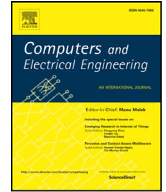
Computers and Electrical Engineering 78 (2019) 400–419

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng



Information leakage of multi-threaded programs☆

Ali A. Noroozi*, Jaber Karimpour, Ayaz Isazadeh

Department of Computer Science, University of Tabriz, Tabriz, Iran

ARTICLE INFO

Article history:
Received 3 December 2018
Revised 30 July 2019
Accepted 30 July 2019

Keywords:
Quantitative information flow
Information leakage
Multi-threaded program
Markovian processes
Confidentiality
PRISM-Leak

ABSTRACT

Quantitative information flow is an important technique for measuring information leakage of a program. It is widely used in analyzing anonymity protocols and timing channels. One area of interest is to quantify the information leakage of multi-threaded programs, which has not been well-studied in prior work. In this paper, an automated trace-based approach is proposed to precisely quantify information leakage of shared-memory multi-threaded programs. The approach takes into account the effect of schedulers and leakage in intermediate states of executions. The programs are modeled by Markovian processes. Then, variants of information leakage, including expected, bounded time, maximum, and minimum leakages are measured. The validity of the approach is demonstrated by implementing it in a tool PRISM-Leak, which is built upon PRISM, a probabilistic model checker. Finally, two case studies are utilized to analyze and compare the approach against state-of-the-art leakage quantification tools.

© 2019 Elsevier Ltd. All rights reserved.

1. Introduction

Quantitative information flow is an important field of information security which aims to compute how much information is leaked from a program's *secret inputs* into its *public outputs*. It has been successfully applied to security problems like analyzing anonymity protocols [1,2], checking whether patches to the Linux kernel effectively fix the security issues they address or not [3], and analysis of the OpenSSL Heartbleed vulnerability [4].

Assume a program with a secret input `h` and a public output `l`. When information is leaked from `h` into `l`, an *attacker* observing the program outputs is be able to deduce information about `h`. For example, the attacker can infer the whole `h` in the program `l:=h` or the least significant bit of `h` in the program `l:=h mod 2`. The amount of information deduced by the attacker is called *information leakage* [5].

Information leakage can be quantified using information theory: the attacker's *uncertainty* about `h` is reduced after observing the value of `l`. The attacker's initial uncertainty quantifies the amount of information in `h` and the attacker's final uncertainty (after program execution) quantifies the amount of unleaked information about `h`. Therefore, the amount of information leakage is computed as the difference between initial uncertainty and final uncertainty [5].

One promising approach to compute the information leakage is to use state transition systems. Recent works [1,6–10] model programs as state transition systems, with the inputs as initial states, the outputs as final states and the attacker observing the final states. They compute the leakage as the difference of uncertainty in initial states and final

☆ This paper is for regular issues of CAEE. Reviews processed and recommended for publication to the Editor-in-Chief by Area Editor Dr. Gregorio Martinez Perez.

* Corresponding author.
E-mail address: noroozi@tabrizu.ac.ir (A.A. Noroozi).

https://doi.org/10.1016/j.compeleceng.2019.07.018
0045-7906/© 2019 Elsevier Ltd. All rights reserved.

states of the state transition system. However, these works may not meet the requirements given in quantitative information flow analysis of multi-threaded programs. In particular, most of these works are neither general enough nor scalable for a thorough security analysis of multi-threaded programs. Furthermore, they do not consider leakage in intermediate states. A multi-threaded program may leak information in intermediate states of executions and hence considering only the final states is not enough [6,8,11]. For instance, consider the program

```
P1 ≡ [T1 || T2]; T3
```

where

```
T1 ≡ if l=1 then l:=h else skip fi
T2 ≡ l:=1
T3 ≡ l:=2
```

and `h` is the secret input, `l` is the public output with the initial value of 0, `||` is the parallel operator with shared variables and `;` is the sequential composition operator. Assume a scheduler which always picks `T2` first. A quantitative analysis that considers only the final states marks `P1` secure, whereas it leaks the whole secret in an intermediate state (100% leakage). Another concern is the effect of scheduler of a multi-threaded program on the amount of leakage. For instance, in `P1` (the above program) assume the attacker executes the program with a scheduler which always picks `T1` first. With this scheduler, the program is secure. However, if the attacker selects a scheduler which always picks `T2` first, the secret is leaked completely (100% leakage). This example shows that a quantitative security analysis for multi-threaded programs should be scheduler-specific and consider leakage in intermediate states.

The overall problem addressed in this paper is to develop an automated approach for measuring the amount of leakage of shared-memory multi-threaded programs, considering the effect of scheduler and leakage in intermediate states. Assume without loss of generality that the program has a secret input and a public output. For the case where there are more than one secret (or public) variable, all secret (or public) variables are concatenated to form a single secret (or public) variable. Suppose an attacker who has full knowledge of source code of the program, can choose a scheduler for the threads, run the program, possibly multiple times, with the chosen scheduler, and observe sequences of values of the public variable, i.e., execution traces. Furthermore, assume the attacker does not influence the initial value of the public variable, i.e., the program has no public input. Based on these observations and their prior knowledge about the secret input, the attacker can try to guess the value of the secret. Considering these assumptions about the attacker, the following contributions are made:

- formally modeling multi-threaded programs, running under a memoryless probabilistic scheduler, using Markovian models (Section 3.1),
- computing leakage variants of a multi-threaded program, including expected (Section 3.2), bounded time (Section 3.3.1), maximum and minimum (Section 3.4) leakages. Expected leakage is the amount of information inferred by an attacker that is able to run the program multiple times and observe all execution traces of the program. Bounded time leakage is the amount of expected leakage at a given time. Maximum and minimum leakages are upper and lower leakage bounds for an attacker with a uniform prior knowledge about the secret input. We also show how to compute the amount of leaked information from a given time to another given time (Section 3.3.1) and discuss the bounded leakage problem, i.e., computing the amount of time spent to leak a specified amount of information (Section 3.3.2),
- implementing the proposed approach in an automated tool PRISM-Leak, which is available for download at https://github.com/alianoroozi/PRISM-Leak,
- analyzing two case studies using the proposed approach (Section 4).

The paper is organized as follows: the next section provides terminology and notation. Section 3 presents the proposed approach. It starts with how to construct the program model and then discusses how to compute the leakage variants. The implementation and case studies are discussed in Section 4. Section 5 discusses some related work from the literature and compares the proposed approach with them, qualitatively and quantitatively. Finally, Section 6 concludes the paper and proposes some future work.

2. Preliminaries

2.1. Information theory

Let $\mathcal{X}$ denote a random variable with the finite set of values $Val_{\mathcal{X}}$. A *probability distribution* Pr over $Val_{\mathcal{X}}$ is a function $Pr : Val_{\mathcal{X}} \rightarrow [0, 1]$, such that $\sum_{x \in Val_{\mathcal{X}}} Pr(x) = 1$. The set of all probability distributions over $Val_{\mathcal{X}}$ is denoted by $\mathcal{D}(Val_{\mathcal{X}})$.

A widely-used measure of uncertainty is *Shannon entropy*, which is mostly used in the context of an attacker being able to attempt multiple guesses of the random variable.

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

Intuitively, Shannon entropy measures the expected number of bits required to transmit the random variable optimally [5].

The attacker might try to guess the value of the random variable in just one try. This is called one-try guessing model. In the context of this model, *Renyi's min-entropy* gives a better operational security guarantee [5].

Definition 2. The **Renyi's min-entropy** of a random variable $\mathcal{X}$ is given by $\mathcal{H}_\infty(\mathcal{X}) = -\log_2 \max_{x\in Val_{\mathcal{X}}} \; Pr(x)$.

Intuitively, min-entropy measures vulnerability of the random variable to a single guess of the attacker. On a uniform distribution, Renyi's min-entropy and Shannon entropy coincide. For instance, consider the following distribution $Pr(\mathcal{X}) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$ on the variable $\mathcal{X}$ with $Val_{\mathcal{X}} = \{0, 1, 2, 3\}$. The Renyi's min-entropy and Shannon entropy are computed as

$$\mathcal{H}_\infty(\mathcal{X}) = -\log_2 \;\max\left\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right\} = 2, \quad \mathcal{H}(\mathcal{X}) = -\left(\frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4}\right) = 2,$$

which are the same.

2.2. Markovian models and probabilistic schedulers

We need nondeterminism to model parallelism. We also need probabilistic choices to model execution probabilities and the attacker knowledge. The most widely-used state transition system containing both probabilistic and nondeterministic choices is Markov decision process (MDP). We use Markov decision processes (MDPs) to model operational semantics of multi-threaded programs. We also use memoryless probabilistic schedulers to represent the scheduling policy of multi-threaded programs. As a memoryless probabilistic scheduler is applied to an MDP, a Markov chain (MC) is induced, which is the final model used in this paper for quantifying the information leakage of multi-threaded programs.

MDP states are a mapping from variables and program counter to values. In any state of an MDP, a nondeterministic choice between probability distributions exists. Once a probability distribution has been chosen nondeterministically, the next state is selected in a probabilistic manner. Nondeterminism is used to model concurrency between threads by means of *interleaving*, i.e., all possible choices of the threads is considered. A state of a Markov decision process consists of values of the variables, together with the current value of the program counter that indicates the next program statement (command) to be executed. Suppose the program has a public output `l` and Val_l denotes the finite set of values of `l`. Formally,

Definition 3. A **Markov decision process (MDP)** is a tuple $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_l, V_l)$ where S is a set of states, Act is a set of actions, $\mathbf{P}$: $S \times Act \times S \to [0, 1]$ is a transition probability function such that $\forall s \in S.\ \forall \alpha \in Act.\ \sum_{s'\in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\}$, the function ζ: $S \to [0, 1]$ is an initial distribution such that $\sum_{s\in S} \zeta(s) = 1$, Val_l is the set of values of `l`, and V_l: $S \to Val_l$ is a labeling function.

The function V_l *labels* each state with value of the public variable in that state. In fact, a state label is what an attacker observes in that state. An MDP $\mathcal{M}$ is called *finite* if S, Act, and Val_l are finite. An action α is *enabled* in state s if and only if $\sum_{s'\in S} \mathbf{P}(s, \alpha, s') = 1$. Let $Act(s)$ denote the set of enabled actions in s. In our program model, actions represent the program statements (commands). Each state s' for which $\mathbf{P}(s, \alpha, s') > 0$ is called an α-*successor* of s. The set of α-successors of s is denoted by $Post(s, \alpha)$. The set of *successors* of s is defined as $Post(s) = \cup_{\alpha\in Act(s)} Post(s, \alpha)$.

We assume the programs always terminate and there is no deadlock in the program. To ensure $\mathcal{M}$ is non-blocking, we include a self-loop to each state s that has no successor, i.e., $\mathbf{P}(s, \tau, s) = 1$.[1] Then, a state s is called *final* if $Post(s) = \{s\}$. It is assumed that all final states correspond to the termination of the program. A state s with $\zeta(s) > 0$ is considered as an *initial state*. The set of initial states of $\mathcal{M}$ is denoted by $Init(\mathcal{M})$.

2.2.1. Semantics of MDPs

The intuitive operational behavior of an MDP $\mathcal{M}$ is as follows. At the beginning, an initial state s_0 is chosen with probability $\zeta(s_0)$. Assuming that $\mathcal{M}$ is in state s, first a nondeterministic choice between the enabled actions needs to be resolved. Suppose action $\alpha \in Act(s)$ is selected. Then, one of the α-successors of s is selected randomly according to the transition function $\mathbf{P}$. That is, with probability $\mathbf{P}(s, \alpha, s')$ the next state is s'.

An MDP with no action and nondeterminism is called a Markov Chain.

Definition 4. A **(discrete-time) Markov chain (MC)** is a tuple $\mathcal{M} = (S, \mathbf{P}, \zeta, Val_l, V_l)$ where S is a set of states, $\mathbf{P}$: $S \times S \to [0, 1]$ is a transition probability function such that for all states $s \in S$: $\sum_{s'\in S} \mathbf{P}(s, s') = 1$, ζ: $S \to [0, 1]$ is an initial distribution such that $\sum_{s\in S} \zeta(s) = 1$, Val_l is the set of values of `l`, and V_l: $S \to Val_l$ is a labeling function.

The function $\mathbf{P}$ determines for each state s the probability $\mathbf{P}(s, s')$ of a single transition from s to s'. MCs are state transition systems with probability distributions for transitions of each state. That is, the next state is chosen probabilistically, not nondeterministically.

[1] The distinguished action label τ is used to show that the self-loop's action is not of further interest.

A probabilistic scheduler implements scheduling policy of a multi-threaded program. It determines the order and probability of execution of threads. As we modeled concurrency between threads using nondeterminism in MDP, the scheduler is used to resolve the possible nondeterminism in MDP. We consider a simple but important subclass of schedulers called *memoryless probabilistic schedulers*. Given a state s, a memoryless probabilistic scheduler returns a probability for each action $\alpha \in Act(s)$. This random choice is independent of what has happened in the history, i.e., which path led to the current state. This is why it is called memoryless. Formally,

Definition 5. Let $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_l, V_l)$ be an MDP. A **memoryless probabilistic scheduler** for $\mathcal{M}$ is a function $\delta : S \to \mathcal{D}(Act)$, such that $\delta(s) \in \mathcal{D}(Act(s))$ for all $s \in S$.

As all nondeterministic choices in an MDP $\mathcal{M}$ are resolved by a scheduler δ, a Markov chain $\mathcal{M}_\delta$ is induced. Formally,

Definition 6. Let $\mathcal{M} = (S, Act, \mathbf{P}, \zeta, Val_l, V_l)$ be an MDP and $\delta : S \to \mathcal{D}(Act)$ be a memoryless probabilistic scheduler on $\mathcal{M}$. The **MC of $\mathcal{M}$ induced by δ** is given by $\mathcal{M}_\delta = (S, \mathbf{P}_\delta, \zeta, Val_l, V_l)$ where $\mathbf{P}_\delta(s, s') = \sum_{\alpha \in Act(s)} \delta(s)(\alpha).\mathbf{P}(s, \alpha, s')$.

Those states of $\mathcal{M}$ that are unreachable when the program is executed with the scheduler δ, are removed by the transition function $\mathbf{P}_\delta$. In the remaining of this section, the MC $\mathcal{M}_\delta^{\mathrm{P}} = (S, \mathbf{P}_\delta, \zeta, Val_l, V_l)$ is considered as the final model of the multi-threaded program P and the probabilistic scheduler δ.

2.2.2. Path fragments and paths

A finite path fragment $\hat{\pi}$ of $\mathcal{M}_\delta^{\mathrm{P}}$ is a finite state sequence $s_0 s_1 \ldots s_n$ such that $s_0 \in Init(\mathcal{M}_\delta^{\mathrm{P}})$ and $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$. We refer to n as the length of the path fragment $\hat{\pi}$. For $i \geq 0$, $\hat{\pi}[i]$ denotes the state at index i, i.e., $\hat{\pi}[i] = s_i$. A *path* (or execution path) π of $\mathcal{M}_\delta^{\mathrm{P}}$ is an infinite state sequence $s_0 s_1 \ldots s_{n-1} s_n^\omega$ such that $s_0 \in Init(\mathcal{M}_\delta^{\mathrm{P}})$, $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$, s_n is a final state and ω denotes infinite iteration (self-loop over s_n). The set of paths starting from an initial state s_0 is denoted by $Paths(s_0)$. The set of all paths of $\mathcal{M}_\delta^{\mathrm{P}}$ is denoted by $Paths(\mathcal{M}_\delta^{\mathrm{P}})$ and the set of path fragments of length i in $\mathcal{M}_\delta^{\mathrm{P}}$ is denoted by $Paths_{\ll i}(\mathcal{M}_\delta^{\mathrm{P}})$.

2.2.3. Trace fragments and traces

A *trace* of a path is the sequence of public values of states of the path. Formally, the trace of a path $\pi = s_0 s_1 \ldots s_n^\omega$ is defined as $\overline{T} = trace(\pi) = V_l(s_0)V_l(s_1)\ldots V_l(s_n)^\omega$. The trace of a finite path fragment $\hat{\pi} = s_0 s_1 \ldots s_n$ is defined as $\hat{T} = trace(\hat{\pi}) = V_l(s_0)V_l(s_1)\ldots V_l(s_n)$ and is called a trace fragment of length n. $Traces(\mathcal{M}_\delta^{\mathrm{P}})$ denotes the set of traces of $\mathcal{M}_\delta^{\mathrm{P}}$, i.e., $Traces(\mathcal{M}_\delta^{\mathrm{P}}) = \{trace(\pi) \mid \pi \in Paths(\mathcal{M}_\delta^{\mathrm{P}})\}$. The set of trace fragments of length i in $\mathcal{M}_\delta^{\mathrm{P}}$ is denoted by $Traces_{\ll i}(\mathcal{M}_\delta^{\mathrm{P}})$. We will use the concept of traces to measure expected, maximum and minimum leakages. For computing bounded time leakage, we will use trace fragments.

Let $Paths(\overline{T})$ be the set of paths that have the trace $\overline{T}$, i.e., $Paths(\overline{T}) = \{\pi \mid \pi \in Paths(\mathcal{M}_\delta^{\mathrm{P}}) : trace(\pi) = \overline{T}\}$. Assuming $\hat{T}$ is a trace fragment of size i, $Paths_{\ll i}(\hat{T})$ is defined as the set of path fragments that have the trace fragment $\hat{T}$, i.e., $Paths_{\ll i}(\hat{T}) = \{\hat{\pi} \mid \hat{\pi} \in Paths_{\ll i}(\mathcal{M}_\delta^{\mathrm{P}}) : trace(\hat{\pi}) = \hat{T}\}$.

2.2.4. Trace probabilities

The occurrence probability of a trace $\overline{T}$ in $\mathcal{M}_\delta^{\mathrm{P}}$ is defined as the sum of the occurrence probabilities of paths that have the trace $\overline{T}$, i.e.,

$$Pr(T = \overline{T}) = \sum_{\pi \in Paths(\overline{T})} Pr(\pi), \tag{1}$$

where T is a trace variable and

$$Pr(\pi = s_0 s_1 \ldots s_n^\omega) = \begin{cases} \zeta(s_0) & \text{if } n = 0, \\ \zeta(s_0). \prod_{0 \leq i < n} \mathbf{P}(s_i, s_{i+1}) & \text{otherwise.} \end{cases}$$

Let the function $V_h(s)$ determine the value of the variable h in the state $s \in S$. Then, the occurrence probability of paths that have the trace $\overline{T}$ and the secret value $\overline{h}$ is defined as

$$Pr(h = \overline{h}, T = \overline{T}) = \sum_{\pi \in Paths(\overline{T}),\ V_h(\pi[0]) = \overline{h}} Pr(\pi), \tag{2}$$

Note that $Pr(T = \overline{T}) = \sum_{\overline{h} \in Val_h} Pr(h = \overline{h}, T = \overline{T})$. Similarly, the occurrence probability of a trace fragment $\hat{T} \in Traces_{\ll i}(\mathcal{M}_\delta^{\mathrm{P}})$ is defined as $Pr(T_{\ll i} = \hat{T}) = \sum_{\hat{\pi} \in Paths_{\ll i}(\hat{T})} Pr(\hat{\pi})$, where $T_{\ll i}$ is a random variable for trace fragments of length i and

$$Pr(\hat{\pi} = s_0 s_1 \ldots s_i) = \begin{cases} \zeta(s_0) & \text{if } i = 0, \\ \zeta(s_0). \prod_{0 \leq j < i} \mathbf{P}(s_j, s_{j+1}) & \text{otherwise.} \end{cases}$$

The occurrence probability of path fragments that have the trace fragment $\hat{T}$ and the secret value $\overline{h}$, i.e., $Pr(h = \overline{h}, T_{\ll i} = \hat{T})$ is defined similar to $Pr(h = \overline{h}, T = \overline{T})$.

2.2.5. DAG structure of program models

We assumed the programs always terminate and states indicate the current values of the variables and the program counter. Furthermore, loops of the program are unfolded. This implies that Markovian models of every program takes the form of a *directed acyclic graph (DAG)*, ignoring later-added self-loops of final states. Initial states of the program are represented as roots of the DAG and final states as leaves. Therefore, there is no loop in the Markovian models (except later-added self-loops) and reachability probabilities coincide with long-run probabilities [12].

3. The proposed approach

In this section, the proposed approach for leakage quantification is explained. The approach should consider leakages in intermediate states of the program executions, take into account effect of the scheduling policy, and measure the leakage in the context of attacker model described in the following.

3.1. Modeling the program and the attacker

Suppose a terminating shared-memory multi-threaded program `P`, running under control of a scheduling policy δ. Furthermore, assume an *attacker* being able to observe source code of the program, select a scheduling policy, run the program with the chosen policy, and observe the execution traces. For simplicity, it is assumed the multi-threaded program has a secret input variable `h`, a public output variable `l`, and possibly several neutral variables. Val_l and Val_h denote the finite sets of values of `l` and `h`, respectively. The variable `h` has a fixed value during the program executions. If the attacker can deduce information about `h` by observing sequences of values of `l`, then the program is said to have a *leakage*.

Suppose the public variable and the secret variable are stored in the shared memory of the multi-threaded program and the attacker can read the public variable, but cannot access the secret variable (i.e., access control and memory protection works correctly). Furthermore, suppose the attacker might try to guess the value of `h` (in just one try or multiple tries), after executing the program and observing the sequence of values of `l`.

The attacker has a *prior knowledge* of the secret, which is modeled as a *prior probability distribution* over the possible values of the secret. Let $Pr(h)$ denote the attacker's prior knowledge. Here, the attacker is assumed to be *probabilistic*, i.e., he knows size of the secret, in addition to some accurate constraints about the values of `h`. For instance, the attacker could know that `h` is 2 bits long, its value is not 3, the probability that its value is 1 is 0.4, and the probability that its value is 0 is twice the probability that it is 2. The prior distribution encoding these constraints is $Pr(h) = \{0 \mapsto 0.4, 1 \mapsto 0.4, 2 \mapsto 0.2\}$[2] A special case of the probabilistic attacker is *ignorant* attacker [7], who has no prior information about the value of `h` except its size. Thus, the ignorant attacker's initial knowledge is a uniform prior distribution on `h`.

Considering these assumptions about the attacker, the program model is built in two steps.

Step 1: Define an MDP representing `P`. Operational semantics of the multi-threaded program `P` is represented by an MDP $\mathcal{M}^{\mathtt{P}} = (S, Act, \mathbf{P}, \zeta, Val_l, V_l)$. It models executions of `P`, with all possible interleavings of the threads. As the attacker is able to observe the public values, the labeling function is restricted to `l` and traces of $\mathcal{M}^{\mathtt{P}}$ are the sequences of public values valid during the executions. The initial distribution ζ is determined by the prior knowledge of the probabilistic attacker, i.e., $\zeta(s_0) = Pr(h = V_h(s_0))$ for all $s_0 \in Init(\mathcal{M}^{\mathtt{P}})$.

Example 1. Consider the program `P1` from Section 1 and suppose `h` is a 2-bit random variable. The attacker's prior knowledge is the size of `h`, yielding a uniform distribution on `h`: $Pr(h) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$. The MDP $\mathcal{M}^{\mathtt{P1}}$ of the program is shown in Fig. 1. The initial distribution ζ is determined by $Pr(h)$, i.e., $\zeta = \{s_0 \mapsto \frac{1}{4}, s_3 \mapsto \frac{1}{4}, s_5 \mapsto \frac{1}{4}, s_7 \mapsto \frac{1}{4}\}$. Each state is labeled by the value of `l` in that state. Each transition is labeled by an action (a program statement) and a probability. For instance, the transition from s_0 to s_1 has the action α : `if l=1 then l:=h else skip fi` and the probability $\mathbf{P}(s_0, \alpha, s_1) = 1$; or the transition from s_0 to s_9 has the action β : `l:=1` and the probability $\mathbf{P}(s_0, \beta, s_9) = 1$.

Step 2: Resolve the nondeterminism in the MDP and produce an MC. The scheduling policy is represented by a memoryless probabilistic scheduler δ. As the MDP $\mathcal{M}^{\mathtt{P}}$ is executed under the control of the scheduler δ, all nondeterministic transitions are resolved and an MC $\mathcal{M}^{\mathtt{P}}_{\delta} = (S, \mathbf{P}_{\delta}, \zeta, Val_l, V_l)$ is produced. Each state of $\mathcal{M}^{\mathtt{P}}_{\delta}$ shows the current values of `h`, `l`, possibly neutral variables, and the program counter. The MC $\mathcal{M}^{\mathtt{P}}_{\delta}$ contains all execution traces that the attacker may observe.

Back to example 1, we choose the scheduler to be uniform. The uniform scheduler, denoted by the function *uni*, picks each thread with the same probability. This yields the scheduler is defined as $uni(s_0) = \{\alpha \mapsto \frac{1}{2}, \beta \mapsto \frac{1}{2}\}$, $uni(s_1) = \{\beta \mapsto 1\}$, $uni(s_9) = \{\alpha \mapsto 1\}$, $uni(s_{10}) = \{\gamma \mapsto 1\}$, $uni(s_2) = \{\gamma \mapsto 1\}$, $uni(s_{17}) = \{\tau \mapsto 1\}$, $uni(s_3) = \{\alpha \mapsto \frac{1}{2}, \beta \mapsto \frac{1}{2}\}$, $uni(s_4) = \{\beta \mapsto 1\}$, $uni(s_{11}) = \{\alpha \mapsto 1\}$, $uni(s_{12}) = \{\gamma \mapsto 1\}$, $uni(s_{18}) = \{\tau \mapsto 1\}$, $uni(s_5) = \{\alpha \mapsto \frac{1}{2}, \beta \mapsto \frac{1}{2}\}$, $uni(s_6) = \{\beta \mapsto 1\}$, $uni(s_{13}) = \{\alpha \mapsto 1\}$, $uni(s_{14}) = \{\gamma \mapsto 1\}$, $uni(s_{19}) = \{\gamma \mapsto 1\}$, $uni(s_{20}) = \{\tau \mapsto 1\}$, $uni(s_7) = \{\alpha \mapsto \frac{1}{2}, \beta \mapsto \frac{1}{2}\}$, $uni(s_8) = \{\beta \mapsto 1\}$, $uni(s_{15}) = \{\alpha \mapsto 1\}$, $uni(s_{16}) = \{\gamma \mapsto 1\}$, $uni(s_{22}) = \{\gamma \mapsto 1\}$, $uni(s_{21}) = \{\tau \mapsto 1\}$.

The MC $\mathcal{M}^{\mathtt{P1}}_{uni}$ of the program `P1` running under control of the uniform scheduler is depicted in Fig. 2. In this figure, transitions are labeled by the transition probability. This MC was produced by the *PRISM* model checking software [13]. We

[2] Only elements with a positive probability are shown.

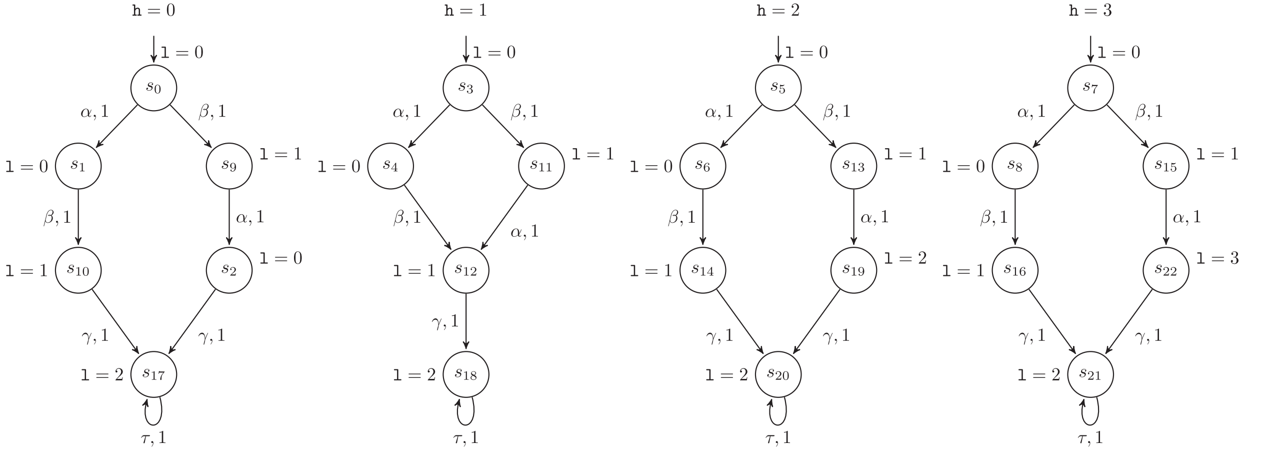


Fig. 1. $\mathcal{M}^{P1}$: MDP of the program P1, with α denoting `ifl=1thenl:=helseskipfi`, β denoting `l:=1`, γ denoting `l:=2`, and τ denoting termination of the program.

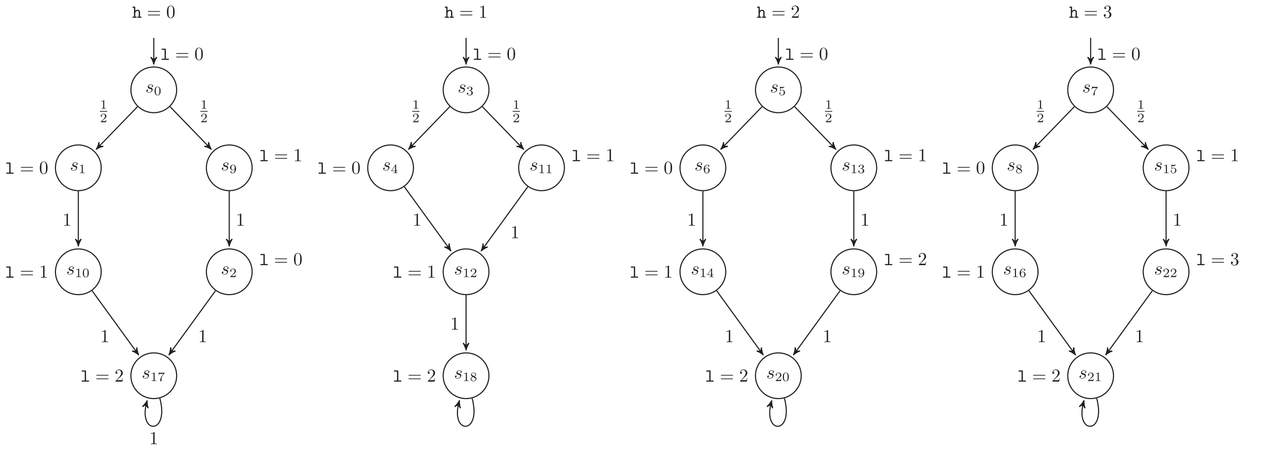


Fig. 2. $\mathcal{M}_{uni}^{P1}$: MC of the program P1 with the uniform scheduler.

implemented the program in the language and PRISM automatically built the model. PRISM has its own way of numbering states. That's why state numbers (indexes) in $\mathcal{M}^{P1}$ (Fig. 1) and $\mathcal{M}_{uni}^{P1}$ (Fig. 2) are not sequential.

$\mathcal{M}_{uni}^{P1}$ has 5 traces with different occurrence probabilities: $T_0 =< 0, 0, 1, 2^\omega >,\ Pr(T = T_0) = \frac{1}{2},\ T_1 =< 0, 1, 0, 2^\omega >,\ Pr(T = T_1) = \frac{1}{8},\ T_2 =< 0, 1, 1, 2^\omega >,\ Pr(T = T_2) = \frac{1}{8},\ T_3 =< 0, 1, 2, 2^\omega >,\ Pr(T = T_3) = \frac{1}{8},\ T_4 =< 0, 1, 3, 2^\omega >,\ Pr(T = T_4) = \frac{1}{8}$ and different joint probabilities of secrets and traces: $Pr(h = 0, T = T_0) = \frac{1}{8},\ Pr(h = 1, T = T_0) = \frac{1}{8},\ Pr(h = 2, T = T_0) = \frac{1}{8},\ Pr(h = 3, T = T_0) = \frac{1}{8},\ Pr(h = 0, T = T_1) = \frac{1}{8},\ Pr(h = 1, T = T_2) = \frac{1}{8},\ Pr(h = 2, T = T_3) = \frac{1}{8},\ Pr(h = 3, T = T_4) = \frac{1}{8}$. The remaining joint probabilities of secrets and traces are zero.

3.2. Expected leakage

In this section, a trace-based method is provided to compute the expected leakage of a multi-threaded program. In what follows, we fix an MC $\mathcal{M}_\delta^{P} = (S, \mathbf{P}_\delta, \zeta, Val_l, V_l)$ which models the executions of the multi-threaded program P under the control of a scheduler δ. The attacker, with the prior knowledge $Pr(h)$, chooses the scheduler δ, executes the program (possibly several times), and observes the traces, i.e., sequences of public values. He cannot discriminate between those paths that have the *same trace*. For instance, in $\mathcal{M}_{uni}^{P1}$ (Fig. 2) the attacker only observes 5 traces, whereas there are eight different paths. The implication is that he cannot distinguish those pairs of paths that have the same trace. He also does not know the value of the secret input. He only knows a priori probability distribution of h and can compute possible values of it after observing each execution trace.

As mentioned earlier, we assume that the attacker, after observing the program traces, might try to guess the value of h. We can compute the uncertainty of the attacker using any gain function such as Shannon entropy or Renyi's min-entropy. However, Renyi's min-entropy is an upper bound of leakage over all gain functions [14]. Therefore, we measure the uncertainty of the attacker by the Renyi's min-entropy (Definition 2).

The attacker has an initial knowledge on h. Each time he executes the program and observes a trace, he may gain more knowledge on the secret. For instance in $\mathcal{M}_{uni}^{\texttt{P1}}$ when the attacker observes the trace $T_1 =< 0, 1, 0, 2^{\omega} >$, he learns that the value of h is 0; or when he observes T_2, he can conclude that the value of h is 1. This means that as the attacker observes the execution traces, he may learn more knowledge and his uncertainty about h may be reduced. In fact, the attacker's initial knowledge on h is modeled by the *prior distribution* $Pr(h)$ and his final knowledge after observing the traces is modeled by the *posterior distribution* $Pr(h|T)$. These yield the expected leakage of the MC $\mathcal{M}_{\delta}^{\texttt{P}}$ is computed as

$$\begin{aligned}\mathcal{L}_{\mathcal{E}}(\texttt{P}_{\delta}) &= \text{initial uncertainty - remaining uncertainty}\\ &= \mathcal{H}_{\infty}(h) - \mathcal{H}_{\infty}(h|T)\\ &= \mathcal{H}_{\infty}(h) - \sum_{\overline{T}\in Traces(\mathcal{M}_{\delta}^{\texttt{P}})} Pr(T=\overline{T}).\mathcal{H}_{\infty}(h|T=\overline{T})\\ &= -\log_2 \max_{\overline{h}\in Val_h} Pr(h=\overline{h}) + \sum_{\overline{T}\in Traces(\mathcal{M}_{\delta}^{\texttt{P}})} Pr(T=\overline{T}).\log_2 \max_{\overline{h}\in Val_h} Pr(h=\overline{h}|T=\overline{T}),\end{aligned}$$

where

$$Pr(h=\overline{h}|T=\overline{T}) = \frac{Pr(h=\overline{h}, T=\overline{T})}{Pr(T=\overline{T})}, \tag{3}$$

in which $Pr(h=\overline{h}, T=\overline{T})$ is the occurrence probability of paths that have the trace $\overline{T}$ and the secret value $\overline{h}$ (Eq. (2)) and $Pr(T=\overline{T})$ is the occurrence probability of the trace $\overline{T}$ (Eq. (1)).

The quantification method employs a *depth-first path exploration* algorithm to exhaustively traverse $\mathcal{M}_{\delta}^{\texttt{P}}$ and compute $Paths(\mathcal{M}_{\delta}^{\texttt{P}})$ and $Traces(\mathcal{M}_{\delta}^{\texttt{P}})$. Then, prior and posterior probabilities and entropies are calculated, from which the expected leakage is computed.

Back to the example 1, the initial uncertainty is quantified as the Renyi's min-entropy of h in the initial states: $\mathcal{H}_{\infty}(h) = -\log_2 \frac{1}{4} = 2$ $(bits)$. The remaining uncertainty is quantified as the Renyi's min-entropy of h after observing the traces. There are 5 traces, which result in different posterior distributions: $Pr(h|T=T_0) = \{0 \mapsto \frac{1}{4},\ 1 \mapsto \frac{1}{4},\ 2 \mapsto \frac{1}{4},\ 3 \mapsto \frac{1}{4}\}$, $Pr(h|T=T_1) = \{0 \mapsto 1\}$, $Pr(h|T=T_2) = \{1 \mapsto 1\}$, $Pr(h|T=T_3) = \{2 \mapsto 1\}$, $Pr(h|T=T_4) = \{3 \mapsto 1\}$. Consequently, the remaining uncertainty is quantified as

$$\sum_{i\in\{0,1,2,3,4\}} Pr(T=T_i).\mathcal{H}_{\infty}(h|T=T_i) = -\frac{1}{2} \times \log_2 \frac{1}{4} - 4 \times \frac{1}{8} \times \log_2 1 = 1\ (bit).$$

Finally, the expected leakage of the program P1 running with the uniform scheduler is computed as $\mathcal{L}_{\mathcal{E}}(\texttt{P1}_{uni}) = 2 - 1 = 1$ (bit).

3.3. Bounded time/leakage analysis

There are two similar bounded approaches to the quantitative information flow: measuring the leakage of a program at a given time t, or measuring how long it takes to leak a given amount of information [15]. Here, time is abstracted by considering each time unit as a transition step in the Markov chain of the program.

3.3.1. Bounded time

We want to measure the leakage of a program at a given time. Assume an attacker who is able to observe the program traces for a finite time t. Then, the question is how much information he can infer about h. The set of trace fragments observable at the given time t is $Traces_{\ll t}(\mathcal{M}_{\delta}^{\texttt{P}})$ and the uncertainty at t is computed by $\mathcal{H}_{\infty}(h|T_{\ll t})$. These yield the (expected) leakage up to a given time t is measured by

$$\begin{aligned}\mathcal{L}_t(\texttt{P}_{\delta}) &= \mathcal{H}_{\infty}(h) - \mathcal{H}_{\infty}(h|T_{\ll t})\\ &= \mathcal{H}_{\infty}(h) - \sum_{\hat{T}\in Traces_{\ll t}(\mathcal{M}_{\delta}^{\texttt{P}})} Pr(T_{\ll t}=\hat{T}).\mathcal{H}_{\infty}(h|T_{\ll t}=\hat{T}).\end{aligned}$$

where $\mathcal{H}_{\infty}(h|T_{\ll t}=\hat{T}) = \log_2 \max_{\overline{h}\in Val_h} Pr(h=\overline{h}|T_{\ll t}=\hat{T})$ and $Pr(h=\overline{h}|T_{\ll t}=\hat{T})$ is defined similar to $Pr(h=\overline{h}|T=\overline{T})$ (Eq. (3)).

The bounded time leakage function $\mathcal{L}_t$ is non-decreasing [15] and finally converges to $\mathcal{L}_{\mathcal{E}}$. One can also compute the (expected) leakage from a time t_1 to a time $t_2 > t_1$:

$$\mathcal{L}_{(t_1,t_2)}(\texttt{P}_{\delta}) = \mathcal{H}_{\infty}(h|T_{\ll t_1}) - \mathcal{H}_{\infty}(h|T_{\ll t_2}). \tag{4}$$

Back to the example 1 ($\texttt{P1}_{uni}$), at time $t = 0$ the leakage is 0. At time $t = 1$, there are two trace fragments: $Traces_{\ll 1}(\texttt{P1}_{uni}) = \{\hat{T}_0, \hat{T}_1\}$, where $\hat{T}_0 =< 0, 0 >$, $Pr(T_{\ll 1}=\hat{T}_0) = \frac{1}{2}$, $\hat{T}_1 =< 0, 1 >$, $Pr(T_{\ll 1}=\hat{T}_1) = \frac{1}{2}$. Each trace fragment results

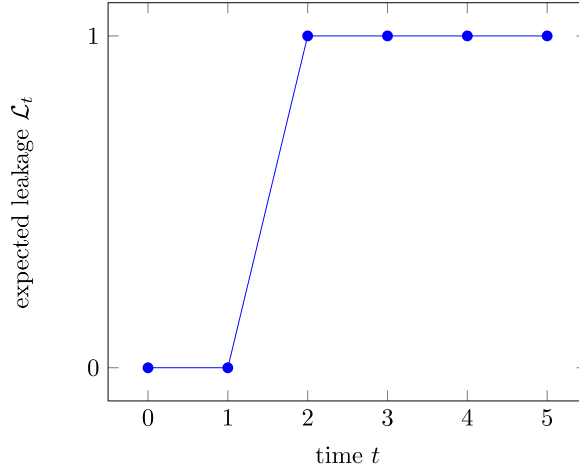


Fig. 3. Bounded time leakage results for $\mathtt{P1}_{uni}$.

in different posterior distributions: $Pr(h|T_{\ll 1} = \hat{T}_0) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$, $Pr(h|T_{\ll 1} = \hat{T}_1) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$. These yield the expected leakage of the program $\mathtt{P1}_{uni}$ up to time $t = 1$ is computed as $\mathcal{L}_{t=1}(\mathtt{P1}_{uni}) = 2 - (-\frac{1}{2} \times \log_2 \frac{1}{4} - \frac{1}{2} \times \log_2 \frac{1}{4}) = 0$.

At time $t = 2$, there are five trace fragments: $Traces_{\ll 2}(\mathtt{P1}_{uni}) = \{\hat{T}_0, \hat{T}_1, \hat{T}_2, \hat{T}_3, \hat{T}_4\}$, where $\hat{T}_0 =< 0, 0, 1 >$, $Pr(T_{\ll 2} = \hat{T}_0) = \frac{1}{2}$, $\hat{T}_1 =< 0, 1, 0 >$, $Pr(T_{\ll 2} = \hat{T}_1) = \frac{1}{8}$, $\hat{T}_2 =< 0, 1, 1 >$, $Pr(T_{\ll 2} = \hat{T}_2) = \frac{1}{8}$, $\hat{T}_3 =< 0, 1, 2 >$, $Pr(T_{\ll 2} = \hat{T}_3) = \frac{1}{8}$, $\hat{T}_4 =< 0, 1, 3 >$, $Pr(T_{\ll 2} = \hat{T}_4) = \frac{1}{8}$. Each trace fragment results in different posterior distributions: $Pr(h|T_{\ll 2} = \hat{T}_0) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$, $Pr(h|T_{\ll 2} = \hat{T}_1) = \{0 \mapsto 1\}$, $Pr(h|T_{\ll 2} = \hat{T}_2) = \{1 \mapsto 1\}$, $Pr(h|T_{\ll 2} = \hat{T}_3) = \{2 \mapsto 1\}$, $Pr(h|T_{\ll 2} = \hat{T}_4) = \{3 \mapsto 1\}$. These yield the expected leakage of the program $\mathtt{P1}_{uni}$ up to time $t = 2$ is computed as $\mathcal{L}_{t=2}(\mathtt{P1}_{uni}) = 2 - (-\frac{1}{2} \times \log_2 \frac{1}{4} - 4 \times \frac{1}{8} \times \log_2 1) = 1$.

For $t \geq 3$ the expected leakage was computed in Section 3.2: $\mathcal{L}_{t \geq 3}(\mathtt{P1}_{uni}) = 1$. The leakage results for $\mathtt{P1}_{uni}$ are summarized in Fig. 3. As you can see the leakage function is non-decreasing.

The following theorem discusses the relationship between expected and bounded time leakages.

Theorem 7. *Let $\mathcal{M}_\delta^{\mathtt{P}}$ be an MC, modeling the executions of a terminating multi-threaded program $\mathtt{P}$ under control of a scheduler δ and $Pr(h)$ denote the prior knowledge of a probabilistic attacker. Then, $\mathcal{L}_t(\mathtt{P}_\delta)$ finally converges to $\mathcal{L}_\mathcal{E}(\mathtt{P}_\delta)$.*

Proof. Since we assumed $\mathtt{P}$ is terminating, all path fragments eventually lead to final states and form paths, from which $\mathcal{L}_\mathcal{E}(\mathtt{P}_\delta)$ is computed. Therefore, the bounded time leakage function $\mathcal{L}_t(\mathtt{P}_\delta)$ eventually converges to $\mathcal{L}_\mathcal{E}(\mathtt{P}_\delta)$. □

3.3.2. Bounded leakage

We want to determine how many time units it takes for the program to leak c bits of information. Since the program is assumed to be terminating, the bounded time leakage function $\mathcal{L}_t$ converges to $\mathcal{L}_\mathcal{E}$ in a finite time. There are two cases:

- $c > \mathcal{L}_\mathcal{E}$: Then, $\mathcal{L}_t$ will converge before reaching the constant c. Therefore, there is no such a time in which the program leaks c bits.
- $c \leq \mathcal{L}_\mathcal{E}$: Then, $\mathcal{L}_t$ will reach c. The time it reaches c is the answer to the question.

3.4. Maximum and minimum leakages

The attacker may execute the program several times and depending on the value of the secret, he may observe different execution traces. Each execution may result in a different amount of leakage. A possible quantification of the security of a program is the *maximum leakage*, which is the maximal value of leakages occurred in all traces of the program. Maximum leakage, denoted $\mathcal{L}_{\max}(\mathtt{P}_\delta)$, is an upper bound of leakage that an attacker with prior knowledge $Pr(h)$ can infer from $\mathtt{P}_\delta$. Note that maximum leakage is different from channel capacity, which is an upper bound of leakage for all attackers with the same observational power but different prior knowledge on the secret.

The maximum leakage of the MC $\mathcal{M}_\delta^{\mathtt{P}}$ is computed as

$$\mathcal{L}_{max}(\mathtt{P}_\delta) = \mathcal{H}_\infty(h) - \min_{\overline{T} \in Traces(\mathcal{M}_\delta^{\mathtt{P}})} \mathcal{H}_\infty(h|T = \overline{T}).$$

Note that there might be more than one trace with minimum min-entropy. Let $Traces_{maxL}(\mathcal{M}_\delta^{\mathtt{P}})$ denote the set of traces that result in the maximum leakage: $Traces_{maxL}(\mathcal{M}_\delta^{\mathtt{P}}) = \{\overline{T} \in Traces(\mathcal{M}_\delta^{\mathtt{P}}) \mid \mathcal{H}_\infty(h|T = \overline{T}) = \min_{\overline{T}' \in Traces(\mathcal{M}_\delta^{\mathtt{P}})} \mathcal{H}_\infty(h|T = \overline{T}')\}$.

Then, the probability of occurrence of the maximum leakage is computed as

$$Pr_{maxL}(\mathcal{M}_\delta^{\mathtt{P}}) = \sum_{\overline{T} \in Traces_{maxL}(\mathcal{M}_\delta^{\mathtt{P}})} Pr(T = \overline{T}).$$

For instance, in $\mathtt{P1}_{uni}$ (Fig. 2) the traces T_1, T_2, T_3, and T_4 result in the maximum leakage: $\mathcal{L}_{\max}(\mathtt{P1}_{uni}) = 2 - (-\log_2 1) = 2$ (*bits*). The probability of occurrence of this leakage is $Pr_{maxL}(\mathcal{M}_{uni}^{\mathtt{P1}}) = 4 \times \frac{1}{8} = \frac{1}{2}$.

Another possible quantification of the security is the *minimum leakage*, which is the minimal value of leakages occurred in all execution traces of the program. It is a lower bound of leakage that an attacker with prior knowledge $Pr(h)$ can infer from $\mathtt{P}_\delta$. Minimum leakage of the MC $\mathcal{M}_\delta^{\mathtt{P}}$ is computed as

$$\mathcal{L}_{min}(\mathtt{P}_\delta) = \mathcal{H}_\infty(h) - \max_{\overline{T} \in Traces(\mathcal{M}_\delta^{\mathtt{P}})} \mathcal{H}_\infty(h|T = \overline{T}).$$

The set of traces that result in the minimum leakage is denoted by $Traces_{minL}(\mathcal{M}_\delta^{\mathtt{P}})$: $Traces_{minL}(\mathcal{M}_\delta^{\mathtt{P}}) = \{\overline{T} \in Traces(\mathcal{M}_\delta^{\mathtt{P}}) \mid \mathcal{H}_\infty(h|T = \overline{T}) = \max_{\overline{T}' \in Traces(\mathcal{M}_\delta^{\mathtt{P}})} \mathcal{H}_\infty(h|T = \overline{T}')\}$. Then, the probability of occurrence of the minimum leakage is computed as

$$Pr_{minL}(\mathcal{M}_\delta^{\mathtt{P}}) = \sum_{\overline{T} \in Traces_{minL}(\mathcal{M}_\delta^{\mathtt{P}})} Pr(T = \overline{T}).$$

For instance, in $\mathtt{P1}_{uni}$ the trace T_0 results in the minimum leakage: $\mathcal{L}_{\min}(\mathtt{P1}_{uni}) = 2 - (-\log_2 \frac{1}{4}) = 0$ (*bits*). The probability of occurrence of this leakage is $Pr_{minL}(\mathcal{M}_{uni}^{\mathtt{P1}}) = \frac{1}{2}$.

These maximum and minimum bounds restrict the program leakage and may be useful in deciding either to accept or to reject a program. The following theorem discusses the relationship between expected, maximum, and minimum leakages.

Lemma 8. *Let $\mathcal{M}_\delta^{\mathtt{P}}$ be an MC, modeling the executions of a terminating multi-threaded program $\mathtt{P}$ under control of a scheduler δ. Then, we have: $\mathcal{L}_{\min}(\mathtt{P}_\delta) \leq \mathcal{L}_{\mathcal{E}}(\mathtt{P}_\delta) \leq \mathcal{L}_{\max}(\mathtt{P}_\delta)$.*

Proof. This follows immediately from definitions of $\mathcal{L}_{\min}(\mathtt{P}_\delta)$, $\mathcal{L}_{\mathcal{E}}(\mathtt{P}_\delta)$, and $\mathcal{L}_{\max}(\mathtt{P}_\delta)$. □

4. Implementation and case study

The proposed approach has been implemented in a tool *PRISM-Leak*, which is built upon the *PRISM model checker* [13]. PRISM-Leak is available at https://github.com/alianoroozi/PRISM-Leak.

The PRISM model checker uses a state-based language, *the PRISM language*, to specify the program, construct the program model, and analyze it. The PRISM language supports modeling systems that exhibit random or probabilistic behavior. PRISM can build and analyze several types of probabilistic models, including discrete-time Markov chains (dtmc). A PRISM model is composed of a number of *modules* which can interact with each other. The definition of each module contains *variables* and *commands*. The values of the variables at any given time determine the *state* of the module and a change in the value of the variables indicates a *transition*. The commands describe the behavior of the module and take the form:

```
[] guard -> prob_1:update_1 + {\ldots} + prob_n:update_n;
```

The guard is a predicate over the variables in the model. Each update (`update_i`) describes a transition which the module can make if the guard is true and is assigned a probability (`prob_i`) which determines the probability of the corresponding transition.

In addition to the local variables of modules, a PRISM model can contain *global variables*, to which all modules have read-/write access. The global variables encode the shared memory of the program. The global state of the model is determined by the local state of all modules and values of the global variables.

We use modules to model threads of a multi-threaded program. Then, the PRISM model (in our case, dtmc) corresponding to a multi-threaded program is constructed as the parallel composition of its modules representing threads. In every state of the dtmc model, there is a set of commands which are *enabled*. The PRISM language allows to define how modules are composed in parallel. Thus, the scheduler's choices can be encoded by specifying at each step, which modules are enabled to make a transition. If not encoded, each enabled command is selected with equal probability (i.e., uniform scheduling).

We modified parser of PRISM and added two reserved keywords `observable` and `secret` to indicate public (observable) and secret variables. The keyword `observable` is placed before identifier of a public variable and `secret` is placed before identifier of a secret variable. Example declarations of a 2-bit public global variable `l` and a 3-bit secret local variable `h` is:

```
global observable l : [0.3];
secret h : [0.7];
```

When running PRISM, it builds a model (in our case, a dtmc) and stores it using BDDs (Binary Decision Diagrams) and MTBDDs (Multi-Terminal Binary Decision Diagrams), which are state-of-the-art symbolic data structures. PRISM-Leak uses

these data structures to extract the set of *reachable* states and also create a *sparse matrix* containing the transitions. It then traverses the model to extract the paths and traces and finally computes the information leakage values.

To show applicability and feasibility of the proposed approach, two case studies are discussed.

4.1. Case study A: Program from Smith and Volpano

As the first case study, consider the (non-probabilistic) multi-threaded program `[Alpha || Beta || Gamma]`, which is borrowed from [16]. We call it the Smith–Volpano program throughout the paper.

```
Alpha ≡ while mask != 0 do
   while trigger0 = 0 do od; /* busy waiting */
   result := result | mask; /* bitwise 'or' */
   trigger0 := 0;
   maintrigger := maintrigger + 1;
   if maintrigger = 1 then trigger1 := 1 fi;
  od
Beta ≡ while mask != 0 do
   while trigger1 = 0 do od; /* busy waiting */
   result := result & ~mask; /* bitwise 'and' with the complement of mask */
   trigger1 := 0;
   maintrigger := maintrigger + 1;
   if maintrigger = 1 then trigger0 := 1 fi;
  od
Gamma ≡ while mask != 0 do
   maintrigger := 0;
   if (PIN & mask) = 0 then trigger0 := 1 else trigger1 := 1 fi;
   while maintrigger != 2 do od; /* busy waiting */
   mask := mask / 2;
   trigger0 := 1;
   trigger1 := 1;
  od
```

The program consists of three threads. Assume that `PIN` is a secret variable and `result` is a public variable. If the scheduling of the threads is fair, i.e., each thread gets its turn infinitely often, and the program starts in an initial state where `maintrigger`=0, `trigger0`=0, `trigger1`=0, and `result`=0, the program leaks some bits of `PIN` into `result`. Assume `PIN` is n bits long, the attacker's prior knowledge is the size of `PIN`, and the initial value of `mask` equals to 2^k $(k < n)$. Then, Smith–Volpano leaks $k+1$ bits of `PIN` into `result`.

The PRISM description for the Smith–Volpano program is given in Appendix A. It is composed of three modules `Alpha`, `Beta`, and `Gamma`, where each module models a thread. For the sake of brevity, we only allow modules to change turn when they are in the busy waiting state. The global variables `result`, `mask`, `pin`, `trigger0`, `trigger1`, `maintrigger`, and `turn` encode the shared variables (memory) of the program. States of the model are represented by the values of its variables, i.e., (`result`, `mask`, `pin`, `trigger0`, `trigger1`, `maintrigger`, `turn`, `c1`, `c2`, `c3`). In some states of the model, there are more than one enabled command. Since the model type is defined to be a Markov chain (dtmc), the choice between which command is performed is *uniform*, i.e., each command is selected with equal probability. For instance, when $n = 3$ and the initial value of `mask` is set to 2, in state (0, 2, 0, 1, 0, 0, 3, 0, 0, 3) (busy waiting of `Gamma`) there are two commands enabled:

```
[] turn=3 & c3=3 & maintrigger!=2 -> (turn'=2);
[] turn=3 & c3=3 & maintrigger!=2 -> (turn'=1);
```

In this state the following probability distribution results: $0.5 : (0, 2, 0, 1, 0, 0, 1, 0, 0, 3) + 0.5 : (0, 2, 0, 1, 0, 0, 2, 0, 0, 3)$.

The leakage values computed for the Smith–Volpano program with `mask`=2 and an attacker with a uniform prior knowledge on `PIN` are given in Table 1. In this table, $\mathcal{M}^{sv}_{uni}$ denotes the MC of Smith–Volpano run with a uniform scheduler. The

Table 1
Evaluation results for the Smith–Volpano program with `mask`=2 and an attacker with a uniform prior knowledge on the secret.

n	$\mathcal{M}^{sv}_{uni}$		$\mathcal{L}_{\mathcal{E}}$ (bits)	$\mathcal{L}_{\max}$	Pr_{maxL}	$\mathcal{L}_{\min}$	Pr_{minL}
	$\#st$	$\#tr$					
2	228	236	2 (100%)	2	1	2	1
3	456	472	2 (66%)	2	1	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9	29,184	30,208	2 (22%)	2	1	2	1
10	58,368	60,416	2 (20%)	2	1	2	1
11	116,736	120,832	2 (18%)	2	1	2	1

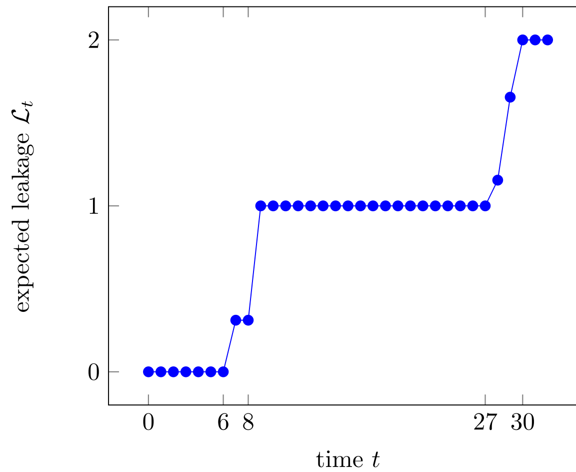


Fig. 4. Bounded time leakage results for the Smith–Volpano program with `mask=2` and a uniform distribution of `PIN`.

column $\mathcal{L}_\mathcal{E}$ contains the expected leakage values along with the percentage of leakage, which is defined as the amount of leakage over the initial uncertainty.

Each increase in n (i.e., the bit size of `PIN`) doubles the number of states and transitions of the Markov model. Since `mask` is initially set to 2, the expected leakage in all cases is equal to 2. All runs of the program result in a leakage of 2 bits. That is why $\mathcal{L}_{\max} = \mathcal{L}_{\min} = \mathcal{L}_\mathcal{E} = 2$ and $Pr_{maxL} = Pr_{minL} = 1$.

The bounded time leakage results for all $n \geq 2$ are summarized in Fig. 4. This figure demonstrates that the Smith–Volpano program converges to the expected leakage after 30 steps (transitions).

Leakage in intermediate states of execution. Consider the following scenario: an attacker tries to get the whole value of `PIN` by running the following thread concurrently with other threads:

```
Delta ≡ result := PIN; result := 0
```

This scenario is similar to scenarios discussed at [11] and aims to show how our approach computes leakage in intermediate states of execution. Now, the program consists of four threads, which run concurrently: `[Alpha || Beta || Gamma || Delta]`. The uniform scheduler might run the thread `Delta` at any time of the program execution. Thus, at any time there is a possibility of complete leakage. We implement the program in PRISM for `mask=2` and PRISM-Leak computes the expected leakage to be n. This is while, if we avoided leakage in intermediate states and only considered the final values of `result`, the expected leakage would be 0.

4.2. Case study B: The Single Preference voting protocol

As the second case study, the *Single Preference voting protocol* is analyzed. Assume there are c candidates and n voters. In this randomized protocol each voter expresses a preference for one of the candidates. Then, votes for each candidate are summed up and only the results are published in order to preserve the anonymity of the voters. Votes are secret and results are public. Thus, there are c^n secret values and the secret size is $\log_2 c^n = n \log_2 c$ bits. Note that all vote variables are concatenated to form a single secret variable and all result variables are concatenated to form a single public variable.

Assume the attacker's prior knowledge is the size of the secret, i.e, a uniform distribution on values of the secret. Then, the expected leakage for the protocol with n voters and c candidates corresponds to [1]

$$-\sum_{k_1+k_2+\ldots+k_c=n} \frac{n!}{c^n k_1! k_2! \ldots k_c!} \log_2 \left(\frac{n!}{c^n k_1! k_2! \ldots k_c!} \right).$$

The PRISM description for the protocol in the case where $c = 4$ and $n = 4$ is given in Appendix A. It is composed of four modules `voter1`, `voter2`, `voter3`, and `voter4`, where each module models a voter. The global variables `s1`, `vot1`, `s2`, `vot2`, `s3`, `vot3`, `s4`, `vot4`, `result1`, `result2`, `result3`, and `result4` encode the shared memory of the program. States of the model are represented by the values of its variables, i.e., (`s1`, `vot1`, `s2`, `vot2`, `s3`, `vot3`, `s4`, `vot4`, `result1`, `result2`, `result3`, `result4`). In states that there are more than one enabled command a uniform scheduler selects the command to perform. For instance,

Table 2
Evaluation results for the Single Preference protocol and an attacker with a uniform prior knowledge on the secret.

c	n	$\mathcal{M}_{uni}^{SP}$		$\mathcal{L}_{\mathcal{E}}$ (bits)	$\mathcal{L}_{\max}$	Pr_{maxL}	$\mathcal{L}_{\min}$	Pr_{minL}
		#st	#tr					
4	4	4096	8448	4.815 (60%)	8	0.015	3.415	0.093
	5	32,768	82,944	5.352 (53%)	10	0.003	4.093	0.234
	6	262,144	790,528	5.792 (48%)	12	0.0009	4.508	0.263
	7	2097,152	7356,416	6.164 (44%)	14	0.0002	4.7	0.153
5	4	10,000	20,625	5.838 (62%)	9.287	0.008	4.702	0.192
	5	100,000	253,125	6.546 (56%)	11.609	0.001	4.702	0.038
	6	1000,000	3015,625	7.135 (51%)	13.931	0.0003	5.439	0.115
6	4	20,736	42,768	6.709 (64%)	10.339	0.004	5.754	0.277
	5	248,832	629,856	7.574 (58%)	12.924	0.0007	6.017	0.092
7	4	38,416	79,233	7.468 (66%)	11.229	0.002	6.644	0.349
	5	537,824	1361,367	8.477 (60%)	14.036	0.0004	7.129	0.149
8	4	65,536	135,168	8.139 (67%)	12	0.001	7.415	0.41
	5	1048,576	2654,208	9.28 (61%)	15	0.0002	8.093	0.205

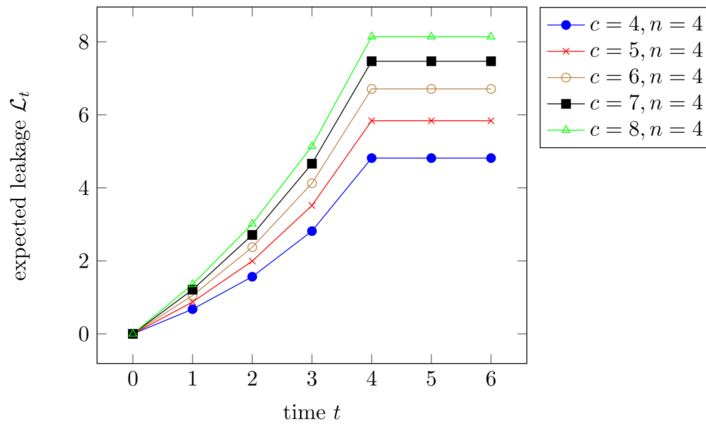


Fig. 5. Bounded time leakage results for the Single Preference protocol and a uniform distribution on the secret.

in state (0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0) there are two commands enabled, one from each module:

```
[] s1=0 & vot1 = 1 & result1<N -> (result1'=result1+1) & (s1'=1);
[] s2=0 & vot2 = 1 & result2<N -> (result2'=result2+1) & (s2'=1);
[] s3=0 & vot3 = 1 & result3<N -> (result3'=result3+1) & (s3'=1);
[] s4=0 & vot4 = 1 & result4<N -> (result4'=result1+4) & (s4'=1);
```

Thus, in this state the following probability distribution results: $0.25 : (1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0) + 0.25 : (0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0) + 0.25 : (0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0) + 0.25 : (0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1)$.

The leakage values computed for the Single Preference protocol and an attacker with a uniform prior knowledge on the secret are given in Table 2. In this table, $\mathcal{M}_{uni}^{SP}$ denotes the MC of the protocol run with a uniform scheduler and symbols #*st* and #*tr* denote the number of states and transitions of the Markov chain. For the purpose of making the table concise, we only show results for $c \geq 4$ and $n \geq 4$.

The results in Table 2 show that as the number of voters increases, the percentage of the expected leakage decreases. For instance, for $c = 4$ as n increases from 4 to 7, the Single Preference leaks 16% less information. However, for a constant n, as the number of candidates increases, the protocol leaks more information. For example, for $n = 4$ as c increases from 4 to 8, the protocol leaks 7% more information.

In all cases of Table 2 the maximum leakage is equal to $n\log_2 c$, which is the secret size. This means in some runs of the Single Preference the whole secret gets leaked. However, as n increases, the probability of maximum leakage decreases. For instance, for $c = 4$ as n increases from 4 to 7, Pr_{maxL} decreases from 0.015 to 0.0002. Another point is that in all cases the minimum leakage is greater than 0. This means all runs of the Single Preference leads to a leakage.

Fig. 5 shows the bounded time leakage results for some cases of the Single Preference protocol. In this figure, bounded time leakage values converge to the final value (expected leakage) after 4 steps.

Notice that the leakage values in Table 2 and Fig. 5 are computed for an attacker with a uniform prior knowledge on the secret. For other prior distributions, the leakage would be different. For instance, consider the case where $c = 2$ and $n = 2$, i.e., the secret values are '1,1', '1,2', '2,1', and '2,2'. For the uniform distribution, the expected leakage is 1.5 (75%). For the prior distribution $Pr(h) = \{`1,1' \mapsto \frac{1}{2}, `1,2' \mapsto \frac{1}{4}, `2,1' \mapsto \frac{1}{8}, `2,2' \mapsto \frac{1}{8}\}$, the expected leakage is 1.405 (80%), or for the prior distribution $Pr(h) = \{`1,1' \mapsto \frac{997}{1000}, `1,2' \mapsto \frac{1}{1000}, `2,1' \mapsto \frac{1}{1000}, `2,2' \mapsto \frac{1}{1000}\}$, the expected leakage is 0.032 (94%), which is close to 0. The latter leakage is almost 0 because the attacker already has a lot of knowledge on the secret (he knows that with probability $\frac{997}{1000}$ the secret value is '1,1') and learns a little knowledge by running the protocol.

5. Related work and comparison

In this section, we review related work and compare our approach with approaches and tools from the literature. We then compare the accuracy, speed, and performance of our approach against various state-of-the-art tools that compute the (expected) leakage.

Classical quantitative information flow analyses [5,17] model the program as an input-output channel with the secret variable as the input, the public variable as the output and the attacker being able to observe the public outputs as the attacker model. However, input-output channels are not a suitable model for quantitative information flow analysis of multi-threaded programs. They get too complex when computing intermediate leakages or analyzing scheduler effect. Furthermore, the size of the input-output channel is exponential in the bit sizes of the secret and the output [7].

Chen and Malacaria [11] discuss quantitative analysis of a multi-threaded language based on a probabilistic scheduler. They analyze the effect of probabilistic schedulers and leakage in intermediate states. They transform the multi-threaded program into an equivalent single-threaded looping program and execute all possible interleavings of the threads in a single thread. This transformation makes the runtime of the program too long. They compute the expected leakage and channel capacity of some example programs. Channel capacity is an upper bound of leakage over *all* possible distributions of the secret input. This is different from the expected or the maximum leakages, for which a *prior* distribution on the secret input is assumed.

Chen and Malacaria [6] use Bellman's optimality principle to determine the minimal and maximal leakages for multi-threaded programs. They model these programs as state transition systems. Their method is confined to the minimal and maximal leakages and does not deal with other variants of leakage such as the expected leakage or the bounded time leakage.

Meng and Smith [18] compute channel capacity of deterministic imperative programs using min-entropy. They use STP solver to determine for every two bits in the output, which of the four possible bit values are feasible. These two-bit patterns give an estimation of the capacity iin deterministic programs. We have discussed the approach for shared-memory multi-threaded programs, but since the PRISM language supports randomization, nondeterminism, and synchronization, the approach, without any change, is applicable to randomized programs, synchronized parallel programs, and nondeterministic programs. For more information on randomization, synchronization, and nondeterminism support in PRISM, please refer to the PRISM language manual, available at http://prismmodelchecker.org/manual/ThePRISMLanguage.

Ngo and Huisman [8] propose an approach for quantitative information flow analysis of multi-threaded programs. They discuss the effect of schedulers and leakage in intermediate states. They model programs with a special variant of probabilistic Kripke structures, a state transition system similar to Markovian processes. In this structure, each state is labeled with a probability distribution, showing *possible values* of the secret variable in that state. This is different from our model, in which each state contains *a single value* of the secret variable. Ngo and Huisman compute the expected leakage using probability distributions of final states of the Kripke structure. They assume the program model contains such distributions, but do not explain how to construct such a model. They have not implemented their approach. They also assume a uniform distribution on the secret (i.e., ignorant attacker), while in our approach the prior knowledge can be any probability distribution (i.e., probabilistic attacker). Another difference is that they do not discuss the bounded time, maximum, or minimum leakages.

Phan et al. [10] run symbolic execution to extract all symbolic paths of a deterministic Java program and use a model counting technique to compute channel capacity. They have implemented their approach into a tool, called QILURA.

Boreale et al. [17] discuss how to compute min-entropy capacity of a system modeled as a channel. They assume an attacker who runs the program multiple times and tries to guess the secret in just one try. They show that when the secret values have a uniform distribution, min-entropy capacity of the system converges asymptotically to the logarithm of the number of distinct rows in the channel matrix. Their work is different from ours, since we model programs as Markovian processes and compute the expected leakage, not capacity. Computing min-entropy capacity is left as future work.

Americo et al. [2] incorporate probabilistic model checking and in particular PCTL (Probabilistic Computational Tree Logic) to derive channel capacity for different cases of the dining cryptographers and Crowds protocols. They have implemented their approach using the PRISM model checker. PCTL has logical operators that allow PRISM to calculate reachability probabilities of certain events occurring. Americo et al. use this feature of PCTL to compute posterior distributions, from which channel capacity is calculated.

A closely-related field to quantitative information flow is qualitative information flow, in which a program is classified as insecure if it has any amount of leakage, even a minor one [19]. In qualitative information flow, a security property is defined to specify secure information flow and a verification technique is used to check whether the property holds or not.

There are various properties in the literature, such as probabilistic noninterference [20] or observational determinism [21]. These properties are too restrictive and classify many intuitively secure programs as insecure. For example, they classify a password checking program as insecure, as it leaks information about what the password is *not*.

Automated tools for computing the expected leakage.. There are various automated tools for computing the expected leakage of different kinds of programs. Here we introduce QUAIL [1,22], LeakWatch [23], Moped-QLeak [9], and HyLeak [24] and compare them with our approach qualitatively. In Section 5.1 we will compare them quantitatively. None of these tools, except LeakWatch, support concurrency and none of them, including LeakWatch, consider the leakage in intermediate states. Furthermore, they only compute the expected leakage, not bounded time leakage values, or maximum and minimum leakages. Another point worthy of mentioning here is that our approach, QUAIL, and Moped-QLeak compute the exact value for the expected leakage, while LeakWatch and HyLeak estimate its value.

Biondi et al. [7] propose an approach for quantifying the information leakage of randomized protocols using Markovian processes. They have implemented their approach into a tool, called QUAIL [22] (version 1.0). QUAIL accepts an input program written in QUAIL's simple imperative language. It uses symbolic execution to construct a Markov chain representing all executions of the input program. A state in the constructed Markov chain contains a value for each public variable and a set of values for each secret variable. It reduces the model by hiding all internal states and creating three quotients of the reduced model: the attacker's, the secret's, and the joint quotients. It then computes the expected leakage as the sum of entropies of the attacker's and the secret's quotients minus entropy of the joint quotient. Biondi et al. define entropy of each quotient as the expected value of local entropies of each state. They also discuss the bounded leakage problem in [15]. In QUAIL version 2 [1], Biondi et al. introduce a trace-based approach for computing the expected leakage. They use this approach to find final states, group them based on final values of the public variable, and compute the remaining uncertainty. This approach is much faster than the approach introduced in QUAIL version 1.0 [7]. A difference of our approach with that of QUAIL version 2 is that they only consider leakages at final states, whereas we consider intermediate steps, too. Another difference is that in our Markov model of the program each state contains a value for each public variable and *a value* for each secret variable, not *a set of values* for each secret variable. This difference does not affect the analysis results, but makes our leakage computation method quite different from that of QUAIL version 2. Besides, QUAIL v2 does not support concurrency and only computes the expected leakage.

LeakWatch [23] approximates the expected leakage of a Java program by executing it repeatedly for each possible value of the secret. It estimates min-entropy leakage from trial executions of the program, providing 95% confidence intervals using Pearson's χ^2 testes. It also supports leakage estimation using *mutual information*, which is another widely-used measure of information leakage. Intuitively, mutual information measures the amount of information shared between two variables (here, secret and public variables). Due to Java's support for multi-threading, LeakWatch can estimate the expected leakage of multi-threaded programs. However, it does not consider leakage in intermediate states. LeakWatch is based on point-to-point information leakage model of Chothia et al. [25] in which secret and public variables may occur at any given points in the program and the leakage is computed between these points. In spite of the fact that our model considers leakage in intermediate states, it computes the expected leakage between the initial states (start of the program) and final states (end of the program), not between given states (points) of the program. Point-to-point leakage model of Chothia et al. is comparable to our bounded time leakage function $\mathcal{L}_{(t_1,t_2)}$ (Eq. (4)) which computes the leakage from a time (step) t_1 to a time $t_2 > t_1$ in the program model.

Moped-QLeak [9] uses symbolic algorithms based on Binary Decision Diagrams (BDDs) to extract the relation between the inputs and outputs of the program. It employs this relation for precise computation of the expected leakage using Shannon entropy, as well as min-entropy. To obtain the relation between the inputs and outputs of the program, Moped-QLeak uses Moped, a BDD-based symbolic model checker. The Moped-QLeak tool processes programs written in the Remopla language, an input language of the Moped model checker. Due to the use of BDDs, Moped-QLeak is a scalable tool and can handle programs with large state space. However, its support for probabilistic programs is limited and cannot analyze most of the anonymity protocols. It also computed different leakage values for the Shannon entropy and min-entropy in most of the programs we analyzed. For instance, as discussed in the next Section, for the Single Preference voting protocol with a uniform prior distribution on the secret, Moped-QLeak computes different values for Shannon and min-entropy leakages.

HyLeak [24] takes as input a slight extension of the QUAIL's imperative language and computes its expected leakage using Shannon entropy. It divides the input program code into terminal components and analyzes them by either precise or statistical approximation analysis. The analysis results are used to compute an approximate joint probability distribution of the secret and public variables, by which the Shannon leakage is computed. The statistical approximation is done for improving the scalability of the approach. A limitation of HyLeak and also QUAIL is that the prior distribution on the secret variable is always assumed to be uniform. This is while in our approach the prior distribution is not limited and is a parameter of the algorithm.

5.1. Quantitative comparison with the other leakage tools

We evaluate our approach and the tools QUAIL version 2 [1], LeakWatch [23], Moped-QLeak [9], and HyLeak [24] with the case studies described in the previous Section. In order to compare them, we consider three criteria: accuracy, speed

Table 3
Comparison of PRISM-Leak to the other tools. Expected leakage and runtime in seconds for the Smith–Volpano program and a uniform distribution on PIN. Timeout is set to five minutes.

n	Expected leakage			Runtime	
	LeakWatch [23]		PRISM-Leak	LeakWatch [23]	PRISM-Leak
	Min-entropy	Mutual information			
2	2 (100%)	1.97 (98%)	**2 (100%)**	1.8	**0.3**
⋮	⋮	⋮	⋮	⋮	⋮
8	2 (25%)	1.898 (23%)	**2 (25%)**	36.4	**0.8**
9	2 (22%)	1.897 (21%)	**2 (22%)**	73.8	**2.1**
10	2 (20%)	1.9 (19%)	**2 (20%)**	148.7	**12.8**
11	2 (18%)	1.892 (17%)	**2 (18%)**	273.1	**65.1**
12	timeout	timeout	**timeout**	timeout	**timeout**

(runtime), and scalability. The experiments were run on a laptop with an Intel Core i7-2640M CPU @ 2.80GHz × 2 and 8 GB RAM. For measuring time values, we ran each scenario five times and computed the mean value.

Note that these tools only compute the expected leakage, not bounded time leakage values, or maximum and minimum leakages. This is why we only compare the expected leakage results. Since QILURA [10] computes channel capacity and not expected leakage, it has not been included in the comparisons.

5.1.1. The Smith–Volpano program

QUAIL, Moped-QLeak, and HyLeak do not support any kind of concurrency and hence are omitted from the comparisons for the Smith–Volpano program. The Java source code of LeakWatch and the PRISM description of our approach for the Smith–Volpano program are given in Appendix A.

Accuracy. Table 3 shows a comparison of the expected leakage values computed by PRISM-Leak and LeakWatch for the Smith–Volpano program and an attacker with a uniform prior knowledge on the secret. Here, percentages denote the percentage of the secret leaked by the program. PRISM-Leak computes the exact values. LeakWatch, when using min-entropy, estimates the exact values, but it underestimates the leakage, by 1 to 2%, when using mutual information. Note that PRISM-Leak outputs the same leakage values when using Shannon entropy or min-entropy.

Speed. In Table 3 the time values needed by PRISM-Leak and LeakWatch for the Smith–Volpano program are shown. The results show that PRISM-Leak is faster than LeakWatch.

Scalability. Concerning the scalability, we see that PRISM-Leak and LeakWatch both time out at $n = 12$. Recall that PRISM-Leak computes the exact values, but LeakWatch provides an approximation of the expected leakage by running the program repeatedly. For instance, for $n = 2$ it executed the program 216 times, or for $n = 11$ it executed the program 41,088 times.

Leakage in intermediate states. Consider the Smith–Volpano program, composed of four parallel threads: `[Alpha || Beta || Gamma || Delta]` (the scenario discussed in the end of Section 4.1). We implement this scenario using PRISM-Leak and LeakWatch. PRISM-Leak correctly computes the expected leakage to be n. Since, LeakWatch does not consider intermediate values of `result`, it computes the mutual information to be 0 and gives timeout for the min-entropy leakage.

5.1.2. The single preference voting protocol

The Single Preference voting protocol is a highly-studied randomized protocol and thus a suitable case study for comparing the performance of various information leakage tools. The source code of QUAIL, LeakWatch, Moped-QLeak, and HyLeak for the Single Preference protocol are available at [1]. The PRISM description is given in Appendix A.

Accuracy. Table 4 shows a comparison of the expected leakage values computed by various tools for the Single Preference protocol. The attacker is assumed to have a uniform prior knowledge on the secret. As with Table 3, percentages denote the percentage of the secret leaked by the protocol. Since Shannon entropy and min-entropy coincide on the uniform distribution, the tools should compute the same values for both criteria. However, Moped-QLeak computes different leakage values for the Shannon entropy and min-entropy. This is why there are two columns for Moped-QLeak in Table 4. LeakWatch also computes different leakage values for the min-entropy and mutual information.

PRISM-Leak (with both Shannon and min-entropy measures), QUAIL, Moped-QLeak with Shannon entropy, and HyLeak compute the exact values. Interestingly, HyLeak computes the exact values for this case study. Moped-QLeak with min-entropy has 2 to 5% error in computing the expected leakage. This is while Moped-QLeak with Shannon entropy computes the exact values, without any error. LeakWatch with mutual information underestimates the leakage, by 1 to 2%. It also has 2 to 5% error in computing the leakage using min-entropy. Note that LeakWatch uses simulation, i.e., random sampling

Table 4
Accuracy comparison of PRISM-Leak to the other tools. Expected leakage for the Single Preference protocol and an attacker with a uniform prior knowledge on the secret. Timeout is set to five minutes.

c	n	QUAIL [1]	LeakWatch [23]		Moped-QLeak [9]		HyLeak [24]	PRISM-Leak
			Min-entropy	Mutualinformation	Shannon	Min-entropy		
4	4	4.815	5.129 (64%)	4.69 (58%)	4.815	5.129 (64%)	4.815	**4.815 (60%)**
	5	5.352	5.807 (58%)	5.215 (52%)	5.352	5.807 (58%)	5.352	**5.352 (53%)**
	6	5.792	timeout	timeout	5.792	6.392 (53%)	5.792	**5.792 (48%)**
	7	6.164	timeout	timeout	6.164	6.906 (49%)	6.164	**6.164 (44%)**
5	4	5.838	6.129 (66%)	5.704 (61%)	5.838	6.129 (66%)	5.838	**5.838 (62%)**
	5	6.546	timeout	timeout	6.546	6.977 (60%)	6.546	**6.546 (56%)**
	6	7.135	timeout	timeout	7.135	7.714 (55%)	7.135	**7.135 (51%)**
6	4	6.709	timeout	timeout	6.709	6.977 (67%)	6.709	**6.709 (64%)**
	5	7.574	timeout	timeout	7.574	7.977 (61%)	7.574	**7.574 (58%)**
	6	8.303	timeout	timeout	8.303	8.851 (57%)	8.303	**8.303 (53%)**
7	4	7.468	timeout	timeout	7.468	7.714 (68%)	7.468	**7.468 (66%)**
	5	8.477	timeout	timeout	8.477	8.851 (63%)	8.477	**8.477 (60%)**
8	4	8.139	timeout	timeout	8.139	8.366 (69%)	8.139	**8.139 (67%)**
	5	9.28	timeout	timeout	9.28	9.629 (64%)	9.28	**9.28 (61%)**

Table 5
Runtime comparison of PRISM-Leak to the other tools. Runtime in seconds for the Single Preference protocol. Timeout is set to five minutes.

c	n	QUAIL [1]	LeakWatch [23]	Moped-QLeak [9]	HyLeak [24]	PRISM-Leak
4	4	1.1	18.3	1.2	1.1	**0.5**
	5	2.6	108.2	1.5	2.1	**1.1**
	6	12.8	timeout	1.8	4.9	**7.3**
	7	191.4	timeout	2.3	10.6	**173.2**
	8	timeout	timeout	2.7	37.8	**timeout**
5	4	1.6	83.4	1.3	1.3	**0.6**
	5	6.6	timeout	1.9	3.5	**2.2**
	6	59	timeout	2.6	10.7	**27.4**
	7	timeout	timeout	3.9	49.4	**timeout**
6	4	2.5	293.6	1.8	2	**0.9**
	5	12.6	timeout	2.8	6.7	**4.1**
	6	235.8	timeout	4.8	32.5	**96**
	7	timeout	timeout	8.6	248.8	**timeout**
7	4	3.9	timeout	2.7	3	**2**
	5	25.9	timeout	5.5	13.8	**8.9**
	6	timeout	timeout	11.4	114.9	**timeout**
8	4	6.2	timeout	4.4	4.7	**1.6**
	5	50.5	timeout	10.5	29.6	**19.6**
	6	timeout	timeout	36	timeout	**timeout**

of execution paths and thus it may compute a slightly different value for each run of the tool. An interesting observation in this table is that LeakWatch with min-entropy computes exactly the same values as Moped-QLeak with min-entropy. Another interesting observation is that PRISM-Leak computes the same values as QUAIL. This demonstrates that in our implementation of the Single Preference voting protocol there are no intermediate leakages.

Speed. We compare the execution time of PRISM-Leak and the other four tools for the Single Preference protocol in Table 5. The results show that PRISM-Leak is faster than QUAIL (version 2) and LeakWatch. Moped-QLeak and HyLeak are faster than PRISM-Leak, but as c grows, running time of PRISM-Leak gets closer to HyLeak and in $c = 8$, PRISM-Leak outperforms HyLeak.

Scalability. Finally, we evaluate the scalability of PRISM-Leak and the other four tools. To compare the scalability we observe the maximum number of voters each tool can handle, before it times out or gives an error. For example, for $c = 4$ PRISM-Leak and QUAIL time out at 8 voters, LeakWatch stops at 6, and HyLeak times out at 10, but Moped-QLeak outputs `-inf` at 10, which seems a precision error. The results in Table 5 show that PRISM-Leak is as scalable as QUAIL, more scalable than LeakWatch and comparable to HyLeak. However, Moped-QLeak is the most scalable and outperforms PRISM-Leak. Of course, for the cases $c = 4$ with $n = 10$, $c = 5$ with $n = 11$, and $c = 6$ with $n = 11$ Moped-QLeak does not stop due to timeout, but outputs `-inf`.

Table 6
Overall comparison of PRISM-Leak to the related work. The scalability ranks are given according to the results obtained for the single preference voting protocol.

Approach	expected leakage	scalability rank	intermediate leakage	concurrency support	bounded time leakage	min max leakage	channel capacity
Chen and Malacaria [11]	exact	✗	✓	✓	✗	✗	✓
Chen and Malacaria [6]	✗	✗	✓	✓	✗	✓	✗
Meng and Smith [18]	✗	✗	✗	✗	✗	✗	✓
Ngo and Huisman [8]	exact	✗	✓	✓	✗	✗	✗
Phan et al. [10]	✗	✗	✗	✗	✗	✗	✓
Boreale et al. [17]	✗	✗	✗	✗	✗	✗	✓
Biondi et al. [15]	exact	✗	✗	✗	✓	✗	✗
Americo et al. [2]	✗	✗	✗	✗	✗	✗	✓
QUAIL [1]	exact	3	✗	✗	✗	✗	✗
LeakWatch [23]	estimated	4	✗	✓	✗	✗	✗
Moped-QLeak [9]	exact*	1	✗	✗	✗	✗	✗
HyLeak [24]	estimated*	2	✗	✗	✗	✗	✗
PRISM-Leak	**exact**	**3**	✓	✓	✓	✓	✗

* Moped-QLeak computes the exact value of the expected leakage, but in some cases yields two different values for Shannon and min-entropy leakages.
** HyLeak uses a combination of exact and estimation methods. So, in some cases it computes the exact values, but in most cases it estimates them.

An overall comparison of PRISM-Leak to the related work that we discussed in this section is given in Table 6. A scalability rank is given to each of the five tools that were used to analyze the single preference voting protocol. These ranks are given based on the runtime values of Table 5. Moped-QLeak has the highest rank (1) and LeakWatch has the lowest rank (4).

PRISM is a symbolic model checker and model construction time varies according to the order in which the variables of the model appear in the model file. There are some heuristics for good variable ordering. For instance, variables related to most of other variables should appear at first, or variables related to each other should appear close to each other.

We also implemented sequential version of the Single Preference protocol in PRISM, using just one module. The following is the PRISM description of the module for $c = 2$ and $n = 3$.

```
module voting
 s : [0.n]; // module status
 // count the votes
 [] s=0 & vot1 = 1 & result1<n -> (result1'=result1+1) & (s'=1);
 [] s=0 & vot1 = 2 & result2<n -> (result2'=result2+1) & (s'=1);
 [] s=1 & vot2 = 1 & result1<n -> (result1'=result1+1) & (s'=2);
 [] s=1 & vot2 = 2 & result2<n -> (result2'=result2+1) & (s'=2);
 [] s=2 & vot3 = 1 & result1<n -> (result1'=result1+1) & (s'=3);
 [] s=2 & vot3 = 2 & result2<n -> (result2'=result2+1) & (s'=3);
endmodule
```

Since sequences of values of `result1` and `result2` reveals who voted whom, PRISM-Leak computes the expected leakage to be 3 bits (100.0%).

Note that we compared our results with QUAIL version 2, which is much faster and more scalable than QUAIL version 1.0 [22]. We performed some experiments with QUAIL version 1.0: for $c = 4$ and $n = 4$ it took 73.676 seconds to compute the leakage and for $c = 4$ and $n = 5$ it timed out.

6. Conclusions and future work

We proposed an automated approach to quantify the information leakage of multi-threaded programs. The approach assumes a probabilistic attacker capable of observing the internal behavior of the programs and models the programs by Markovian processes. The program model contains all execution traces that the attacker may observe. Using this program model we computed the expected leakage, bounded time leakage, maximum and minimum leakages. The main contribution of the paper is computing the leakage variants for multi-threaded programs, taking into account the effect of schedulers and leakage in intermediate states. Two case studies were analyzed to show applicability and feasibility of the proposed approach. We implement the approach into a tool PRISM-Leak and compared it to the other leakage tools. The experimental results showed that PRISM-Leak computes the exact values for the expected leakage and its speed and scalability is comparable to the other automated tools.

We defined maximum leakage as an upper bound of leakage for an ignorant attacker. We are working on quantifying channel capacity, which is an upper bound of leakage for all attackers with the same observational power but different prior knowledge. We also aim to implement the proposed approach symbolically using binary decision diagrams, which enables

handling of very large program models and improves scalability of the approach. Another interesting future work would be to use Probabilistic Computation Tree Logic for computing posterior probabilities of the secret variable. We also plan to incorporate statistical approximation methods into our trace-based approach in order to estimate the expected leakage faster.

Acknowledgments

The authors would like to thank Khayyam Salehi for useful feedback and comments. We are also grateful of Alireza Kandeh for his help in modifying the PRISM parser and Joachim Klein for his valuable comments on how to implement the approach in PRISM.

Appendix A

The PRISM description for the Smith–Volpano program in the case where the initial value of `mask` equals to 2 is given below.

```
dtmc
const int n = 3; // num of bits of the pin variable
global result : [0.pow(2, n)-1];
global mask : [0.pow(2, n)-1];
global pin : [0.pow(2, n)-1];
global trigger0 : [0.1];
global trigger1 : [0.1];
global maintrigger : [0.2];
global turn : [1.3];
module Alpha
 c1 : [0.5];
 [] turn=1 & c1=0 & mask!=0 -> (c1'=1);
 [] turn=1 & c1=1 & trigger0=0 & trigger1=1 -> (turn'=2);
 [] turn=1 & c1=1 & trigger0=0 & trigger1!=1 -> (turn'=3);
 [] turn=1 & c1=1 & trigger0!=0 -> (c1'=2);
 [] turn=1 & c1=2 & mod(floor(result/mask),2)=0 &
     result+mask<=pow(2,n)-1 -> (result'=result+mask) & (c1'=3);
 [] turn=1 & c1=2 & mod(floor(result/mask),2)=1 -> (c1'=3);
 [] turn=1 & c1=3 -> (trigger0'=0) & (c1'=4);
 [] turn=1 & c1=4 & maintrigger<2 -> (maintrigger'=maintrigger+1) & (c1'=5);
 [] turn=1 & c1=5 & maintrigger=1 -> (trigger1'=1) & (c1'=0);
 [] turn=1 & c1=5 & maintrigger!=1 -> (c1'=0);
endmodule
module Beta
 c2 : [0.5];
 [] turn=2 & c2=0 & mask!=0 -> (c2'=1);
 [] turn=2 & c2=1 & trigger1=0 & trigger0=1 -> (turn'=1);
 [] turn=2 & c2=1 & trigger1=0 & trigger0!=1 -> (turn'=3);
 [] turn=2 & c2=1 & trigger1!=0 -> (c2'=2);
 [] turn=2 & c2=2 & mod(floor(result/mask),2)=1 -> (result'=result-mask) & (c2'=3);
 [] turn=2 & c2=2 & mod(floor(result/mask),2)=0 -> (c2'=3);
 [] turn=2 & c2=3 -> (trigger1'=0) & (c2'=4);
 [] turn=2 & c2=4 & maintrigger<2 -> (maintrigger'=maintrigger+1) & (c2'=5);
 [] turn=2 & c2=5 & maintrigger=1 -> (trigger0'=1) & (c2'=0);
 [] turn=2 & c2=5 & maintrigger!=1 -> (c2'=0);
endmodule
module Gamma
 c3 : [0.6];
 [] turn=3 & mask!=0 & c3=0 -> (c3'=1);
 [] turn=3 & mask=0 & c3=0 -> (c3'=5);
 [] turn=3 & c3=1 -> (maintrigger'=0) & (c3'=2);
 [] turn=3 & c3=2 & mod(floor(pin/mask),2)=0 -> (trigger0'=1) & (c3'=3);
 [] turn=3 & c3=2 & mod(floor(pin/mask),2)=1 -> (trigger1'=1) & (c3'=3);
 [] turn=3 & c3=3 & maintrigger=2 -> (c3'=4);
 [] turn=3 & c3=3 & maintrigger!=2 -> 0.5:(turn'=1) + 0.5:(turn'=2);
 [] turn=3 & c3=4 -> (mask'=floor(mask/2)) & (c3'=0);
 [] turn=3 & c3=5 -> (trigger0'=1) & (c3'=6);
 [] turn=3 & c3=6 -> (trigger1'=1) ;
endmodule
init
 mask=2 & result=0 & maintrigger=0 & trigger0=0 & trigger1=0 & c1=0 & c2=0 & c3=0 & turn=3
endinit
```

The PRISM description for the Single Preference protocol in the case where $c = 4$ and $n = 4$ is given below.

```
dtmc
const int c = 4; // number of candidates
const int N = 4; // number of voters
// state and preference of each voter
global s1 : [0.1];
global secret vot1 : [1.c];
global s2 : [0.1];
global secret vot2 : [1.c];
global s3 : [0.1];
global secret vot3 : [1.c];
global s4 : [0.1];
global secret vot4 : [1.c];
// number of votes for each candidate: public variables
global observable result1 : [0.N];
global observable result2 : [0.N];
global observable result3 : [0.N];
global observable result4 : [0.N];
module voter1 // module for first voter1
 // voter1 voted to candidate 1
 [] s1=0 & vot1 = 1 & result1<N -> (result1'=result1+1) & (s1'=1);
 // voter1 voted to candidate 2
 [] s1=0 & vot1 = 2 & result2<N -> (result2'=result2+1) & (s1'=1);
 // voter1 voted to candidate 3
 [] s1=0 & vot1 = 3 & result3<N -> (result3'=result3+1) & (s1'=1);
 // voter1 voted to candidate 4
 [] s1=0 & vot1 = 4 & result4<N -> (result4'=result4+1) & (s1'=1);
endmodule
// construct further voters with renaming
module voter2 = voter1 [ vot1=vot2, s1=s2 ] endmodule
module voter3 = voter1 [ vot1=vot3, s1=s3 ] endmodule
module voter4 = voter1 [ vot1=vot4, s1=s4 ] endmodule
// set of initial states: ''vot1, {\ldots}, votN'' can be anything
init s1=0 & s2=0 & s3=0 & s4=0 & result1=0 & result2=0 & result3=0 & result4=0 endinit
```

The Java source code of LeakWatch for the Smith–Volpano program in the case where the initial value of `mask` equals to 2 is given below.

```
import bham.leakwatch.LeakWatchAPI;
import java.security.SecureRandom;
public class SmithVolpano {
 static int maintrigger=0, trigger0=0, trigger1=0, result=0, mask = 2;
 public static void main (String [] args) throws Exception {
  int n = 3; // num of bits of the pin variable
  int PIN = new SecureRandom().nextInt((int)Math.pow(2,n)); // secret variable
  LeakWatchAPI.secret(''PIN'',PIN);
  Thread alpha = new Thread(new Runnable() {
   @Override
   public synchronized void run() {
    while (mask != 0) {
     while (trigger0 == 0)
      try{ Thread.sleep(1); } catch(Exception e){ System.err.println(e); }
     result = result | mask; trigger0 = 0; maintrigger ++;
     if (maintrigger == 1) trigger1 = 1;
  } } });
  Thread beta = new Thread(new Runnable() {
   @Override
   public synchronized void run() {
    while (mask != 0) {
     while (trigger1 == 0)
      try{ Thread.sleep(1); } catch(Exception e){ System.err.println(e); }
     result = result & ~mask; trigger1 = 0; maintrigger ++;
     if (maintrigger == 1) trigger0 = 1;
  } } });
  Thread gamma = new Thread(new Runnable() {
   @Override
   public synchronized void run() {
    while (mask != 0) {
     maintrigger = 0;
     if ((PIN & mask) == 0) trigger0 = 1; else trigger1 = 1;
     while (maintrigger < 2)
      try{ Thread.sleep(1); } catch(Exception e){ System.err.println(e); }
     mask = mask/2;
    }
    trigger0 = 1; trigger1 = 1;
  } });
  gamma.start (); alpha.start (); beta.start ();
  gamma.join (); alpha.join (); beta.join ();
  LeakWatchAPI.observe(result); // public variable
}}
```

References

[1] Biondi F, Legay A, Quilbeuf J. Comparative analysis of leakage tools on scalable case studies. In: Proceedings of the twenty second international symposium on model checking software - volume 9232. In: SPIN 2015. Berlin, Heidelberg: Springer-Verlag; 2015. p. 263–81. doi:10.1007/978-3-319-23404-5_17.

[2] Américo A, Vaz A, Alvim MS, Campos SVA, McIver A. Formal analysis of the information leakage of the dc-nets and crowds anonymity protocols. In: Cavalheiro S, Fiadeiro J, editors. Formal methods: foundations and applications. Cham: Springer International Publishing; 2017. p. 142–58.

[3] Heusser J, Malacaria P. Quantifying information leaks in software. In: Proceedings of the twenty sixth annual computer security applications conference, ACSAC. New York, NY, USA: ACM; 2010. p. 261–9. doi:10.1145/1920261.1920300.

[4] Biondi F, Enescu MA, Heuser A, Legay A, Meel KS, Quilbeuf J. Scalable approximation of quantitative information flow in programs. In: Dillig I, Palsberg J, editors. Verification, model checking, and abstract interpretation. Cham: Springer International Publishing; 2018. p. 71–93.

[5] Smith G. On the foundations of quantitative information flow. In: Proceedings of the twelfth international conference on foundations of software science and computational structures - volume 5504. Berlin, Heidelberg: Springer-Verlag; 2009. p. 288–302. doi:10.1007/978-3-642-00596-1_21.

[6] Chen H, Malacaria P. The optimum leakage principle for analyzing multi-threaded programs. In: Proceedings of the fourth international conference on information theoretic security, ICITS. Berlin, Heidelberg: Springer-Verlag; 2010. p. 177–93.

[7] Biondi F, Legay A, Malacaria P, Wasowski A. Quantifying information leakage of randomized protocols. Theor. Comput. Sci. 2015;597(C):62–87. doi:10.1016/j.tcs.2015.07.034.

[8] Ngo TM, Huisman M. Quantitative security analysis for multi-threaded programs. In: Proceedings of the eleventh international workshop on quantitative aspects of programming languages and systems, QAPL. Rome, Italy; 2013. p. 34–48. March 23–24. doi: 10.4204/EPTCS.117.3.

[9] Chadha R, Mathur U, Schwoon S. Computing information flow using symbolic model-checking. In: Raman V, Suresh SP, editors. Proceedings of the thirty fourth international conference on foundation of software technology and theoretical computer science (FSTTCS). Leibniz International Proceedings in Informatics (LIPIcs), 29. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2014. p. 505–16. doi:10.4230/LIPIcs.FSTTCS.2014.505.

[10] Phan Q-S, Malacaria P, Păsăreanu CS, d'Amorim M. Quantifying information leaks using reliability analysis. In: Proceedings of the international SPIN symposium on model checking of software. In: SPIN 2014. New York, NY, USA: ACM; 2014. p. 105–8. doi:10.1145/2632362.2632367.

[11] Chen H, Malacaria P. Quantitative analysis of leakage for multi-threaded programs. In: Proceedings of the workshop on programming languages and analysis for security, PLAS. New York, NY, USA: ACM; 2007. p. 31–40. doi:10.1145/1255329.1255335.

[12] Baier C, Katoen J. Principles of model checking. MIT Press; 2008.

[13] Kwiatkowska M, Norman G, Parker D. Prism 4.0: verification of probabilistic real-time systems. In: Proceedings of the twenty third international conference on computer aided verification. Berlin, Heidelberg: Springer-Verlag; 2011. p. 585–91.

[14] Mario SA, Chatzikokolakis K, Palamidessi C, Smith G. Measuring information leakage using generalized gain functions. In: Proceedings of the IEEE twenty fifth computer security foundations symposium. IEEE; 2012. p. 265–79.

[15] Biondi F, Legay A, Nielsen BF, Malacaria P, Wasowski A. Information leakage of non-terminating processes. In: Proceedings of the IARCS annual conference on foundations of software technology and theoretical computer science. Delhi, India; 2014.

[16] Smith G, Volpano D. Secure information flow in a multi-threaded imperative language. In: Proceedings of the twenty fifth ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL. New York, NY, USA: ACM; 1998. p. 355–64. doi:10.1145/268946.268975.

[17] Boreale M, Pampaloni F, Paolini M. Asymptotic information leakage under one-try attacks. Math. Struct. Comput. Sci. 2015;25(2):292–319. doi:10.1017/S0960129513000613.

[18] Meng Z, Smith G. Calculating bounds on information leakage using two-bit patterns. Proceedings of the ACM SIGPLAN sixth workshop on programming languages and analysis for security, PLAS. New York, NY, USA: ACM; 2011. p. 1:1–1:12. 10.1145/2166956.2166957.

[19] Karimpour J, Isazadeh A, Noroozi AA. Verifying observational determinism. In: Federrath H, Gollmann D, editors. Proceedings of the thirtieth IFIP international information security conference (SEC). ICT Systems Security and Privacy Protection, AICT-455. Hamburg, Germany; 2015. p. 82–93. doi:10.1007/978-3-319-18467-8_6.

[20] Noroozi AA, Karimpour J, Isazadeh A, Lotfi S. Verifying weak probabilistic noninterference. Int. J. Adv. Comput. Sci. Appl. 2017;8(10). doi:10.14569/IJACSA.2017.081026.

[21] Noroozi AA, Karimpour J, Isazadeh A. Bisimulation for secure information flow analysis of multi-threaded programs. Math. Comput. Appl. 2019;24(2):64. doi:10.3390/mca24020064.

[22] Biondi F, Legay A, Traonouez L-M, Wasowski A. Quail: a quantitative security analyzer for imperative code. In: Proceedings of the twenty fifth international conference on computer aided verification - volume 8044. In: CAV 2013. New York, NY, USA: Springer-Verlag New York, Inc.; 2013. p. 702–7. doi:10.1007/978-3-642-39799-8_49.

[23] Chothia T, Kawamoto Y, Novakovic C. LeakWatch: estimating information leakage from java programs. In: Proceedings of the nineteenth European symposium on research in computer security, ESORICS - Vol. 8713. New York, NY, USA: Springer-Verlag New York, Inc.; 2014. p. 219–36. doi:10.1007/978-3-319-11212-1_13.

[24] Biondi F, Kawamoto Y, Legay A, Traonouez L-M. HyLeak: hybrid analysis tool for information leakage. In: Proceedings of the fifteenth international symposium on automated technology for verification and analysis, ATVA. Pune, India; 2017. p. 14.

[25] Chothia T, Kawamoto Y, Novakovic C, Parker D. Probabilistic point-to-point information leakage. In: Proceedings of the IEEE twenty sixth computer security foundations symposium; 2013. p. 193–205. doi:10.1109/CSF.2013.20.

Ali A. Noroozi is currently a Ph.D. candidate of Computer Science at University of Tabriz, under supervision of Dr. Jaber Karimpour and Dr. Ayaz Isazadeh. He received his M.Sc. degree in Management of Information Systems from K. N. Toosi University of Technology in 2011. His research interests are quantification of information leakage, secure information flow, and model checking.

Jaber Karimpour received his M.Sc. degree in Applied Mathematics from University of Tabriz in 2000 and his Ph.D. in Computer Systems from University of Tabriz in 2009. He is currently an associate professor in the Department of Computer Science at University of Tabriz. His research interests include network security, formal specification, and verification.

Ayaz Isazadeh is a professor in the Department of Computer Science at University of Tabriz. He received an M.Sc. degree in Electrical Engineering and Computer Science from Princeton University in 1978 and a Ph.D. degree in Computing and Information science from Queen's University in 1996. His research interests include software engineering and formal methods.