n)$ given that the input to the program P is $(z_1, \ldots, z_n)$.

Just as the case for non-probabilistic programs, the summary relation for probabilistic programs can be computed using ADD-based fixed-point algorithms. This is indeed carried out in several symbolic model-checkers for probabilistic systems such as PRISM [21]. Once again, we can give symbolic algorithms to compute the information leaked (see Appendix E). Please note that currently we do not support restricted domains.

We have implemented these symbolic algorithms in `Moped-QLeak`. `Moped` does not support probabilistic model-checking, so we also implemented the symbolic fixed-point algorithms for computing the summary also in `Moped-QLeak`. We used `Moped-QLeak` to compute information leakage in the dining cryptographer's problem. The results are discussed in Appendix F.

6 Conclusions and future work

We gave symbolic algorithms for computing the information leaked by Boolean programs when information leakage is measured using min-entropy and Shannon entropy. The advantage of our approach is that these algorithms can be integrated with any BDD-based model checking tool that computes reachability in Boolean programs. We made such an integration with `Moped`, with promising experimental results. The leakage calculations themselves add little overhead. The main limiting factor in these calculations seems to be the size of the OBDDs constructed in the computation. As is standard with symbolic approaches, the size of BDDs is sensitive to the variable ordering. Since `Moped` by itself does not compute the most efficient ordering (and puts the onus on the user), we sometimes had to rewrite our examples to achieve good performance. We also generalized our symbolic algorithms for computing information leakage in probabilistic programs. These algorithms have also been integrated in `Moped`.

In order to make symbolic modelchecking more amenable to automation, many automated abstraction refinement techniques have been proposed in literature. We plan to investigate these techniques for our symbolic algorithms. In particular, we plan to integrate the counterexample guided abstraction-refinement framework in our symbolic algorithms. Currently, our implementation only supports non-recursive programs. However, the algorithms we presented for computing information leakage assume only that program summaries be computed. Thus, in principle, we can support programs that have both recursion and probabilistic choices, and we plan to extend support to such programs in future.

References

1 S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

2 M. E. Andrés, C. Palamidessi, P. van Rossum, and G. Smith. Computing the leakage of information-hiding systems. In *TACAS*, pages 373–389, 2010.

3 M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

4 T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. SPIN*, LNCS 1885, pages 113–130, 2000.

5 F. Biondi, A. Legay, L.Traonouez, and A. Wasowski. Quail: A quantitative security analyzer for imperative code. In *Computer Aided Verification*, pages 702–707, 2013.

6 R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

7 R. Chadha, D. Kini, and M. Viswanathan. Quantitative information flow in boolean programs. In *Principles of Security and Trust*, pages 103–119, 2014.

8 K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *TACAS '10*, pages 390–404, 2010.

9 K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Probability of error in information-hiding protocols. In *CSF '07*, pages 341–354, 2007.

10 K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4), 2008.

11 D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

12 T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker. Probabilistic point-to-point information leakage. In *Computer Security Foundations Symposium*, pages 193–205, 2013.

13 D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science (Proc. QAPL '04)*, 112:49–166, 1984.

14 D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic Computation*, 15(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 E.M. Clarke, K.L. Mcmillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. *Formal Methods in System Design*, 10(2-3):137–148, 1997.

17 D. E. R. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

18 J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *TACAS*, LNCS 3920, pages 489–503, 2006.

19 J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

20 J. W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.

21 A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, pages 441–444, 2006.

22 B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In *ACM Conference on Computer and Communications Security*, pages 286–296, 2007.

23 B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF '10*, pages 3–14, 2010.

24 C. Y Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

25 Z. Meng and G. Smith. Faster two-bit pattern analysis of leakage. 2nd International Workshop on Quantitative Aspects of Security Assurance, 2013.

26 Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *PLAS '11*, 2011.

27 J. K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

28 J. C. Reynolds. Syntactic control of interference. In *POPL '78*, pages 39–46, 1978.

29 J. M. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems.* AMS, 2004.

30 G. Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.

31 P. Černý, K. Chatterjee, and T. A. Henzinger. The complexity of quantitative information flow problems. In *CSF '11*, pages 205–217, 2011.

32 H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF '10*, pages 15–27, 2010.

33 H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. In *QAPL 2012*, pages 77–91, 2012.

A Proof of Lemma 4

The ADD $T_{\mathsf{out},P}$ represents a function from domain $2^{\mathcal{G}'}$ to the set $\{0,1\}$. Thanks to definition of $\mathsf{orAbstract}(\mathcal{G}', T_P)$ and the definition of the *summary* relation, we have that for each $\bar{g'} \in 2^{\mathcal{G}'}$, $T_{\mathsf{out},P}(\bar{g'}) = 1$ iff $\bar{g'} \in \mathrm{post}(2^{\mathcal{G}})$. Now, by the definition of ADD operations:

$$\mathsf{val}(\mathsf{abstract}(+, \mathcal{G}', T_{\mathsf{out},P})) = \sum_{\bar{g'} \in 2^{\mathcal{G}'}} T_{\mathsf{out},P}(\bar{g'}) = |\mathrm{post}(2^{\mathcal{G}})|.$$

Note that thanks to definition of $\mathsf{orAbstract}(\mathcal{G}', T_P)$, $T_{\mathsf{term},P}(\bar{g}) = 1$ iff there is a $\bar{g'}$ such that $T_P(\bar{g}, \bar{g'}) = 1$. Observe now that since P is a deterministic program, we have that, on any input $\bar{g}$ either P terminates or does not terminate. Thus, P will terminate on $\bar{g}$ iff there is a $\bar{g'}$ such that $T_P(\bar{g}, \bar{g'}) = 1$. This means, P will terminate on $\bar{g}$ if $T_{\mathsf{term},P}(\bar{g}) = 1$. The lemma follows.

B Proof of Theorem 8

Consider the ADD $T_{\mathsf{eq\text{-}size},P} = \mathsf{abstract}(+, \mathcal{G}, T_P)$. Observe that by definition of the summary relation:

$$T_{\mathsf{eq\text{-}size},P}(\bar{g'}) = \sum_{\bar{g} \in \mathcal{G}} T_P(\bar{g}, \bar{g'}) = |P^{-1}(\bar{g'})|.$$

It is also easy to see that

$$\sum_{\bar{g'} \in 2^{\mathcal{G}'}} |P^{-1}(\bar{g'})| \log |P^{-1}(\bar{g'})| = \mathsf{val}(\mathsf{abstract}(\star, \mathcal{G}', T_{\mathsf{eq\text{-}size},P})).$$

Thanks to our observations above, we only need to show that $\mathsf{num}_{\mathsf{non\text{-}term}}$ computes the number of inputs on which the program does not terminate. Consider the ADD $T_{\mathsf{term},P} = \mathsf{orAbstract}(\mathcal{G}', T_P)$ on $\mathcal{G}$. As in the proof of Proposition 4, we have that $T_{\mathsf{term},P}(\bar{g}) = 1$ iff P terminates on $\bar{g}$. Let $T_{\mathsf{non\text{-}term},P} = \mathsf{cmpl}(T_{\mathsf{term},P})$. We get that $T_{\mathsf{non\text{-}term},P}(\bar{g}) = 1$ iff P does not terminate on $\bar{g}$. Thus, the quantity $\mathsf{num}_{\mathsf{non\text{-}term}} = \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\mathsf{non\text{-}term},P}))$ is the number of inputs on which the program P does not terminate.

C Leakage computation with restricted set of inputs

In Section 3, we showed how we can compute leakage in a program P with $\mathcal{G}$ as the set of variables when any possible valuation of $\mathcal{G}$ is a valid input to the program. It is sometimes useful to consider restricted domains (when we allow only a subset of all possible valuations to be valid). Our tool supports restricted domains in the case of non-probabilistic programs and we describe here how this is achieved.

First, we reserve a special boolean variable out_of_domain for the purposes of ruling out certain valuations. This variable is always initialized to false in the tool. If the user wants to exclude certain inputs in computation, the user may set this variable to be true for those particular inputs. Our tool automatically does not include those inputs for which the out_of_domain is set to *true* by the end of the program execution in the computation of information leakage.

▶ **Example 10.** Consider the electronic purse example from [23]:

```
O = 0;
while(S >= 5)
 {
   S = S - 5;
   O = O + 1;
  }
```

S above is the amount of money that a client has in a bank and O counts the number of times the client can withdraw 5 dollars from his/her account. S is the secret input and O is the public output of this program. In [23], S is taken to be from the range $[0, 20)$. In our tool, we specify the example as (as usual we have to set S to false on exit):

```
if (S>=20)
 {out_of_domain=true;}
else
{   O = 0;
    while(S >= 5)
    {
      S = S - 5;
      O = O + 1;
    }
 }
 S=0;
```

Now, with the new variable out_of_domain, the summary is no longer just a 0/1 ADD on the set of variables $\mathcal{G} \cup \mathcal{G}'$ but a 0/1 ADD on the set $\mathcal{G} \cup \mathcal{G}' \cup \{\mathsf{out\_of\_domain}\}$. More precisely, T_P is a function $T_P : 2^{(\mathcal{G} \cup \mathcal{G}' \cup \{\mathsf{out\_of\_domain}\})} \to \{0, 1\}$ such that for every $z_1, \ldots, z_n, z'_1, \ldots, z'_n \in \{\mathtt{true}, \mathtt{false}\}$, we have

- $T_P(z_1, \ldots, z_n, z'_1, \ldots, z'_n, false) = 1$ iff $z_1, \ldots, z_n$ is a valid input (ie, is in the domain for computation of leakage), the program P terminates on $z_1 \ldots, z_n$ and outputs $z'_1, \ldots, z'_n$.
- $T_P(z_1, \ldots, z_n, z'_1, \ldots, z'_n, true) = 1$ iff $z_1, \ldots, z_n$ is an invalid input to P (and P terminates on $z_1, \ldots, z_n$). Please note that we shall assume that on invalid inputs the program always terminates. Furthermore, on an invalid input, each variable of $\mathcal{G}'$ will always have some unique value on exit (the exact value is irrelevant for our computations).

Now, the algorithm for computing information leakage with restricted domains can be easily computed. For these algorithms, we need a new ADD operation valueAbstract. If T is an ADD on $\mathcal{V}$ and x an element of $\mathcal{V}$, then valueAbstract(x, vl, T) returns the result of setting the value of the variable x to vl (vl can be $true$ or $false$). For example, if T represents the function $f : 2^{\{x_1\} \cup \{x_2, \ldots, x_n\}} \to \mathbb{R}^+$ then valueAbstract(x_1, vl, T) represents the function $f^{x_1}_{vl} : 2^{\{x_2, \ldots, x_n\}} \to \mathbb{R}^+$ where $f^{x_1}_{vl}(x_2, \ldots, x_n) = f(vl, x_2, \ldots, x_n)$.

Algorithm 3 shows the computation of min-entropy leakage. The main difference is in the first step of the computation where we use the ADD operation valueAbstract$(*)$ to compute the summary of the program on valid inputs. The other difference is in the computation of the ADD $T_{\mathsf{term},P}$ which represents all inputs (valid and invalid) on which the program terminates. Here, we have to abstract over all outputs and out_of_domain.

Algorithm 3: Symbolic computation of min-entropy leakage of a deterministic program with restricted domain

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{ME}_{\mathsf{U}}(P)$
1 begin
2     $T_{\mathsf{valid},P} \longleftarrow \mathsf{valueAbstract}(\mathsf{out\_of\_domain}, false, T_P)$
3     $T_{\mathsf{out},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}, T_{\mathsf{valid},P})$
4     $\mathsf{num}_{\mathsf{out}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}', T_{\mathsf{out},P}))$
5     $T_{\mathsf{term},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}' \cup \{\mathsf{out\_of\_domain}\}, T_P)$
6     if $isConst(T_{term,P}) = \mathtt{false}$ or $val(T_{term,P}) = 0$ then
7         $\mathsf{num}_{\mathsf{out}} \longleftarrow \mathsf{num}_{\mathsf{out}} + 1$;
8     return $\log\, num_{out}$
```

Algorithm 4 shows the computation of Shannon-entropy leakage. As in the case of min-entropy, we have to compute the summary on valid inputs as well as compute the ADD $T_{\mathsf{term},P}$ which represents all inputs (valid and invalid) on which the program terminates. In addition, we have to also count the number of valid inputs. The latter is achieved in steps 10 –13. In order to compute the number of valid inputs, we first count the number of invalid inputs. This can be done by first computing the program summary on invalid inputs (the ADD $T_{\mathsf{invalid},P}$). Then we compute the ADD representing all invalid inputs $T_{\mathsf{invalidinp},P}$ which is then used to count the number of invalid inputs in step 12.

Algorithm 4: Symbolic computation of Shannon entropy leakage of a deterministic program with restricted domain

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{SE}_{\mathsf{U}}(P)$
1  Let $n$ be the number of variables in $\mathcal{G}$.
2  begin
3      $T_{\mathsf{valid},P} \longleftarrow \mathsf{valueAbstract}(\mathsf{out\_of\_domain}, false, T_P)$
4      $T_{\mathsf{eq\text{-}size},P} \longleftarrow \mathsf{abstract}(+, \mathcal{G}, T_{\mathsf{valid},P})$
5      $\mathsf{sum} \longleftarrow \mathsf{val}(\mathsf{abstract}(\star, \mathcal{G}', T_{\mathsf{eq\text{-}size},P}))$
6      $T_{\mathsf{term},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}' \cup \{\mathsf{out\_of\_domain}\}, T_P)$
7      $T_{\mathsf{non\text{-}term},P} \longleftarrow \mathsf{cmpl}(T_{\mathsf{term},P})$
8      $\mathsf{num}_{\mathsf{non\text{-}term}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\mathsf{non\text{-}term},P}))$
9      $\mathsf{sum} \longleftarrow \mathsf{sum} + \mathsf{num}_{\mathsf{non\text{-}term}} \log(\mathsf{num}_{\mathsf{non\text{-}term}})$
10     $T_{\mathsf{invalid},P} \longleftarrow \mathsf{valueAbstract}(\mathsf{out\_of\_domain}, true, T_P)$
11     $T_{\mathsf{invalidinp},P} \longleftarrow \mathsf{orAbstract}(\mathcal{G}', T_{\mathsf{invalid},P})$
12     $\mathsf{num}_{\mathsf{invalid}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\mathsf{invalidinp},P})$
13     $\mathsf{num}_{\mathsf{valid}} \longleftarrow 2^n - \mathsf{num}_{\mathsf{invalid}}$
14     return $((\log num_{valid}) - \frac{sum}{num_{valid}})$
```

D Further discussion of experiments

Implicit flow. Implicit flow sets O to S if S is $<= 6$; otherwise it sets O to 0.

```
O = 0;
if (S == 0) then O = 0;
else if (S == 1) then O = 1;
else if (S == 2) then O = 2;
else if (S == 3) then O = 3;
else if (S == 4) then O = 4;
else if (S == 5) then O = 5;
else if (S == 6) then O = 6;
else O = 0;
```

As all variables are compared to constants and assigned to constants, the default order works quite well for the example. Furthermore, the `Moped` can handle large integer sizes (upto 30 bits) directly for this example. We have shown the results when we the integer sizes are set to a) 15 bits and b) 30 bits in Table 1. In order to handle 32 bits, we use the Boolean encoding as `Moped` does not handle 32 bit integers directly.

Ten random outputs. This is actually a family of programs, each of which has exactly 10 outputs. A

```
if (S == r1) O = r1;
else if (S == r2) O = r2;
else if (S == r3) O = r3;
...
else if (S = r9) O = r9;
else O = r10;
```

As in [30], we generate 20 programs in this family by generating these 10 outputs from a uniform distribution. In our tests, we use 30 bits for the integers as `Moped` can handle these directly without encoding them as Boolean programs. Each of our program did result in distinct values of $r1, r2, \ldots, r10$ which means that the information leakage is independent of the program generated. We report the average time taken by our tool in Table 1. All values were within 3ms of the reported average. We did not construct the 32 bit version of the family, but our conjecture is that the tool will be to handle 32 bits easily, based on our experience with Implicit flow.

E Leakage in probabilistic programs

Min-entropy based leakage. Figure 5 gives the algorithm for computing the min-entropy of probabilistic programs.

Algorithm 5: Symbolic computation of min-entropy leakage of a probabilistic program

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{ME}_{\mathsf{U}}(P)$
1 begin
2     $T_{\mathsf{out},P} \longleftarrow \mathsf{abstract}(\max, \mathcal{G}, T_P)$
3     $\mathsf{sum}_{\mathsf{out}} \longleftarrow \mathsf{val}(\mathsf{abstract}(+, \mathcal{G}', T_{\mathsf{out},P}))$
4     $T_{\mathsf{term},P} \longleftarrow \mathsf{abstract}(+, \mathcal{G}', T_P)$
5     $\mathsf{sum}_{\mathsf{out}} \longleftarrow \mathsf{sum}_{\mathsf{out}} + (1 - \mathsf{val}(\mathsf{abstract}(\min, \mathcal{G}', T_{\mathsf{out},P})));$
6     return $\log sum_{out}$
```

Observe that in the figure, the ADD $T_{\mathsf{out},P}$ is such that $T_{\mathsf{out},P}(z'_1, \ldots, z'_n)$ is the maximum over all possible inputs $(z_1, \ldots, z_n) \in 2^{\mathcal{G}}$ of the conditional probabilities that the programs outputs $(z'_1, \ldots, z'_n) \in 2^{\mathcal{G}'}$. The ADD $T_{\mathsf{term},P}(z_1, \cdots, z_n)$ represents that the probability that the program terminates on input $(z_1, \cdots, z_n)$. From these observations, it is easy to see that the algorithm outputs $\mathrm{ME}_{\mathsf{U}}(P)$.

Shannon-entropy based leakage. Recall that the amount of information leaked by probabilistic programs is the mutual information between the output and input distribution.

The latter is difference between the entropy of outputs and the conditional Shannon entropy of the outputs given the inputs. Figure 6 gives the symbolic algorithm for computing the Shannon entropy of a probabilistic program P.

Algorithm 6: Symbolic computation of Shannon entropy leakage of a probabilistic program

```
Input: $\mathcal{G}, \mathcal{G}'$ and $T_P$ the summary of $P$.
Output: $\mathrm{SE}_{\mathsf{U}}(P)$
1  Let $n$ be the number of variables in $\mathcal{G}$.
2  begin
3      $T_{\mathsf{norm\text{-}eq\text{-}size},P} \longleftarrow \mathsf{divide}(\mathsf{abstract}(+, \mathcal{G}, T_P), 2^n)$
4      $\mathsf{val}_{\mathsf{out}} \longleftarrow (\text{-}\ \mathsf{val}(\mathsf{abstract}(\star, \mathcal{G}', T_{\mathsf{norm\text{-}eq\text{-}size},P})))$
5      $T_{\mathsf{term},P} \longleftarrow \mathsf{abstract}(+, \mathcal{G}', T_P)$
6      $\mathsf{prob}_{\mathsf{out,non\text{-}term}} \longleftarrow (1 - \frac{\mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\mathsf{term},P}))}{2^n})$
7      $\mathsf{val}_{\mathsf{out,non\text{-}term}} \longleftarrow (\text{-}\ \mathsf{prob}_{\mathsf{out,non\text{-}term}} \log \mathsf{prob}_{\mathsf{out,non\text{-}term}})$
8      $T_{\mathsf{norm\text{-}\star out},P} \longleftarrow \mathsf{divide}(\mathsf{abstract}(\star, \mathcal{G}', T_P), 2^n)$
9      $\mathsf{val}_{\mathsf{cond}} \longleftarrow (\text{-}\mathsf{val}(\mathsf{abstract}(+, \mathcal{G}, T_{\star\mathsf{out},P})))$
10     $T_{\mathsf{non\text{-}term},P} \longleftarrow \mathsf{subtract}(1, T_{\mathsf{term},P})$
11     $\mathsf{val}_{\mathsf{cond,non\text{-}term}} \longleftarrow (\text{-}\frac{\mathsf{val}(\mathsf{abstract}(\star, \mathcal{G}, T_{\mathsf{non\text{-}term\text{-}prob},P}))}{2^n})$
12     return $(val_{out} + val_{out,non\text{-}term} - val_{cond} - val_{cond,non\text{-}term})$
```

Observe that the ADD $T_{\mathsf{norm\text{-}eq\text{-}size},P}$ is a function that for each output $(z'_1, \ldots, z'_n) \in 2^{\mathcal{G}'}$ gives the probability with which the program P outputs $(z'_1, \ldots, z'_n)$ (assuming that all inputs $(z_1, \ldots, z_n) \in 2^{\mathcal{G}}$ are uniformly distributed). Also observe that $\mathsf{prob}_{\mathsf{out,non\text{-}term}}$ gives the total probability with which the prorgam does not terminate. Hence, it is clear that the sum $\mathsf{val}_{\mathsf{out}} + \mathsf{val}_{\mathsf{out,non\text{-}term}}$ gives the Shannon entropy of the all outputs (counting non-termination as a distinct output). Similarly, $\mathsf{val}_{\mathsf{cond}} + \mathsf{val}_{\mathsf{cond,non\text{-}term}}$ equals the conditional Shannon entropy of the outputs given the inputs. The difference of the above two terms gives the leakage value based on Shannon entropy.

F Dining cryptographer's problem

An interesting example of a probabilistic program is the Dining Cryptographers Problem [11] that forms the basis for anonymity systems such as DC-nets. The problem involves 3 cryptographers who gather around a table for dinner. The waiter informs them that the meal has been paid by someone, who could be one of the cryptographers or the National Security Agency (NSA). The cryptographers respect each other's right to make an anonymous payment, but want to find out whether the NSA paid. So they decide to execute a two-stage protocol. In the first stage, every two cryptographers establish a shared one-bit secret. Each cryptographer then announces whether the bits shared with him have

- same parity, if he did not pay the bill.
- different parity, if he paid the bill.

An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is. The following is a brief outline code for the protocol. (The symbol `^^` is the binary XOR operator, and the symbol `!` is the logical-not operator)

```
sh12 = rand({true,false});
sh23 = rand({true,false});
sh31 = rand({true,false});

if (CG1) annc1 = !(sh12^^sh31); else annc1 = sh12^^sh31;
if (CG2) annc2 = !(sh23^^sh12); else annc2 = sh23^^sh12;
if (CG3) annc3 = !(sh31^^sh23); else annc3 = sh31^^sh23;

NSA = !(annc1^^annc2^^annc3);
```

The bit $\mathtt{sh}_{ij}$ is a private bit shared between the i^{th} and the j^{th} cryptographer. $\mathtt{CG}_i$ is a private bit and is true when the i^{th} cryptographer paid the bill. The bit $\mathtt{annc}_i$ is a public bit announced by the i^{th} cryprographer. `NSA` is a public bit representing the outcome of the protocol.

Observe that the protocol ensures that if the bits $\mathtt{annc}_i$ are not counted in the output (this can be done by encoding them as local variables), then the value of variable NSA reveals whether NSA paid or one of the cryptographers paid the bill. Now, leakage based on min-entropy will count it as a 1 bit of leakage and leakage based on Shannon entropy (exact value if 0.81) will count it as less that 1 bit of leakage as the former happens only with probability 1/4. This amount is irrespective of the underlying distribution of the `rand` function. If, however, the bits $\mathtt{annc}_i$ are also counted in the output, then the entropy values increase if `rand` is not a uniform distribution. For example, if `rand` always turns up heads with probability 0.75 leakage based on min-entropy is 1.32193 and leakage based on Shannon entropy is 0.889093. If `rand` always turns up `true` then the value of $\mathtt{annc}_i$ will reveal if cryptographer i paid or not (and the leakage is always 2 regardless of being measured using min-entropy or Shannon entropy). We have verified this observation in the tool **`Moped-QLeak`**. The time taken to calculate the entropy values is less than 0.009 seconds.