Quantifying Information Leakage of Probabilistic Programs Using the PRISM Model Checker

Khayyam Salehi
Dept. of Computer Science
Shahrekord University
Shahrekord, Iran
email: kh.salehi@sku.ac.ir

Ali A. Noroozi
Dept. of Computer Science
University of Tabriz
Tabriz, Iran
email: noroozi@tabrizu.ac.ir

Sepehr Amir-Mohammadian
Dept. of Computer Science
University of the Pacific
Stockton, CA, USA
email: samirmohammadian@pacific.edu

***Abstract*—Information leakage is the flow of information from secret inputs of a program to its public outputs. One effective approach to identify information leakage and potentially preserve the confidentiality of a program is to quantify the flow of information that is associated with the execution of that program, and check whether this value meets predefined thresholds. For example, the program may be considered insecure, if this quantified value is higher than the threshold. In this paper, an automated method is proposed to compute the information leakage of probabilistic programs. We use Markov chains to model these programs, and reduce the problem of measuring the information leakage to the problem of computing the joint probabilities of secrets and public outputs. The proposed method traverses the Markov chain to find the secret inputs and the public outputs and subsequently, calculate the joint probabilities. The method has been implemented into a tool called PRISM-Leak, which uses the PRISM model checker to build the Markov chain of input programs. The applicability of the proposed method is highlighted by analyzing a probabilistic protocol and quantifying its leakage.**

***Index Terms*—Information leakage; Quantitative information flow; Confidentiality; PRISM-Leak.**

I. Introduction

Confidentiality is a major concern in cybersecurity that deals with protecting potentially sensitive data against illegitimate disclosure. Considering different application domains, secret data may range over different kinds of information, for instance, medical records in healthcare systems, financial records in banking systems, and passwords and other factors being used in authentication systems. Disclosure of sensitive data to low-confidentiality users has been identified as one of the common weaknesses in system deployment [1], and Open Web Application Security Project has identified it as one of the top ten privacy risks with "very high impact" [2].

Upon executing a program, a low-confidentiality user, henceforth called an attacker, may gain insight into the program secret data by observing its public outputs. This is known as *information leakage*. For example, assume that `h` is a 4-bit secret variable and `l` is a publicly available data container, i.e., it can be freely read by the attacker. Then, in the program `l := h |` $(1100)_b$, the attacker can infer the two rightmost bits of `h` by observing `l`.

A widely-studied formalism to avoid these leakages is noninterference [3][4]. It enforces the policy that no output should be affected by secret inputs. Although this ensures the security of programs by capturing all explicit and implicit flows, it is too restrictive in at least two respects: (1) Noninterference is a hyperproperty [5], and thus only applicable in meta-level analysis of programs, i.e., it cannot be enforced at runtime. To overcome this in practice, flow analysis is restricted to explicit flows only, e.g., through taint trackers [6][7]; (2) Noninterference is too conservative in many application domains by labeling many intuitively secure programs as insecure. For example, the password-checking program `if user-input = password then success else failure fi` leaks information about what `password` is not when the user cannot login, and hence, it does not satisfy noninterference. This is while, for most applications an acceptable amount of leakage can be tolerated. This limitation can be addressed by quantifying the amount of leakage and considering the ones lower than a predefined threshold as secure, instead of enforcing a no-leakage policy. Quantifying information leakage has been widely used in different realms of cybersecurity, e.g., differential privacy [8][9], the analysis of OpenSSL Heartbleed vulnerability [10], and the evaluation of cryptographic algorithms [11].

This work aligns with the second aforementioned issue of noninterference and in particular, focuses on probabilistic programs, i.e., programs that exhibit probabilistic characteristics. These characteristics are required for modeling systems in different application domains, including randomized and distributed algorithms, unreliable and unpredictable system behaviors, and model-based performance evaluations [12].

Consider a basic scenario in which the program has a secret input `h` and a public output `l`. The attacker has an *initial uncertainty* about `h` and might infer some information after running the program and observing `l`. In this case, the attacker's *remaining uncertainty* is reduced and the difference between the initial uncertainty and the remaining uncertainty is equal to the amount of leaked information. Information theory suggests entropy, e.g., *Shannon entropy* [13], as a solution to quantify uncertainty [14].

Several methods have been proposed to quantify the information leakage of various programs. For example, Klebanov [15] uses symbolic execution besides self-composition to manually compute the leakage of deterministic programs. Biondi et al. [16] develop a tool, HyLeak, for estimating the leakage of simple imperative programs. The method proposed in our work is fully-automated and computes the exact value for the leakage. Noroozi et al. [17] use model checking to compute the leakage of multi-threaded programs. They consider two assumptions, which are required to measure the leakage of concurrent programs: the attacker can select a scheduler and observe intermediate values of the public variable. Since we focus on sequential programs, there is no scheduler and the attacker can only observe final values of the public variable. This is the case in many information flow methods that analyze sequential programs [15][18]–[21].

A. Security and threat model

Any terminating sequential program exhibiting probabilistic characteristics is the subject of our study. These programs may include data associated with different levels of confidentiality, as well as zero or more neutral components. We assume the existence of at least two levels of confidentiality: secret and public. Neutral data specify temporary and/or auxiliary components of the runtime program configuration that are not assigned to a certain confidentiality level by nature, e.g., the stack pointer and loop indexes. The secret input is fixed and does not change during program execution. This is the case in any analysis in the context of confidentiality that assumes data integrity to be out of scope, e.g., [22][23]. Furthermore, the attacker is assumed to have access to the program source code, but she cannot modify it. The secret data are received by the program as input, and thus reading the source code does not directly reveal secret data. On the other hand, the attacker can execute the program arbitrary number of times and observe the public output after execution, i.e., the attacker does not have access to intermediary values, e.g., through debugging the code.

B. An illustrative example

In what follows, we describe an illustrative simple example. We will come back to this example in later sections, to explain different aspects of our formal study. Consider the following program:

```
while l1 < h mod 2 do
    l1 := l1 + 1;
    l2 := random(2);
od                                    (P1)
```

Let us assume that h is a secret variable, l_1 and l_2 are public variables with initial values set to 0, and random(2) produces 0 or 1 randomly. After executing the program, the attacker can infer information about h by observing the final value of l_1. In this paper, we are attempting to propose a method that can measure the amount of leaked information from h to l_1. As mentioned earlier, the quantification of the leakage implies a more flexible and granular security policy enforcement.

C. Contributions

The contributions of this work are as follows:

1) We propose a novel automated method for computing the information leakage of sequential programs with probabilistic characteristics. We model operational semantics of the programs by Markov chains, in the same style as [12][17]. The proposed method explores the Markov chain in a depth-first manner and finds all possible paths, from which it computes joint probabilities of the program's secrets and public outputs. It then calculates the exact value of information leakage using these joint probabilities.
2) The method has been implemented into a tool, called PRISM-Leak [24]. Input programs of PRISM-Leak are written in the PRISM language [25]. PRISM-Leak constructs the Markov chain of the input program using the PRISM model checker [25]. PRISM is a well-established tool for formal modeling and analysis of programs with probabilistic characteristics. It has been used to analyze a wide range of algorithms, protocols, and systems in various application domains such as cybersecurity, computer networking, biology, game theory, etc.
3) Finally, we demonstrate the applicability of our proposed method in a case study by analyzing the grades protocol [26]. This opens the path for evaluating confidentiality of real-world security protocols.

D. Paper outline

The paper proceeds as follows. Section II provides preliminaries of the paper, including formal definition of Markov chain and how we use it to model operational semantics of probabilistic programs. In Section III, the proposed method for computing the information leakage is discussed. Implementation and the case study are discussed in Section IV. Section V reviews related work. Finally, Section VI concludes the paper and discusses future work.

II. Basic Definitions

Let $\mathcal{X}$ be a random variable. A probability distribution Pr of random variable $\mathcal{X}$ is a function $Pr : \mathcal{X} \mapsto [0,1]$, such that $\sum_{x \in \mathcal{X}} Pr(x) = 1$.

A well-established measure to compute uncertainty of a random variable is *Shannon entropy*, which is the average number of bits required to predict a value, considered in the distribution of the random variable.

Definition 1 (Shannon entropy): The *Shannon entropy* of a random variable $\mathcal{X}$ is defined as $\mathcal{H}(\mathcal{X}) = -\sum_{x \in \mathcal{X}} Pr(\mathcal{X} = x).\log_2 Pr(\mathcal{X} = x)$.

We use Markov chains to model operational semantics of probabilistic programs. In what follows, we define Markov chains abstractly. In Section III, we instantiate them for probabilistic programs.

Definition 2 (Markov chain): A (discrete-time) Markov chain (MC) is a tuple $\mathcal{M} = (S, \mathbf{P}, \zeta)$, where

- S is a set of states,
- $\mathbf{P} : S \times S \mapsto [0,1]$ is a transition probability function such that for all $s \in S$, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$, and
- $\zeta : S \mapsto [0,1]$ is the initial distribution of states, i.e., $\sum_{s \in S} \zeta(s) = 1$.

An MC is called finite if S is finite. A state s contains the values of variables (secret, public, and neutral) as well as the program counter in each execution of the program. Given states s and s', the function $\mathbf{P}$ defines the probability $\mathbf{P}(s, s')$ of moving from s to s' in one step. ζ specifies the likelihood of being an initial state of the program. The set of initial states of Markov chain $\mathcal{M}$ is indicated by $Init(\mathcal{M})$, i.e., $Init(\mathcal{M}) = \{s \in S : \zeta(s) > 0\}$. The set of posterior states of each state is defined as $Post(s) = \{s' \in S : \mathbf{P}(s, s') > 0\}$. A state s is terminating if $Post(s) = \emptyset$. A path π in $\mathcal{M}$ is defined as a sequence of states $s_0 s_1 \dots s_n$, in which s_0 is an initial state, s_n is a terminating state, and $s_{i+1} \in Post(s_i)$ for $i \in \{0, 1, \dots, n-1\}$. The occurrence probability of π is defined as

$$Pr(\pi = s_0 s_1 \dots s_n) = \begin{cases} \zeta(s_0) & \text{if } n = 0, \\ \zeta(s_0). \prod\limits_{0 \le i < n} \mathbf{P}(s_i, s_{i+1}) & \text{otherwise.} \end{cases}$$

III. Computing the Information Leakage

In this section, we show how to compute the final leakage of probabilistic programs. Let P be a terminating probabilistic program, with a random secret variable h, a public variable l and possibly some neutral variables. For the cases where there are more than one secret variable, we concatenate them to form a single secret tuple. The same is done for public and neutral variables. This way, we simplify the formal analysis and only track the flow of a single secret data structure to a single public output channel. This results in quantifying the aggregate amount of flow from secrets to public outputs. Indeed, quantification of individual flows in the presence of multiple secrets and public outputs is feasible in our framework by revising the confidentiality labels that are assigned to different variables. For instance, one may only tag single input $\mathtt{h}_i$ as secret and the remaining inputs as neutral to solely study the flow of ith input to the public domain.

We model program P with a Markov chain $\mathcal{M} = (S, \mathbf{P}, \zeta)$. Each state $s \in S$ is a tuple $\langle \overline{l}, \overline{h}, \overline{n}, pc \rangle$, where $\overline{l}$, $\overline{h}$, and $\overline{n}$ are values of the public, secret, and neutral variables, respectively, and pc is the program counter. The transition probability function $\mathbf{P}$ defines probabilities of transitions between states. ζ is determined by $\zeta(s_0) = Pr(h = \overline{h})$ for each initial state s_0 and $s_0 = \langle \cdot, \overline{h}, \cdot, 0 \rangle$. Therefore, the definition of ζ captures the attacker's knowledge about program secrets.

When constructing $\mathcal{M}$ for P, loops of P are unfolded and considering that P is terminating, $\mathcal{M}$ becomes a directed acyclic graph (DAG). Initial states are roots and terminating states are leaves of each DAG. In the following example, we review the MC of program `P1`.

Example 1: MC of `P1`. The MC of `P1` is depicted in Figure 1, where `h` is a 2-bit value and thus either 0, 1, 2, or 3. For the sake of brevity, pc is not shown in the graph. Moreover, there are not any neutral values in this simple example. In each state, `l` is defined as $\langle \mathtt{l}_1, \mathtt{l}_2 \rangle$. Note that branches are due to assigning a random value (0 or 1) to $\mathtt{l}_2$.

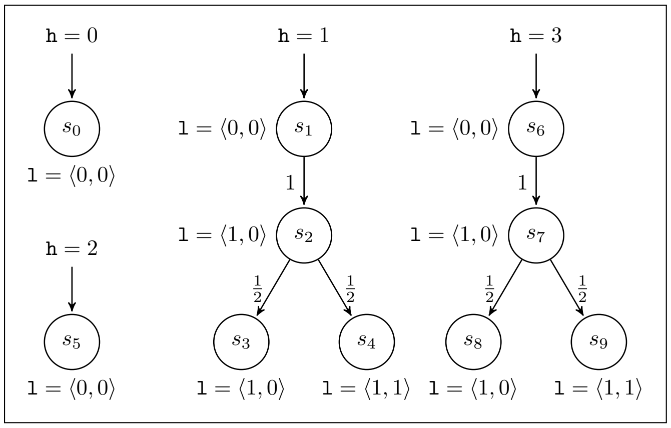

Figure 1. MC of the program `P1`.

The attacker runs the program and observes the public outputs. The public outputs are the values of `l` in terminating states and denoted by o. The prior distribution $Pr(h)$ specifies the initial uncertainty of the attacker and the posterior distribution $Pr(h \mid o)$ specifies the remaining uncertainty of the attacker, which is obtained after running the program and observing the output o. Therefore, the final leakage of $\mathcal{M}$ is computed as

$$\mathcal{L}(\mathcal{M}) = \mathcal{H}(h) - \mathcal{H}(h \mid o). \tag{1}$$

In (1), $\mathcal{H}(h)$ is the initial uncertainty and computed as

$$\mathcal{H}(h) = -\sum_{\overline{h} \in h} Pr(h = \overline{h}). \log_2 Pr(h = \overline{h}).$$

$\mathcal{H}(h \mid o)$ is the remaining uncertainty in (1) and calculated as

$$\mathcal{H}(h \mid o) = \sum_{\overline{o} \in o} Pr(o = \overline{o}). \mathcal{H}(h \mid o = \overline{o}). \tag{2}$$

In (2), $\mathcal{H}(h \mid o = \overline{o})$ is defined as

$$\mathcal{H}(h \mid o = \overline{o}) = \\ -\sum_{\overline{h} \in h} Pr(h = \overline{h} \mid o = \overline{o}). \log_2 Pr(h = \overline{h} \mid o = \overline{o}),$$

and $Pr(h = \overline{h} \mid o = \overline{o})$ is computed by

$$Pr(h = \overline{h} \mid o = \overline{o}) = \frac{Pr(h = \overline{h}, o = \overline{o})}{Pr(o = \overline{o})}.$$

$Pr(h = \overline{h}, o = \overline{o})$ is the joint probability of $h = \overline{h}$ and $o = \overline{o}$. $Pr(o = \overline{o})$ is the occurrence probability of the output $\overline{o}$ and is computed as

$$Pr(o = \overline{o}) = \sum_{\overline{h} \in h} Pr(h = \overline{h}, o = \overline{o}).$$

Thus, computing the remaining uncertainty is reduced to computing the joint probabilities $Pr(h, o)$. Assuming we have all paths of $\mathcal{M}$ and their probabilities, the joint probability $Pr(h = \overline{h}, o = \overline{o})$ can be calculated as the sum of the

$$Pr(h=0, o=\langle 0,0\rangle) = Pr(\pi = s_0) = 1/4$$
$$Pr(h=1, o=\langle 1,0\rangle) = Pr(\pi = s_1 s_2 s_3) = 1/8$$
$$Pr(h=1, o=\langle 1,1\rangle) = Pr(\pi = s_1 s_2 s_4) = 1/8$$
$$Pr(h=2, o=\langle 0,0\rangle) = Pr(\pi = s_5) = 1/4$$
$$Pr(h=3, o=\langle 1,0\rangle) = Pr(\pi = s_6 s_7 s_8) = 1/8$$
$$Pr(h=3, o=\langle 1,1\rangle) = Pr(\pi = s_6 s_7 s_9) = 1/8$$
$$Pr(o=\langle 0,0\rangle) = Pr(h=0, o=\langle 0,0\rangle) + Pr(h=2, o=\langle 0,0\rangle) = 1/2$$
$$Pr(o=\langle 1,0\rangle) = Pr(h=1, o=\langle 1,0\rangle) + Pr(h=3, o=\langle 1,0\rangle) = 1/4$$
$$Pr(o=\langle 1,1\rangle) = Pr(h=1, o=\langle 1,1\rangle) + Pr(h=3, o=\langle 1,1\rangle) = 1/4$$
$$Pr(h=0 \mid o=\langle 0,0\rangle) = 1/2, \quad Pr(h=1 \mid o=\langle 1,0\rangle) = 1/2$$
$$Pr(h=1 \mid o=\langle 1,1\rangle) = 1/2, \quad Pr(h=2 \mid o=\langle 0,0\rangle) = 1/2$$
$$Pr(h=3 \mid o=\langle 1,0\rangle) = 1/2, \quad Pr(h=3 \mid o=\langle 1,1\rangle) = 1/2$$

Figure 2. 1) Joint probabilities, $Pr(h, o)$, 2) public output occurrence probabilities, $Pr(o)$, and 3) the posterior probabilities, $Pr(h \mid o)$, in `P1`.

occurrence probabilities of all paths that lead to a terminating state $s_n = \langle \overline{o}, \overline{h}, \cdot, \cdot \rangle$, i.e.,

$$Pr(h = \overline{h}, o = \overline{o}) = \sum_{s_0 \in Init(\mathcal{M}),\ s_n = \langle \overline{o}, \overline{h}, \cdot, \cdot \rangle} Pr(\pi = s_0 \ldots s_n).$$

In the following example, we calculate the information leakage from `h` to the public domain in program `P1`.

Example 2: Information leakage in `P1`. Assume that initially the attacker only knows the bit length of `h` and thus the probability distribution of `h` becomes uniform, i.e., $Pr(h) = 1/4$ for all four possible values of `h`. Then, the initial uncertainty is computed as $\mathcal{H}(h) = -\sum_{\overline{h}=0,1,2,3} (1/4) \log_2 (1/4) = 2$. As explained earlier, in order to calculate the remaining uncertainty, we need to compute the joint probabilities $Pr(h, o)$. Using the joint probabilities, the public output occurrence probabilities $Pr(o)$ are computed, and then the posterior probabilities $Pr(h|o)$ are calculated. These details are given in Figure 2. Therefore, we would have $\mathcal{H}(h \mid o = \langle 0,0 \rangle) = \mathcal{H}(h \mid o = \langle 1,0 \rangle) = \mathcal{H}(h \mid o = \langle 1,1 \rangle) = 1$. These yield the remaining uncertainty $\mathcal{H}(h \mid o)$ to be equal to 1. Thus, the amount of leakage is calculated as $\mathcal{L} = \mathcal{H}(h) - \mathcal{H}(h \mid o) = 2 - 1 = 1\ bit$. This is in compliance with the intuition that the attacker infers the least significant bit of the secret.

Figure 3 shows the detailed steps of computing $Pr(h, o)$ for the Markov chain $\mathcal{M}$. The algorithm uses a higher-order map function $ohMap : \overline{o} \mapsto (\overline{h} \mapsto Pr(h = \overline{h}, o = \overline{o}))$ to store the joint probabilities. It traverses the Markov chain $\mathcal{M}$ by a depth-first recursive function, called EXPLOREPATHS$(\cdot)$, and extracts all paths. It then calculates $Pr(h, o)$.

Time complexity. The costs of computing the information leakage are dominated by the costs of computing the joint probabilities in the algorithm shown in Figure 3. The core of the algorithm is to find all paths of $\mathcal{M}$ using depth-first exploration. $\mathcal{M}$ is a DAG and the number of all possible paths of a DAG can be exponential in the number of its states. Therefore, computing the leakage of $\mathcal{M}$ takes $O(2^n)$ time in the worst case, where n is the number of states of $\mathcal{M}$. It should be noted that this is the expected time complexity for model checking algorithms, as they analyze the whole state space [12]. Furthermore, the method is used for a limited number of times to analyze the security of a program.

```
    Input: finite MC $\mathcal{M}$
    Output: a map containing the joint probabilities $Pr(h, o)$
 1: Let $ohMap$ be an empty higher-order map function from
    $\overline{o}$ to $\overline{h}$ to $Pr(h = \overline{h}, o = \overline{o})$;
    // i.e. $ohMap : \overline{o} \mapsto (\overline{h} \mapsto Pr(h = \overline{h}, o = \overline{o}))$
 2: Let $\pi$ be an empty list of states for storing a path;
 3: for $s_0$ in $Init(\mathcal{M})$ do
 4:     EXPLOREPATHS($s_0$, $\pi$, $ohMap$);
 5: return $ohMap$;

 6: function EXPLOREPATHS($s$, $\pi$, $ohMap$)
        // add state s to the current path from the initial state
 7:     $\pi$.add($s$);
        // found a path stored in $\pi$
 8:     if $s$ is a terminating state then
 9:         // assume $s = \langle \overline{o}, \overline{h}, \cdot, \cdot \rangle$
            // define $hMap$ as $Pr(h, o = \overline{o})$
10:         if $\overline{o}$ not in $ohMap$ then
11:             Let $hMap$ be an empty map from
                    $\overline{h}$ to $Pr(h = \overline{h}, o = \overline{o})$;
12:         else
13:             $hMap = ohMap.get(\overline{o})$;
14:         if $\overline{h}$ not in $hMap$ then
15:             $prob = Pr(\pi)$;
16:         else
17:             $prob = Pr(\pi) + hMap.get(\overline{h})$;
18:         $hMap.put(\overline{h}, prob)$;  // Update $hMap$
19:         $ohMap.put(\overline{o}, hMap)$;  // Update $ohMap$
20:     else
21:         for $s'$ in $Post(s)$ do
22:             EXPLOREPATHS($s'$, $\pi$, $ohMap$);
        // done exploring from $s$, so remove it from $\pi$
23:     $\pi$.pop();
24:     return ;
```

Figure 3. Computing the joint probabilities $Pr(h, o)$.

IV. IMPLEMENTATION AND CASE STUDY

In this section, we describe an implementation of our proposed algorithm, employing the PRISM model checker. Next, as a case study we study the information leakage in an example protocol.

A. PRISM-Leak: An information leakage quantifier

An efficient implementation of the method requires a model checker to construct the Markov chain of the input program. We have implemented the approach as part of PRISM-

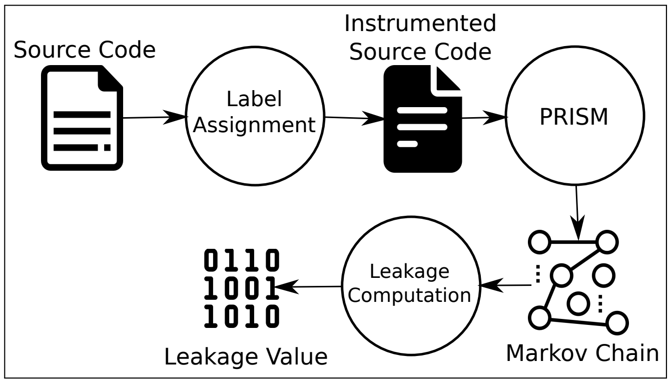

Figure 4. Architecture of PRISM-Leak.

Leak [24]. At a high level, the architecture of PRISM-Leak is depicted in Figure 4. Source code is in the PRISM language and label assignment tags program variables with the public and secret labels. The PRISM model checker builds the Markov chain and stores it via multi-terminal binary decision diagrams. These decision diagrams are efficient symbolic data structures to store states and transitions of Markovian models [27]. PRISM-Leak uses these diagrams to extract the set of reachable states and builds a sparse matrix containing the transitions between the states. Then, it finds the outputs by traversing the model, computes the joint probabilities of the secrets and the public outputs according to the algorithm shown in Figure 3, and employs them to measure the amount of the final leakage.

B. Case study

In order to evaluate the applicability of the proposed method, we consider the grades protocol [26] as a case study and show how the method computes the leakage of probabilistic programs. In the grades protocol, k students $s_1, \ldots, s_k$ are given secret grades $g_1, \ldots, g_k$, where $0 \leq g_i < m$. The students aim to compute the sum of their grades, without revealing their secret grade to other students. For that, each student s_i produces a random number r_i between 0 and $n = (m-1) \times k + 1$ and announces it only to the student $s_{(i-1)\%k}$. Then, the student s_i declares a number $d_i = g_i + r_i - r_{(i+1)\%k}$. The sum of all grades is equivalent to $\left(\sum_i d_i\right) \ \% \ n$. We assume the grades are secret, and the declarations and the sum are public. To evaluate security of the protocol, we consider two cases: 1) the attacker knows the declarations and the sum of the grades, and 2) the attacker only knows the sum. If the amount of leakage is the same for both cases, then the protocol does not leak secret information via the declarations.

Table I reports the amounts of leakage, as well as the number of states and transitions of the Markov chains for the two aforementioned cases. As seen in the table, both leakages are identical and thus, the protocol is secure, i.e., an attacker that knows both the declarations and the sum of the grades gains the same information as an attacker that only knows the sum. PRISM source code of the protocol is available at the Github repository of PRSIM-Leak [24].

V. Related Work

In this section, we discuss the related work and compare them to ours.

Backes et al. [28] propose an automated method to compute information leakage. They employ the ARMC model checker to extract the equivalence relation of high values which have the same output. They enumerate the size of each equivalence class using the omega-calculator and LATTE (Lattice point Enumeration). They only consider deterministic programs. In this respect, our work covers probabilistic programs, as well.

Chothia et al. [29] propose a framework to quantify the information leakage in every two arbitrary points of a program. They extend their method to consider Java programs by developing LeakWatch [30]. LeakWatch can estimate the leakage using statistical approximation techniques. It also considers intermediate leakages. Our proposed method calculates the exact values and does not consider intermediate leakages.

Klebanov [15] uses symbolic execution besides self-composition to precisely compute the information leakage of deterministic programs. Although his method is precise, it is not automated and requires manual effort. On the other hand our work proposes an automated method.

Biondi et al. [16] develop HyLeak, a tool for measuring the leakage of simple imperative programs. They use a combination of stochastic program simulations and precise methods to calculate an estimated joint probability distribution of secrets and outputs. In contrast, we take a precise approach in calculating the joint probability distribution, which results in exact information leakage values.

Pardo et al. [21] develop PRIVUG, which quantifies the leakage of programs written in Java, Scala, and Python. This tool estimates the leakage and does not compute the exact value.

Salehi et al. [31] utilize an evolutionary algorithm to compute channel capacity of concurrent probabilistic programs. Channel capacity concerns with the maximum amount of leakage that an attacker can learn from a program. They employ their method to compute the leakage values of two anonymity protocols, the dining cryptographers and the single preference protocols.

In addition to the proposed method of this paper, PRISM-Leak contains other methods: 1) a quantitative method [17] which employs a trace-based approach, considering scheduler effect and intermediate leakages, to compute various types of information leakage for concurrent programs; and 2) a qualitative method [32] that checks satisfiability of observational determinism, in order to enforce no-leakage policy. This policy is too restrictive for most applications, as there could be some tolerable amount of leakage in these applications [14].

VI. Conclusion and future work

We have presented an automated method to measure the information leakage of probabilistic programs. The method uses the PRISM model checker to build Markov chain of

TABLE I. LEAKAGES OF THE GRADES PROTOCOL AND THE SUM OF THE GRADES

m	k	The grades protocol			The sum of the grades		
		$\mathcal{M}_{grades}$		Leakage (bits)	$\mathcal{M}_{sum}$		Leakage (bits)
		# states	# transitions		# states	# transitions	
2	2	196	228	1.5 (75%)	16	20	1.5
	3	3752	4256	1.81 (60.4%)	64	104	1.81
	4	92496	102480	2.03 (50.8%)	256	528	2.03
3	2	1179	1395	2.2 (69.3%)	36	45	2.2
	3	66366	75600	2.53 (53.1%)	216	351	2.53
	4	439668	597780	2.75 (43.3%)	1296	2673	2.75
4	2	4048	4816	2.66 (66.4%)	64	80	2.66
	3	455104	519040	2.98 (49.7%)	512	832	2.98
	4	3271680	6589440	3.2 (40%)	4096	8448	3.2

the programs. The implementation of the method, PRISM-Leak, extracts states and transitions of this Markov chain, finds secrets and outputs, and computes the information leakage. Finally, we have analyzed a case study to show how the proposed method can evaluate the security of probabilistic programs.

As future work, we aim to compare scalability of the proposed method to other leakage quantification methods, some of which are explored in related work. We also aim to incorporate statistical methods to approximate leakage. This can improve the scalability of the method.

In this paper, we only considered terminating programs. As future work, we are planning to work on a method for computing leakage of non-terminating programs. We also aim to extend the proposed method in order to analyze case studies in other application domains, such as cryptographic algorithms.

References

[1] "CWE-200: Exposure of Sensitive Information to an Unauthorized Actor," https://rb.gy/ac6ui0, [retrieved: 10, 2021].

[2] "OWASP Top 10 Privacy Risks," https://rb.gy/vhq4qj, [retrieved: 10, 2021].

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J-SAC*, vol. 21, no. 1, pp. 5–19, 2003.

[4] G. Smith, "Principles of secure information flow analysis," in *Malware Detection. Advances in Information Security, vol 27*. Springer-Verlag, 2007, pp. 291–307.

[5] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, 2010.

[6] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *EuroS&P*. IEEE, 2016, pp. 15–30.

[7] C. Skalka, S. Amir-Mohammadian, and S. Clark, "Maybe tainted data: Theory and a case study," *J. Comput. Secur.*, vol. 28, no. 3, pp. 295–335, April 2020.

[8] M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, P. Degano, and C. Palamidessi, "Differential privacy: On the trade-off between utility and information leakage," in *FAST*. Springer, 2011, pp. 39–54.

[9] P. Cuff and L. Yu, "Differential privacy as a mutual information constraint," in *CCS*, 2016, pp. 43–54.

[10] F. Biondi and et al., "Scalable approximation of quantitative information flow in programs." in *VMCAI*, 2018, pp. 71–93.

[11] M. Jurado, C. Palamidessi, and G. Smith, "A formal information-theoretic leakage analysis of order-revealing encryption," in *CSF*. IEEE Computer Society, 2021, pp. 1–16.

[12] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press Cambridge, 2008.

[13] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2006.

[14] M. S. Alvim and et al., *The Science of Quantitative Information Flow*. Springer, 2020.

[15] V. Klebanov, "Precise quantitative information flow analysis—a symbolic approach," *Theor. Comput. Sci.*, vol. 538, pp. 124–139, 2014.

[16] F. Biondi, Y. Kawamoto, A. Legay, and L.-M. Traonouez, "Hyleak: hybrid analysis tool for information leakage," in *ATVA*. Springer, 2017, pp. 156–163.

[17] A. A. Noroozi, J. Karimpour, and A. Isazadeh, "Information leakage of multi-threaded programs," *Comput. Electr. Eng.*, vol. 78, pp. 400–419, 2019.

[18] R. Chadha, U. Mathur, and S. Schwoon, "Computing information flow using symbolic model-checking," in *FSTTCS*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014, pp. 505–516.

[19] A. Weigl, "Efficient sat-based pre-image enumeration for quantitative information flow in programs," in *DPM*. Springer, 2016, pp. 51–58.

[20] M. S. Alvim and et al., "An axiomatization of information flow measures," *Theor. Comput. Sci.*, vol. 777, pp. 32–54, 2019.

[21] R. Pardo, W. Rafnsson, C. Probst, and A. Wasowski, "Privug: Quantifying leakage using probabilistic programming for privacy risk analysis," *arXiv preprint arXiv:2011.08742*, 2020.

[22] F. Biondi, A. Legay, P. Malacaria, and A. Wasowski, "Quantifying information leakage of randomized protocols," *Theor. Comput. Sci.*, vol. 597, no. C, pp. 62–87, 2015.

[23] S. Amir-Mohammadian, "A semantic framework for direct information flows in hybrid-dynamic systems," in *CPSS-AsiaCCS*. ACM, June 2021, pp. 5–15.

[24] A. A. Noroozi, K. Salehi, J. Karimpour, and A. Isazadeh, "Prism-leak - a tool for computing information leakage of probabilistic programs," https://rb.gy/elgkyi, [retrieved: 10, 2021].

[25] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*. Springer, 2011, pp. 585–591.

[26] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic bisimulation for parameterized systems," in *CAV*. Springer, 2019, pp. 455–474.

[27] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, 2002.

[28] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *S&P*. IEEE, 2009, pp. 141–153.

[29] T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker, "Probabilistic point-to-point information leakage," in *CSF*. IEEE, 2013, pp. 193–205.

[30] T. Chothia, Y. Kawamoto, and C. Novakovic, "Leakwatch: Estimating information leakage from java programs," in *ESORICS*. Springer, 2014, pp. 219–236.

[31] K. Salehi, J. Karimpour, H. Izadkhah, and A. Isazadeh, "Channel capacity of concurrent probabilistic programs," *Entropy*, vol. 21, no. 9, p. 885, 2019.

[32] A. A. Noroozi, K. Salehi, J. Karimpour, and A. Isazadeh, "Secure information flow analysis using the prism model checker," in *ICISS*. Springer, 2019, pp. 154–172.