An Automated Quantitative Information Flow Analysis for Concurrent Programs

Khayyam Salehi[1][0000−0002−3379−798X], Ali A. Noroozi[2][0000−0003−1173−079X], Sepehr Amir-Mohammadian[3][0000−0002−2301−4283], and Mohammadsadegh Mohagheghi[4][0000−0001−8059−3691]

[1] Department of Computer Science, Shahrekord University, Shahrekord, Iran
kh.salehi@sku.ac.ir

[2] Department of Computer Science, University of Tabriz, Tabriz, Iran
noroozi@tabrizu.ac.ir

[3] Department of Computer Science, University of the Pacific, Stockton, CA, USA
samirmohammadian@pacific.edu

[4] Department of Computer Science, Vali-e-Asr University of Rafsanjan, Rafsanjan, Iran
mohagheghi@vru.ac.ir

Abstract. Quantitative information flow is a rigorous approach for evaluating the security of a system. It is used to quantify the amount of secret information leaked to the public outputs. In this paper, we propose an automated approach for quantitative information flow analysis of concurrent programs. Markovian processes are used to model the behavior of these programs. To this end, we assume that the attacker is capable of observing the internal behavior of the program and propose an equivalence relation, back-bisimulation, to capture the attacker's view of the program behavior. A partition refinement algorithm is developed to construct the back-bisimulation quotient of the program model and then a quantification method is proposed for computing the information leakage using the quotient. Finally, an anonymous protocol, dining cryptographers, is analyzed as a case study to show applicability and scalability of the proposed approach.

Keywords: Information leakage · protocol security · quantitative information flow · confidentiality · Markovian Processes.

1 Introduction

Secure information flow is a rigorous technique for evaluating security of a system. A system satisfies confidentiality requirements if it does not *leak* any secret information to its public outputs. However, imposing no leakage policy is too restrictive and in practice the security policy of the system tends to permit minor leakages. For example, a password checking program leaks information about what the password is not when it shows a message indicating that user has entered a wrong password. *Quantitative information flow* has been a well-established attempt to overcome this deficiency. Given a system with *secret* (high confidentiality) inputs and *public* (low confidentiality) outputs, quantitative information flow addresses the problem of measuring the amount of *information leakage*, i.e., the amount of information that an *attacker* can deduce about the secret inputs by observing the outputs. Quantitative information flow is widely used in analyzing timing attacks [23, 24], differential privacy [1], anonymity protocols [13, 25, 29, 30], and cryptographic algorithms [18, 19].

Assume a program with a secret input and a public output. Furthermore, assume an *attacker* that executes the program and observes the public output. A common approach for measuring the amount of leaked information is to use the notion of *uncertainty* [31]. Before executing the program, the attacker has an *initial uncertainty* about the secret, which is determined by her prior knowledge of the secret. After executing the program and observing the output, she may infer information about the secret and thus her uncertainty may be reduced. This yields the following intuitive definition of the information leakage [31]: leaked information = initial uncertainty - remaining uncertainty.

In this paper, a practical and automated formal approach is proposed to quantify the information leakage of terminating concurrent programs. The approach considers leakages in intermediate states of the program executions and effect of the scheduling policy.

We assume the program has a secret input `h`, a public output `l`, and zero or more neutral variables. Neutral variables specify temporary and/or auxiliary components of the runtime program configuration that do not belong to a certain confidentiality level by nature, e.g., the stack pointer and loop indexes. `h` is fixed and does not change during program executions. This is the case in any analysis in the context of confidentiality that assumes data integrity to be out of scope, e.g., [2, 7]. We also assume that the public and neutral variables have single initial values. Furthermore, a probabilistic attacker is supposed, who has the full knowledge of source code of the concurrent program and is able to choose a scheduler and execute the program under the control of that scheduler. She can observe sequences of values of `l` during the executions, called *execution traces*. We also assume that the attacker can execute the program an arbitrary number of times and can then *guess* the value of `h` in a single attempt. This is called *one-try guessing model* [31].

In order to model the concurrent program, *Markov decision processes* (MDPs) are used. MDPs provide a powerful state transition system, capable of modeling probabilistic and nondeterministic behaviors [28]. The scheduler is assumed to be probabilistic, resolving nondeterminism in the MDP and inducing a *Markov chain* (MC). States of an MC contain values of `h`, `l`, and possible neutral variables. For computing the leakage, however, MC should capture the attacker's view of the program. The attacker, while executing the program and observing the execution traces, does not know the exact value of `h` in each step. She can only guess a set of possible values based on the executed program statements and the observed traces. She also cannot distinguish those executions of MC that have the same trace. In this regard, we define an equivalence relation for a given MC, called *back-bisimulation*, to specify these requirements of the threat model. Back-bisimulation induces a *quotient* which models the *attacker's view* of the program. A partition-refinment algorithm is proposed to compute the back-bisimulation quotient.

Each state of the back-bisimulation quotient contains a *secret distribution*, which shows possible values of `h` in that state, and thus is a determiner of the attacker's uncertainty about `h`. Each execution trace of the quotient shows a *reduction* of the attacker's uncertainty from the initial state to the final state of the trace. Therefore, secret distribution in the initial state of the quotient determines the attacker's initial uncertainty and secret distributions in the final states determine the remaining uncertainty. In the literature, uncertainty is measured based on the notion of entropy. The entropy of `h` expresses the difficulty for an attacker to discover its value. Based on the program model and the attacker's observational power, various definitions of entropy have been proposed. As Smith [31] shows, in the context of one-try guessing model, uncertainty about a random variable should be defined in terms of *Renyi's min-entropy*. This yields that the information leakage is computed as the difference of the Renyi's min-entropy of `h` in the initial state of the quotient and the expected value of the Renyi's min-entropy of `h` in the final states of the quotient.

We also show a subclass of MCs, called *Markov chains with pseudoback-bisimilar states*, in which back-bisimulation cannot correctly construct the attacker's view of the program behavior. Using back-bisimulation to handle this situation is considered a potential future work. Briefly, our contributions include

– proposing back-bisimulation equivalence, in order to capture the attacker's observation of the program,
– developing an algorithm to compute back-bisimulation quotient of an MC,
– proposing a method to compute the leakage of a concurrent program from the back-bisimulation quotient, and
– analyzing the dining cryptographers problem.

1.1 Paper Outline

The paper proceeds as follows. Section 2 provides a core background on some basics, information theory, Markovian models and probabilistic schedulers. Section 3 presents the proposed approach. It starts with introducing the program and threat models. It then formally defines back-bisimulation and discusses how to compute the program leakage. Finally, it describes how to construct the attacker's view of the program model, the back-bisimulation quotient. Section 4 concludes the paper and proposes future work. Finally, the case study and related work are discussed in Appendix A and Appendix B, respectively.

2 Background

In this section, we provide preliminary concepts and notations required for the proposed approach.

2.1 Basics

A *probability distribution* Pr over a set $\mathcal{X}$ is a function $Pr : \mathcal{X} \to [0,1]$, such that $\sum_{x \in \mathcal{X}} Pr(x) = 1$. We denote the set of all probability distributions over $\mathcal{X}$ by $\mathcal{D}(\mathcal{X})$.

Let S be a set and $\mathcal{R}$ an equivalence relation on S. For $s \in S$, $[s]_{\mathcal{R}}$ denotes the *equivalence class* of s under $\mathcal{R}$, i.e., $[s]_{\mathcal{R}} = \{s' \in S \mid s\mathcal{R}s'\}$. Note that for $s' \in [s]_{\mathcal{R}}$ we have $[s']_{\mathcal{R}} = [s]_{\mathcal{R}}$. The set $[s]_{\mathcal{R}}$ is often referred to as the $\mathcal{R}$-equivalence class of s. The *quotient space* of S under $\mathcal{R}$, denoted by $S/\mathcal{R} = \{[s]_{\mathcal{R}} \mid s \in S\}$, is the set consisting of all $\mathcal{R}$-equivalence classes. A partition for S is a set $\Pi = \{B_1, \ldots, B_k\}$ such that $B_i \neq \varnothing$ (for $0 < i \leq k$), $B_i \cap B_j = \varnothing$ (for $0 < i < j \leq k$) and $S = \cup_{0<i\leq k} B_i$. $B_i \in \Pi$ is called a *block*. $C \subseteq S$ is a *superblock* of Π if $C = B_{i_1} \cup \cdots \cup B_{i_l}$ for some $B_{i_1}, \ldots, B_{i_l} \in \Pi$. Note that for equivalence relation $\mathcal{R}$ on S, the quotient space $S/\mathcal{R}$ is a *partition* for S.

2.2 Information Theory

Let X denote a random variable with the finite set of values $\mathcal{X}$. *Vulnerability* [31] of X is defined as $Vul(\mathrm{X}) = \max\limits_{x \in \mathcal{X}} Pr(\mathrm{X} = x)$. Vulnerability is defined as the highest probability of correctly guessing the value of the variable in just a single attempt. In order to quantify information leaks, we convert this probability into bits using *Renyi's min-entropy* [31].

Definition 1. *The* ***Renyi's min-entropy*** *of a random variable* X *is given by* $\mathcal{H}_\infty(\mathrm{X}) = -\log_2 \; Vul(\mathrm{X})$.

2.3 Markovian Models

We use Markov decision processes (MDPs) to model operational semantics of concurrent programs. MDPs are state transition systems that permit both *probabilistic* and *nondeterministic* choices [28]. In any state of an MDP, a nondeterministic choice between probability distributions exists. Once a probability distribution is 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 are considered. Formally,

Definition 2. *A* ***Markov decision process (MDP)*** *is defind as a tuple* $\mathcal{M} = (S, Act, \boldsymbol{P}, \zeta, AP, V)$ *where,*

- S *is a set of states,*
- Act *is a set of actions,*
- $\boldsymbol{P} : S \to (Act \to (S \to [0,1]))$ *is a transition probability function such that for all states* $s \in S$ *and actions* $\alpha \in Act$, $\sum_{s' \in S} \boldsymbol{P}(s)(\alpha)(s') \in \{0,1\}$,
- $\zeta : S \to [0,1]$ *is an initial distribution such that* $\sum_{s \in S} \zeta(s) = 1$.
- AP *is a set of atomic propositions,*
- $V : S \to AP$ *is a labeling function.*

Atomic propositions represent simple known facts about the states. The function V labels each state with atomic propositions. An MDP $\mathcal{M}$ is called *finite* if S, Act, and AP are finite. An action α is *enabled* in state s iff $\sum_{s' \in S} \mathbf{P}(s)(\alpha)(s') = 1$. Let $Act(s)$ denote the set of enabled actions in s. 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 $Succ(s, \alpha)$. The set of *successors* of s is defined as $Succ(s) = \bigcup\limits_{\alpha \in Act(s)} Succ(s, \alpha)$. The set of successors of a set of states $\mathcal{S}$ is defined as $Succ(\mathcal{S}) = \bigcup\limits_{s \in \mathcal{S}} Succ(s)$. The set of *predecessors* of s is defined as $Pre(s) = \{s' \in S \mid s \in Succ(s')\}$. The set of labels that are associated with the predecessors of s is defined as $PreLabels(s) = \{V(s') \mid s' \in Pre(s), s' \neq s\}$.
MDP semantics. The intuitive operational behavior of an MDP $\mathcal{M}$ is as follows. At the beginning, an initial state s_0 is probabilistically chosen such that $\zeta(s_0) > 0$. 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 probabilistically according to the transition function $\mathbf{P}$. That is, with probability $\mathbf{P}(s)(\alpha)(s')$ the next state is s'.
Initial and final states. The states s with $\zeta(s) > 0$ are considered as the *initial states*. The set of initial states of $\mathcal{M}$ is denoted by $Init(\mathcal{M})$. To ensure $\mathcal{M}$ is non-blocking, we include a self-loop to each state s that has no successor, i.e., $\mathbf{P}(s)(\tau)(s) = 1$. The distinguished action label τ is used to show that the self-loop's action is not of further interest. Then, a state s is called *final* if $Succ(s) = \{s\}$. In the literature, these states are called absorbing [3]. We call them final, because in our program model they show termination of the program. The set of final states of $\mathcal{M}$ is denoted by $final(\mathcal{M})$.
Execution paths. Alternating sequences of states that may arise by resolving both nondeterministic and probabilistic choices in an arbitrary MDP $\mathcal{M}$ are called *(execution) paths*. More precisely, a finite path fragment $\hat{\sigma}$ of $\mathcal{M}$ is a finite state sequence $s_0 s_1 \ldots s_n$ such that $s_i \in Succ(s_{i-1})$ for all $0 < i \leq n$. A path σ is an infinite state sequence $s_0 s_1 \ldots s_{n-1} s_n^\omega$ such that $s_0 \in Init(\mathcal{M})$, $s_i \in Succ(s_{i-1})$ for all $0 < i \leq n$, ω denotes infinite iteration, and $s_n \in final(\mathcal{M})$, i.e., s_n^ω denotes the infinite iteration over s_n. The final state of σ, i.e. s_n, is given by $final(\sigma)$. The set of execution paths of $\mathcal{M}$ is denoted by $Paths(\mathcal{M})$. The set of finite path fragments starting in s and ending in s' is denoted by $PathFrags(s, s')$.
Traces and trace fragments. A *trace* of an execution path is the sequence of atomic propositions of the states of the path. Formally, the trace of a finite path fragment $\hat{\sigma} = s_0 s_1 \ldots s_n$ is defined as $\hat{T} = trace(\hat{\sigma}) = V(s_0)V(s_1) \ldots V(s_n)$. For a path $\sigma = s_0 s_1 \ldots$, $trace_{\ll i}(\sigma)$ is defined as the prefix of $trace(\sigma)$ up to index i, i.e., $trace_{\ll i}(\sigma) = V(s_0)V(s_1) \ldots V(s_i)$. Let $Paths(T)$ be the set of paths that have the trace T, i.e., $Paths(T) = \{\sigma \in Paths(\mathcal{M}) \mid trace(\sigma) = T\}$. We define $final(Paths(T))$ to denote the set of final states that result from the trace T, i.e., $final(Paths(T)) = \{final(\sigma) \mid \sigma \in Paths(T)\}$.

MDPs are suitable for modeling concurrent programs, but since they contain nondeterministic choices, they are too abstract to implement. We need to resolve these nondeterministic choices into probabilistic ones. The result is a Markov chain, which does not contain action and nondeterminism.

Definition 3. *A* ***(discrete-time) Markov chain (MC)*** *is a tuple* $\mathcal{M} = (S, \boldsymbol{P}, \zeta, AP, V)$ *where,*

- S *is a set of states,*
- $\boldsymbol{P} : S \times S \to [0, 1]$ *is a transition probability function such that for all states* $s \in S$, $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$,
- $\zeta : S \to [0, 1]$ *is an initial distribution such that* $\sum_{s \in S} \zeta(s) = 1$,
- AP *is a set of atomic propositions,*
- $V : S \to AP$ *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'. Note that for all states $s \in S$, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$.
Reachability probabilities. We define the probability of *reaching* a state s from an initial state in an MC $\mathcal{M}$ as $Pr(s) = \sum_{\substack{\hat{\sigma} \in PathFrags(s_0, s) \\ s_0 \in Init(\mathcal{M})}} Pr(\hat{\sigma})$, where

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

Trace probabilities. The occurrence probability of a trace T is defined as $Pr(T) = \sum_{\sigma \in Paths(T)} Pr(\sigma)$, where $Pr(\sigma = s_0 s_1 \dots s_n^{\omega}) = Pr(\hat{\sigma} = s_0 s_1 \dots s_n)$.
DAG structure of program models. We assumed that the programs always terminate and states indicate the current values of the variables and the program counter. This implies that Markovian models of every terminating program takes the form of a *directed acyclic graph (DAG)*, modulo self-loops in final states. Therefore, reachability probabilities coincide with long-run probabilities [3]. Initial states of the program are represented as roots of the DAG, and final states as leaves. Each state of a Markovian model is located at a level equal to the least distance of that state from an initial state. Level of state s is denoted by $level(s)$.

2.4 Probabilistic Schedulers

A probabilistic scheduler implements the scheduling policy of a concurrent program. It determines the order and probability of execution of threads. When a probabilistic scheduler is applied to a concurrent program, nondeterministic choices are replaced by probabilistic ones. As we modeled concurrency between threads using nondeterminism in MDP, the scheduler is used to resolve the possible nondeterminism in MDP. For demonstration purposes, it suffices to 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[5]. Formally,

[5] A rather general notion of schedulers is to let them use the full history of execution to make decisions. Here, this general definition is not needed and only makes the program model unnecessarily complex.

Definition 4. *Let $\mathcal{M} = (S, Act, \boldsymbol{P}, \zeta, AP, V)$ 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 5. *Let $\mathcal{M} = (S, Act, \boldsymbol{P}, \zeta, AP, V)$ 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 $\mathcal{M}_\delta = (S, \mathbf{P}_\delta, \zeta, AP, V)$ where $\boldsymbol{P}_\delta(s, s') = \sum\limits_{\alpha \in Act(s)} \delta(s)(\alpha).\boldsymbol{P}(s)(\alpha)(s')$*

3 The Proposed Approach

Suppose a concurrent program `P`, running under control of a scheduling policy δ. The proposed approach proceeds in three steps: (1) defining an MDP representing `P` and applying δ to the MDP to resolve the nondeterminism in the MDP (Section 3.1), (2) constructing a back-bisimulation quotient (Section 3.2), and (3) computing the leakage (Section 3.3). Finally, an algorithm for computing the back-bisimulation quotient is presented (Section 3.4).

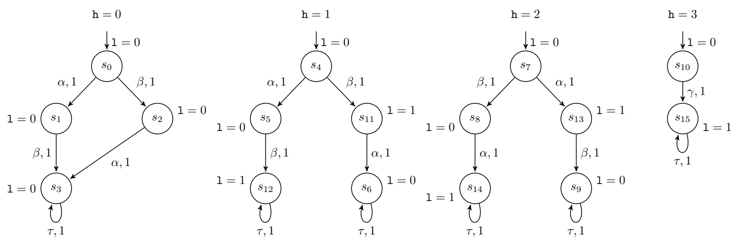
3.1 The Program and Threat Models

It is assumed `P` has a secret input variable `h` and a public output variable `l` and `h` has a fixed value during the program executions. If the program has several secret variables, they can be encoded (e.g. concatenated) into one secret variable. The same is done for public and neutral variables. Possible values of `l` and `h` are denoted by Val_l and Val_h.

The attacker has a prior knowledge of the secret, which is modeled as a prior probability distribution over the possible values of `h`, i.e. $Pr(h)$. Here, the attacker is assumed to be probabilistic, i.e., she 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 1, the probability that its value is 2 is 0.6, and the probability that its value is 3 is thrice the probability that it is 0. The prior distribution encoding these constraints is $Pr(h) = \{0 \mapsto 0.1, 2 \mapsto 0.6, 3 \mapsto 0.3\}$[6]. A special case of the probabilistic attacker is *ignorant* [6], 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`.

Define an MDP representing `P`. Operational semantics of the concurrent program `P` is represented by an MDP $\mathcal{M}^{\mathtt{P}} = (S, Act, \mathbf{P}, \zeta, Val_l, V)$. Each state $s \in S$ is a tuple $\langle \overline{l}, \overline{h}, \overline{n}, pc \rangle$, where $\overline{l}$, $\overline{h}$, and $\overline{n}$ are values of the public, secret, and neutral variables, respectively, and pc is the program counter. Actions Act are program statements of `P`. The function $\mathbf{P}$ defines probabilities of transitions between states. Atomic propositions are Val_l and the function V labels each

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



h = 0
h = 1
h = 2
h = 3
l = 0
l = 1
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15}
$\alpha, 1$
$\beta, 1$
$\gamma, 1$
$\tau, 1$

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

state with value of `l` in that state. In fact, a state label is what an attacker observes in a state and traces of $\mathcal{M}^{\texttt{P}}$ are the sequences of public values that are valid during the execution.

The initial distribution ζ is determined by the prior knowledge of the attacker $Pr(h)$, i.e., $\zeta(s) = Pr(h = \overline{h})$ for all $s \in Init(\mathcal{M}^{\texttt{P}})$, where $s = \langle ., \overline{h}, ., . \rangle$.

Example 1 (Program `P1`). Consider the following program, where `h` is a 2-bit random variable and `||` denotes the concurrency of the executions:

```
l:=0; if h=3 then l:=1  else (l:=h/2 || l:=h mod 2) (P1)
```

The attacker's prior knowledge is the size of `h`, yielding a uniform distribution on `h`, i.e., $Pr(h) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$. The MDP $\mathcal{M}^{\texttt{P1}}$ of the program is shown in Figure 1. The initial distribution ζ is determined by $Pr(h)$, i.e., $\zeta = \{s_0 \mapsto \frac{1}{4}, s_4 \mapsto \frac{1}{4}, s_7 \mapsto \frac{1}{4}, s_{10} \mapsto \frac{1}{4}\}$. 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 : `l:=h/2` and the probability $\mathbf{P}(s_0)(\alpha)(s_1) = 1$; Or the transition from s_0 to s_2 has the label β : `l:=h mod 2` and the probability $\mathbf{P}(s_0)(\beta)(s_2) = 1$.

Resolve the nondeterminism in the MDP. The scheduling policy is represented by a memoryless probabilistic scheduler δ. As the MDP $\mathcal{M}^{\texttt{P}}$ is run under the control of the scheduler δ, all nondeterministic transitions are resolved and an MC $\mathcal{M}^{\texttt{P}}_{\delta} = (S, \mathbf{P}_{\delta}, \zeta, Val_l, V)$ is produced.

Example 2 (MC of `P1`). 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 definition of the scheduler as follows:

$$uni(s_0) = uni(s_4) = uni(s_7) = \{\alpha \mapsto \frac{1}{2}, \beta \mapsto \frac{1}{2}\}, \qquad uni(s_{10}) = \{\gamma \mapsto 1\},$$
$$uni(s_1) = uni(s_5) = uni(s_{13}) = \{\beta \mapsto 1\}, \quad uni(s_2) = uni(s_8) = uni(s_{11}) = \{\alpha \mapsto 1\},$$
$$uni(s_3) = uni(s_6) = uni(s_9) = uni(s_{12}) = uni(s_{14}) = uni(s_{15}) = \{\tau \mapsto 1\}.$$

The MC $\mathcal{M}^{\texttt{P1}}_{uni}$ of the program `P1` running under control of the uniform scheduler is depicted in Figure 2. In this Figure, transitions are labeled by the transition probability.

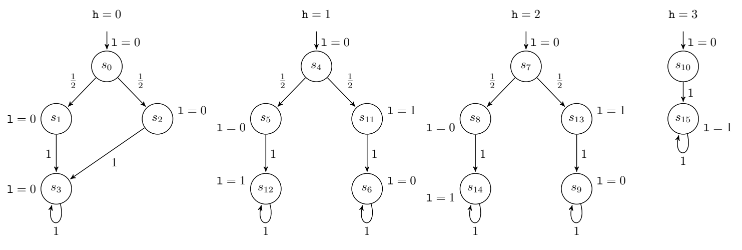


Fig. 2: $\mathcal{M}^{\mathtt{P1}}_{uni}$: MC of the program `P1` with the uniform scheduler

3.2 The Attacker's View of the Program: Back-bisimulation Quotient

In order to measure the amount of information the attacker can deduce about `h`, we need to construct the attacker's view of the program. First, the attacker can distinguish a final state from a non-final one by observing termination of the program. Second, she cannot discriminate between those paths that have the same trace. For instance, in $\mathcal{M}^{\mathtt{P1}}_{uni}$ (Figure 2) the attacker only observes the traces $\{\langle 0,0,0^\omega\rangle, \langle 0,0,1^\omega\rangle, \langle 0,1,0^\omega\rangle, \langle 0,1^\omega\rangle\}$, whereas there are seven different execution paths. The implication is that she cannot distinguish those final states that have the same public values and result from the same traces. Third, she does not know secret values in the final states, but may guess the value of `h` based on a probability distribution that she can compute according to the possible values of `h` in each final state. These three requirements are captured by an equivalence relation, called *back-bisimulation*, denoted by $\sim_b$.

Definition 6. *Let $\mathcal{M}^{\mathtt{P}}_{\delta}$ be an MC. A* ***back-bisimulation*** *for $\mathcal{M}^{\mathtt{P}}_{\delta}$ is a binary relation $\mathcal{R}$ on S such that for all $s_1\ \mathcal{R}\ s_2$, the following three conditions hold: (1) $V(s_1) = V(s_2)$, (2) if $s'_1 \in Pre(s_1)$, then there exists $s'_2 \in Pre(s_2)$ with $s'_1\ \mathcal{R}\ s'_2$, (3) if $s'_2 \in Pre(s_2)$, then there exists $s'_1 \in Pre(s_1)$ with $s'_1\ \mathcal{R}\ s'_2$.*

States s_1 and s_2 are back-bisimilar*, denoted by $s_1 \sim_b s_2$, if there exists a back-bisimulation $\mathcal{R}$ for $\mathcal{M}^{\mathtt{P}}_{\delta}$ with $s_1\ \mathcal{R}\ s_2$.*

Condition (1) requires that the states s_1 and s_2 have the same public values. According to condition (2), every incoming transition of s_1 must be matched by an incoming transition of s_2; the reverse is assured by condition (3).

Theorem 1. *Back-bisimulation is an equivalence relation.*[7]

As $\sim_b$ is an equivalence relation, it induces a set of equivalence classes on the state space of an MC. Given MC $\mathcal{M}^{\mathtt{P}}_{\delta}$, a quotient space $\mathcal{M}^{\mathtt{P}}_{\delta}/\sim_b$ captures the attacker's view of the program `P`. The MC $\mathcal{M}^{\mathtt{P}}_{\delta}/\sim_b$ aggregates same-trace paths of $\mathcal{M}^{\mathtt{P}}_{\delta}$ into one path.

[7] The proofs of the theorems have been omitted due to meet the page limit.

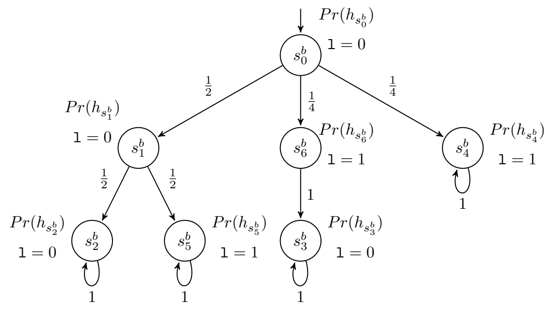


Fig. 3: $\mathcal{M}_{uni}^{\mathtt{P1}}/\sim_b$: back-bisimulation quotient of $\mathcal{M}_{uni}^{\mathtt{P1}}$

Definition 7. *For MC $\mathcal{M}_\delta^{\mathtt{P}} = (S, \boldsymbol{P}_\delta, \zeta, Val_l, V)$ and back-bisimulation $\sim_b$, the **back-bisimulation quotient** is defined by $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$ where*
$\mathcal{M}_\delta^{\mathtt{P}}/\sim_b = (S/\sim_b, \boldsymbol{P'}_\delta, s_{init}^b, Val_l, V, Pr(h))$

- *$S/\sim_b$ is the quotient space of S under $\sim_b$,*
- *$\boldsymbol{P'}_\delta : (S/\sim_b) \times (S/\sim_b) \to [0,1]$ is a probability transition function between equivalence classes of $S/\sim_b$ such that $\forall\ s^b, t^b \in S/\sim_b$*

$$\boldsymbol{P'}_\delta(s^b, t^b) = \frac{\sum\limits_{s \in s^b,\ t \in t^b} Pr(s) * \boldsymbol{P}_\delta(s,t)}{Pr(s^b)},$$

  where $Pr(s)$ and $Pr(s^b)$ are the probability of reaching to s and s^b in MC $\mathcal{M}_\delta^{\mathtt{P}}$ and $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$, respectively,
- *$s_{init}^b = Init(\mathcal{M}_\delta^{\mathtt{P}})$,*
- *$V([s]_{\sim_b}) = V(s)$,*
- *$Pr(h)$ is a mapping from each quotient state s^b to $Pr(h_{s^b})$, where $Pr(h_{s^b})$ is the probability distribution of $\mathtt{h}$ in the state s^b and is computed, for all $\overline{h} \in Val_h$, as*

$$Pr(h_{s^b} = \overline{h}) = \frac{\sum\limits_{s_i \in s^b,\ s_i = \langle .,\overline{h},.,.\rangle} Pr(s_i)}{Pr(s^b)}.$$

The public variable has a single initial value. Thus, all of the initial states of $\mathcal{M}_\delta^{\mathtt{P}}$ have the same public value and form a single equivalence class s_{init}^b. Each state s^b is labeled with a probability distribution $Pr(h_{s^b})$ which shows the probabilities of possible values of $\mathtt{h}$ in that state.

Example 3 (Back-bisimulation quotient of P1). The back-bisimulation quotient $\mathcal{M}_{uni}^{\mathtt{P1}}/\sim_b$ is depicted in Figure 3. Each state of $\mathcal{M}_{uni}^{\mathtt{P1}}/\sim_b$ is an equivalence class, containing related states of $\mathcal{M}_{uni}^{\mathtt{P1}}$:

$$s_0^b = \{s_0, s_4, s_7, s_{10}\}, \quad s_1^b = \{s_1, s_2, s_5, s_8\}, \quad s_2^b = \{s_3\}, \quad s_3^b = \{s_6, s_9\},$$
$$s_4^b = \{s_{15}\}, \quad s_5^b = \{s_{12}, s_{14}\}, \quad s_6^b = \{s_{11}, s_{13}\}.$$

States are labeled with the value of `l`, together with the distribution of `h`:

$$Pr(h_{s_0^b}) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}, \qquad Pr(h_{s_3^b}) = \{1 \mapsto \frac{1}{2}, 2 \mapsto \frac{1}{2}\},$$
$$Pr(h_{s_1^b}) = \{0 \mapsto \frac{1}{2}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}\}, \quad Pr(h_{s_5^b}) = \{1 \mapsto \frac{1}{2}, 2 \mapsto \frac{1}{2}\},$$
$$Pr(h_{s_2^b}) = \{0 \mapsto 1\}, \quad Pr(h_{s_4^b}) = \{3 \mapsto 1\}, \quad Pr(h_{s_6^b}) = \{1 \mapsto \frac{1}{2}, 2 \mapsto \frac{1}{2}\}.$$

The back-bisimulation quotient can be automatically constructed from the MC. This will be discussed in Section 3.4. After constructing the quotient, the next step is to compute the program leakage from the back-bisimulation quotient.

3.3 Measuring the Leakage Using Back-bisimulation Quotient

Let $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b = (S/\sim_b, \mathbf{P}'_\delta, s_{init}^b, Val_l, V, Pr(h))$ be the attacker's view of the program `P` running with the scheduler δ. In each state s^b of $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$, the secret distribution $Pr(h_{s^b})$ determines the attacker's uncertainty about `h`. Depending on the program statements that are chosen and executed by the scheduler, and the public values observed by the attacker, the distribution of `h` changes from state to state along each trace of $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$. In fact, $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$ *transforms* a priori distribution of `h` in the initial state s_{init}^b to posterior distributions in the final states $\mathit{final}(\mathcal{M}_\delta^{\mathtt{P}}/\sim_b)$.

The attacker's uncertainty about `h` in a state s^b with the secret distribution $Pr(h_{s^b})$ is measured by $\mathcal{H}_\infty(\mathtt{h}_{s^b})$. Thus, the initial uncertainty is measured by $\mathcal{H}_\infty(\mathtt{h}_{s_{init}^b})$.

Since there might be more than one final state with different reachability probabilities and the MC can be seen as a discrete probability distribution over all of its final states, the remaining uncertainty is defined as the expectation of uncertainties in all final states: $\sum\limits_{s_f^b \in \mathit{final}(\mathcal{M}_\delta^{\mathtt{P}}/\sim_b)} Pr(s_f^b)\mathcal{H}_\infty(\mathtt{h}_{s_f^b})$, where $Pr(s_f^b)$ is the probability of reaching s_f^b from the initial state s_{init}^b. It now follows that the leakage of the concurrent program `P` running under control of the scheduler δ is computed as $\mathcal{L}(\mathtt{P}_\delta) = \mathcal{H}_\infty(\mathtt{h}_{s_{init}^b}) - \sum_{s_f^b \in \mathit{final}(\mathcal{M}_\delta^{\mathtt{P}}/\sim_b)} Pr(s_f^b).\mathcal{H}_\infty(\mathtt{h}_{s_f^b})$.

Notice that for measuring the leakage of `P`, we computed min-entropy of initial and final states, and did not consider min-entropy of intermediate states. This is not in contrast with our assumption of taking into account the intermediate values of `l` along the execution paths. This is because in $\mathcal{M}_\delta^{\mathtt{P}}/\sim_b$ distributions of `h` in the final states result from values of `l` and distributions of `h` in the intermediate states. Thus, when computing the remaining uncertainty from the final distributions, the intermediate values of `l` are automatically taken into account. The final distributions of `h` also result from the program statements which are chosen by the scheduler. Therefore, the effect of the scheduler choices is considered, as well.

Moreover, in the literature, the remaining uncertainty is usually measured by the conditional entropy $\mathcal{H}_\infty(h|l)$, but we measure it by the non-conditional entropy $\mathcal{H}_\infty(h)$. These entropies are identical in our program model, because in

$\mathcal{M}^{\mathtt{P}}_{\delta}/\sim_b$ the entropy $\mathcal{H}_\infty(h)$ is computed from final states that result from traces observed by the attacker. This is exactly the same as the conditional entropy $\mathcal{H}_\infty(h|l)$.

Example 4 (Back-bisimulation quotient of `P1` is a distribution transformer.). In the initial state s^b_0 of $\mathcal{M}^{\mathtt{P1}}_{uni}/\sim_b$, the distribution of `h` is $Pr(h_{s^b_0}) = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$. This means that before executing the program, the attacker only knows that the value of `h` belongs to the set $\{0, 1, 2, 3\}$ and if she guesses the value of `h`, then the likelihood of her being successful is $\frac{1}{4}$. Therefore, $Pr(h_{s^b_0})$ determines the attacker's initial uncertainty about `h`. Now consider the final state s^b_3, in which the distribution of `h` is $Pr(h_{s^b_3}) = \{1 \mapsto \frac{1}{2}, 2 \mapsto \frac{1}{2}\}$. In this state, the attacker knows that the value of `h` belongs to $\{1, 2\}$, and thus her uncertainty about `h` is reduced. This means that after executing the program and observing the trace $\langle 0, 1, 0^\omega \rangle$, if the attacker guesses the value of `h`, then the likelihood of her being successful is $\frac{1}{2}$. These considerations imply that the back-bisimulation quotient is a distribution transformer.

Example 5 (Information leakage of `P1`). The initial uncertainty is quantified as the Renyi's min-entropy of `h` in the initial state s^b_0, i.e., $\mathcal{H}_\infty(\mathtt{h}_{s^b_0}) = -\log_2 \frac{1}{4} = 2$ $(bits)$.

The remaining uncertainty is quantified as the Renyi's min-entropy of `h` in the final states. There are four final states with different reachability probabilities: s^b_2, s^b_5, s^b_3, s^b_4. Consequently, the remaining uncertainty is quantified as the *expectation* of the Renyi's min-entropy of `h` in these states:

$$\sum_{s^b \in \{s^b_2, s^b_5, s^b_3, s^b_4\}} Pr(s^b).\mathcal{H}_\infty(\mathtt{h}_{s^b}) = -\frac{1}{4} * \log_2 1 - \frac{1}{4} * \log_2 \frac{1}{2} - \frac{1}{4} * \log_2 \frac{1}{2}$$
$$-\frac{1}{4} * \log_2 1 = 0.5\ (bits),$$

where $Pr(s^b)$ denotes the probability of reaching s^b from the initial state s^b_0. Finally, the leakage of the program `P1` running with the uniform scheduler is computed as $\mathcal{L}(\mathtt{P1}_{uni}) = 2 - 0.5 = 1.5$ $(bits)$.

The following section formally defines the back-bisimulation equivalence and explains how to compute the back-bisimulation quotient.

3.4 Computing Back-bisimulation Quotient

In this section, we discuss how to compute the back-bisimulation quotient. Before that, we first explain a subclass of MCs, called *Markov chains with pseudoback-bisimilar states*, in which back-bisimulation cannot correctly construct the attacker's view of the program behavior.

Pseudoback-bisimilar States In order to compute the states of a back-bisimulation quotient, we need to aggregate back-bisimilar states into one equivalence class. For that, we define *Back-bisimulation signature*, which is defined as a kind of fingerprint for states of a back-bisimulation equivalence class.

Definition 8. *The* ***back-bisimulation signature*** *of a state s is defined as*

$$sig_{\sim_b}(s) = \{\ \big(V(s), [s']_{\sim_b}\big) \mid \exists s' \in Pre(s)\ \}.$$

It asserts that two states that have the same public value and their predecessors belong to the same equivalence class, have the same signature.

Definition 9. *Let $\mathcal{M}_\delta^P$ be an MC. Two states $s_1, s_2 \in S$ are* ***pseudoback-bisimilar*** *iff (1) $V(s_1) = V(s_2)$, (2) $level(s_1) = level(s_2)$, (3) $sig_{\sim_b}(s_1) \neq sig_{\sim_b}(s_2)$, (4) $PreLabels(s_1) \cap PreLabels(s_2) \neq \varnothing$. An MC that contains some pseudoback-bisimilar states is denoted by $MC_\mathtt{p}$ and an MC with no pseudoback-bisimilar state is denoted by $MC_\mathtt{n}$.*

Stated in words, two states are pseudoback-bisimilar if they have the same label, are at the same level (distance from an initial state), and have different signatures, but intersecting pre-labels. In an $\mathrm{MC}_\mathtt{n}$, states at the same level and with the same label, either have no intersecting pre-labels or have the same pre-labels.

Example 6 (An example $MC_\mathtt{p}$). Consider the following program:

```
l:=0;
if h=1 then l:=1; l:=2; l:=3; l:=4; l:=5
else (l:=1 || l:=2); l:=3; (l:=4 || l:=5)            (P2)
```

where $Val_h \in \{0, 1\}$ and $Pr(h) = \{0 \mapsto \frac{1}{2}, 1 \mapsto \frac{1}{2}\}$. A uniform scheduler is selected for both parallel operators. The MC $\mathcal{M}_{uni}^{\mathtt{P2}}$ is shown in Figure 4a.

In $\mathcal{M}_{uni}^{\mathtt{P2}}$, states s_8 and s_9 are pseudoback-bisimilar. They both have the label 3, are at the level 3, and have different signatures:

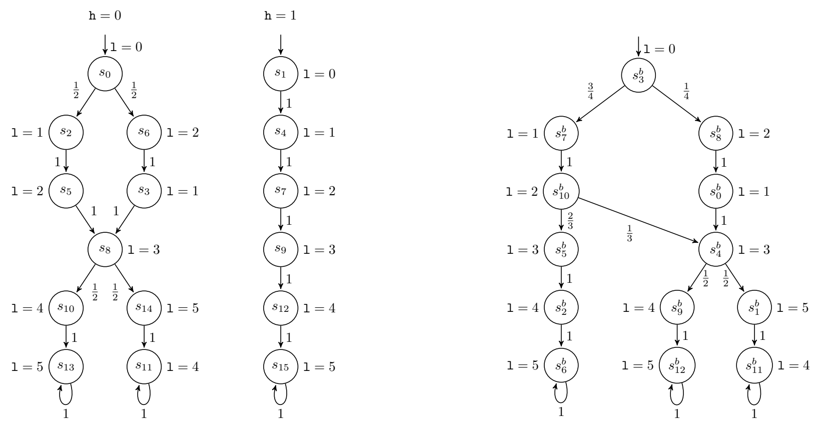
$$sig_{\sim_b}(s_8) = \Big\{\big(3, \{s_5, s_7\}\big), \big(3, \{s_3\}\big)\Big\}, sig_{\sim_b}(s_9) = \Big\{\big(3, \{s_5, s_7\}\big)\Big\},$$

but intersecting pre-labels: $preLabels(s_8) = \{2, 1\}, \qquad preLabels(s_9) = \{2\}$.

Back-bisimulation captures the attacker's view for all programs that do not contain pseudoback-bisimilar states, i.e., those final states that have the same public values and result from the same trace are indistinguishable and fall into the same $\sim_b$-equivalence class. Formally,

Theorem 2. *Let $\mathcal{M}_\delta^P$ be an $MC_\mathtt{n}$. For all paths $\sigma_1, \sigma_2 \in Paths(\mathcal{M}_\delta^P)$ with $\sigma_1 = s_{0,1}s_{1,1}\ldots s_{n-1,1}(s_{n,1})^\omega$, $\sigma_2 = s_{0,2}s_{1,2}\ldots s_{n-1,2}(s_{n,2})^\omega$, and $n \geq 0$ it holds that $s_{n,1} \sim_b s_{n,2}$ iff $trace(\sigma_1) = trace(\sigma_2)$.*

Theorem 2 argues that same-trace final states are back-bisimilar. A similar argument can be made for non-final states.



(a) MC of the program P2 with the uniform scheduler

(b) back-bisimulation quotient of $\mathcal{M}^{\texttt{P2}}_{uni}$

Fig. 4: $\mathcal{M}^{\texttt{P2}}_{uni}$ and $\mathcal{M}^{\texttt{P2}}_{uni}/\sim_b$

Theorem 3. *Let $\mathcal{M}^{\texttt{P}}_{\delta}$ be an MC$_{\texttt{n}}$. For all paths $\sigma_1, \sigma_2 \in Paths(\mathcal{M}^{\texttt{P}}_{\delta})$ with $\sigma_1 = s_{0,1}s_{1,1}\dots s_{n-1,1}(s_{n,1})^{\omega}$, $\sigma_2 = s_{0,2}s_{1,2}\dots s_{m-1,2}(s_{m,2})^{\omega}$, $n, m > 0$, and $0 \le i < min(n,m)$ it holds that $s_{i,1} \sim_b s_{i,2}$ iff $trace_{\ll i}(\sigma_1) = trace_{\ll i}(\sigma_2)$.*

Therefore, all paths of $\mathcal{M}^{\texttt{P}}_{\delta}$ with the same trace form a single path in $\mathcal{M}^{\texttt{P}}_{\delta}/\sim_b$. Stated formally, let $\sigma' \in Paths(\mathcal{M}^{\texttt{P}}_{\delta}/\sim_b)$, $s^b_f = final(\sigma') \in final(\mathcal{M}^{\texttt{P}}_{\delta}/\sim_b)$, and $T = trace(\sigma')$. The path σ' is the aggregation of all paths $Paths(T) \subseteq Paths(\mathcal{M}^{\texttt{P}}_{\delta})$. All final states of $\mathcal{M}^{\texttt{P}}_{\delta}$ that result from the trace T fall into the same $\sim_b$-equivalence class $s^b_f = \{s_f \mid s_f \in final(Paths(T))\}$. For s^b_f, the secret distribution $Pr(h_{s^b_f})$ contains probabilities of possibles values of h that the attacker might be able to guess by observing T.

Pseudoback-bisimilar states do not fall into the same $\sim_b$-equivalence class and thus $\sim_b$ is not able to aggregate all paths with the same trace. For instance, in $\mathcal{M}^{\texttt{P2}}_{uni}/\sim_b$ (Figure 4b) there are two paths $s^b_3 s^b_7 s^b_{10} s^b_5 s^b_2 (s^b_6)^{\omega}$ and $s^b_3 s^b_7 s^b_{10} s^b_4 s^b_9 (s^b_{12})^{\omega}$ with the same trace $\langle 0, 1, 2, 3, 4, 5^{\omega}\rangle$. The attacker, after observing the trace, cannot discriminate the value of h to be 0 or 1. But, in the attacker's view of the MC constructed by back-bisimulation (Figure 4b) the value of h is distinguished in the final states of the two paths. The implication is that back-bisimulation cannot correctly construct the attacker's view of an MC$_{\texttt{p}}$. For MC$_{\texttt{p}}$s, we use the trace-exploration-based method, introduced in [25], which computes the program leakage directly from the MC$_{\texttt{p}}$ $\mathcal{M}^{\texttt{P}}_{\delta}$.

Algorithm for Computing the Back-bisimulation Quotient Space In this section, an algorithm is proposed for obtaining the back-bisimulation quotient space for a finite MC$_{\texttt{n}}$. This algorithm is similar to Kanellakis and Smolka's algorithm for computing the bisimulation quotient space [20]. It relies on a *partition refinement* technique, where the state space is partitioned into *blocks*. It

Algorithm 1 A first iterative quotienting algorithm

```
    Input: finite MC_n $\mathcal{M}_\delta^{\mathtt{P}}$ with state space $S$
    Output: back-bisimulation quotient space $S/\sim_b$

    /* Determine the initial partition $\Pi_0$ */
 1: $s^b_{init} := Init(\mathcal{M}_\delta^{\mathtt{P}})$;
 2: $\mathcal{R} := \{(s_1, s_2) \mid V(s_1) = V(s_2)\}$;
 3: $\Pi_0 = \{s^b_{init}\} \cup \Big(\big(S \setminus Init(\mathcal{M}_\delta^{\mathtt{P}})\big) \;/\; \mathcal{R}\Big)$;

 4: $\Pi := \Pi_0$;
 5: $\Pi_{old} := \{S\}$; // $\Pi_{old}$ contains the previous partition
    /* loop until no refinement possible */
 6: while $\Pi\ != \ \Pi_{old}$ do
 7:     $\Pi_{old} := \Pi$;
        /* search through the blocks of $\Pi_{old}$ to find a splitter candidate for $\Pi$ */
 8:     for all $\mathcal{C} \in \Pi_{old}$ do
 9:         $\Pi := Refine_b(\Pi, \mathcal{C})$;
10: return $\Pi$;
```

starts from an initial partition Π_0 and computes successive refinements of Π_0 until a stable partition is reached. A partition is stable if no further refinements are possible. The obtained partition is $S/\sim_b$, the largest back-bisimulation over the input finite MC$_{\mathtt{n}}$. The essential steps are outlined in Algorithm 1. The algorithm consists of two main parts: (a) computing the initial partition, and (b) successively refining the partitions.

Computing the Initial Partition. Since back-bisimilar states have the same public value, it is sensible to use this in determining the initial partition Π_0.

All initial states have the same public value and have no predecessors. Consequently, they form a single block $s^b_{init} = Init(\mathcal{M}_\delta^{\mathtt{P}})$. This block will remain unchanged during the refinements.

For the remaining states $S \setminus s^b_{init}$, each group of states with the same public value forms a block. Same-label blocks can be obtained by the equivalence relation $\mathcal{R} = \{(s_1, s_2) \mid V(s_1) = V(s_2)\}$, which induces the quotient spaces $\big(S \setminus Init(\mathcal{M}_\delta^{\mathtt{P}})\big) \;/\; \mathcal{R}$. Thus, the initial partition is obtained as $\Pi_0 = \{s^b_{init}\} \cup \Big(\big(S \setminus Init(\mathcal{M}_\delta^{\mathtt{P}})\big) \;/\; \mathcal{R}\Big)$.

Partition Refinement. Since all partitions are a refinement of the initial partition Π_0, each block in these partitions contains states with the same public value. However, blocks of Π_0, except s^b_{init}, do not consider the one-step predecessors of states. This is taken care of in the successive refinement steps, by the refinement operator.

Definition 10. *Let Π be a partition for S and $\mathcal{C}$ be a superblock of Π. Then,*

$$Refine_b(\Pi, \mathcal{C}) = \bigcup_{\mathcal{B} \in \Pi} Refine_b(\mathcal{B}, \mathcal{C}),$$

where $\mathit{Refine}_b(\mathcal{B}, \mathcal{C}) = \{\mathcal{B} \cap \mathit{Succ}(\mathcal{C}),\ \mathcal{B} \setminus \mathit{Succ}(\mathcal{C})\} \setminus \{\varnothing\}$. *Here,* $\mathcal{C}$ *is called a* splitter *for* Π*, refining blocks of* Π *to subblocks.*

Using $\mathcal{C}$, $\mathit{Refine}_b(\mathcal{B}, \mathcal{C})$ decomposes the block $\mathcal{B}$ into two subblocks, provided that the subblocks are nonempty.

A key step of computing the back-bisimulation quotient space is to determine a splitter $\mathcal{C}$ for a given partition Π. Algorithm 1 uses the blocks of the previous partition Π_{old} as splitter candidates for Π.

Theorem 4. *Algorithm 1 always terminates and correctly computes the back-bisimulation quotient space* $S/\sim_b$.

The following theorem discusses the time complexity of Algorithm 1.

Theorem 5. *The time complexity of Algorithm 1 is* $O(|S|.|E|)$*, where* E *denotes the set of transitions of* $\mathcal{M}_\delta^P$.

4 Conclusions and Future Work

In this paper, a quantification approach is proposed for concurrent programs. Back-bisimulation equivalence is defined to model the attacker's view of the program behavior. Then a partition refinement algorithm is developed to compute the back-bisimulation quotient of the program. The back-bisimulation quotient is automatically constructed and contains secret distributions, which are used to compute the information leakage of the program.

The back-bisimulation quotient contains all execution traces which the attacker can observe during executing the program. Thus, it can be used to compute maximal and minimal leakages that might occur during the program executions. Furthermore, the quotient is an abstract model of the program and the quantification analysis is done on a minimized model, most likely saving time and space.

Back-bisimulation equivalence creates a lot of exciting opportunities for future works. It can be used to verify any trace-equivalence-based property, such as *observational determinism* [21, 26, 33], a widely-studied confidentiality property of secure information flow for concurrent programs. It can also be defined on multi-terminal binary decision diagrams (MTBDDs), in order to improve the scalability of the quantification approach to a great extent. We aim to lift the program-termination restriction and extend the proposed approach to non-terminating concurrent programs. We also aim to study *bounded leakage problem* [25] and *channel capacity* [29] on the back-bisimulation quotient. Probably, using some reductions, such as on-the-fly techniques, can improve the scalability of the problem. Furthermore, handling programs with pseudoback-bisimilar states using back-bisimulation is a possible future work. Another avenue to consider the current work is to perform time analysis of the proposed approach, e.g. on dining cryptographers protocol.

A Case Study

In this section, we analyze a case study to show applicability and feasibility of the approach.

The Dining Cryptographers Protocol. We consider the *dining cryptographers problem* [11] to show how an attacker can deduce secret information through execution observations. The dining cryptographers problem was first proposed by David Chaum in 1988 as an example of anonymity and identity hiding [11]. In this problem, N cryptographers are sitting at a round table to have dinner at their favorite restaurant. The waiter informs them that the meal has been arranged to be paid by one of the cryptographers or their master. The cryptographers respect each other's right to stay anonymous, but would like to know whether the master is paying or not. So, they decide to take part in the following two-stage protocol:

- Stage 1: Each cryptographer tosses a coin and only informs the cryptographer on the right of the outcome.
- Stage 2: Each cryptographer publicly announces whether the two coins that she can see are the same ('agree') or different ('disagree'). However, if she actually paid for the dinner, then she lies, i.e., she announces 'disagree' when the coins are the same, and 'agree' when they are different.

Let the variable *parity* be exclusive-or (XOR) between all the announcements. If N is odd, then an odd number of 'agree's (parity=1) implies that none of the cryptographers paid (the master paid), while an even number (parity=0) implies that one of the cryptographers paid. The latter is reverse for an even N.

The payer can be either

i. one of the cryptographers, i.e., $Val_{payer} = \{c_1, \ldots, c_N\}$, or
ii. the master (m, for short) or one of the cryptographers, i.e., $Val_{payer} = \{m, c_1, \ldots, c_N\}$.

Assume an attacker who tries to find out the payer's identity. The attacker is external, i.e., none of the cryptographers. This attacker can observe the parity and also the announcements of the cryptographers. All observable variables are concatenated to form a single public variable. The program model is an $\mathrm{MC}_{\mathfrak{n}}$ and we employ the proposed algorithms to compute the leakage.

The experimental results for the cases in which the coin probability is 0.5 are shown in Table 1. In this table, N denotes the number of cryptographers. $\mathcal{M}_{uni}^{\mathtt{DC}_N}$ and $\mathcal{M}_{uni}^{\mathtt{DC}_N} / \sim_b$ denote the MC of the program run with a uniform scheduler and the back-bisimulation quotient, respectively. Symbols $\#st$ and $\#tr$ denote the number of states and transitions, respectively.

Similar results for the coin probability of 0 or 1 are shown in Table 2. As shown in Tables 1 and 2, back-bisimulation results in impressive reductions of the state space. For example, when the coin probability is 0.5 (Table 1) reductions vary between 92% and 99.5%.

Table 1: Evaluation results for the dining cryptographers protocol with the coin probability 0.5

Val_{payer}	N	$\mathcal{M}_{uni}^{\mathtt{DC}_N}$		$\mathcal{M}_{uni}^{\mathtt{DC}_N} / \sim_b$		**leakage (bits)**
		$\#st$	$\#tr$	$\#st$	$\#tr$	
$\{m, c_1, \ldots, c_N\}$	3	380	776	26	45	**0.811 (40%)**
	4	2165	5720	64	144	**0.721 (31%)**
	5	11850	38772	152	420	**0.65 (25%)**
	6	63063	246820	352	1152	**0.59 (21%)**
$\{c_1, \ldots, c_N\}$	3	285	582	22	36	**0**
	4	1732	4576	56	121	**0**
	5	9875	32310	136	365	**0**
	6	54054	211560	320	1125	**0**

Consider the last three cases of Table 1, where the coin probability is 0.5 and the payer is one of the cryptographers ($Val_{payer} = \{c_1, \ldots, c_N\}$). In these cases, the program leakage is 0. This shows that the attacker cannot identify the payer. This is why the dining cryptographers protocol is said to be secure in the context of anonymity.

The analysis results in Table 2 show that when the probability of the coin is 0 or 1, no matter whoever the payer is, the leakage is $\log_2 |Val_{payer}|$, proving that the secret gets completely leaked and thus the attacker learns the identity of the payer.

B Related Work

The notion of back-simulation is similar to the notion of backward strong bisimulation considered by De Nicola and Vaandrager [15]. They use a different notion than our definition, as they only allow to move back from a state along the path representing the history that brought one into that state. Högberg et al. [17] defined and considered backward bisimulation minimization on tree automata, Sproston and Donatelli [32] considered a probabilistic version of backward bisimulation and studied the logical properties it preserves, and Cardelli et al. [9] who considered backward bisimulation in the stochastic setting of chemical reaction networks. None of these works use backward bisimulation in quantitative information flow.

Chen and Malacaria [12] model multi-threaded programs as state transition systems. They use Bellman's optimality principle to determine the leakage bounds, i.e., minimal and maximal leakage occurred during possible program executions.

Table 2: Evaluation results for the dining cryptographers protocol with the coin probability 0 or 1

Val_{payer}	N	$\mathcal{M}_{uni}^{\mathtt{DC}_N}$		$\mathcal{M}_{uni}^{\mathtt{DC}_N} / \sim_b$		**leakage (bits)**
		$\#st$	$\#tr$	$\#st$	$\#tr$	
$\{m, c_1, \ldots, c_N\}$	3	72	124	21	37	**2 (100%)**
	4	235	525	47	107	**2.32 (100%)**
	5	738	2046	103	286	**2.585 (100%)**
	6	2254	7483	223	729	**2.807 (100%)**
$\{c_1, \ldots, c_N\}$	3	54	93	20	34	**1.585 (100%)**
	4	188	420	46	103	**2 (100%)**
	5	615	1705	102	281	**2.32 (100%)**
	6	1932	6414	222	723	**2.585 (100%)**

Phan et al. [27] propose to use symbolic execution, a verification technique which bounds runtime behavior of the program, thus mitigating state-space explosion problem. In state-space explosion problem, the amount of state-space of the program model gets too huge to store in the memory, thus making the analysis difficult. Phan et al. run symbolic execution to extract all symbolic paths of the program. Then, paths with a direct information flow are labeled. Finally, they use a model counting technique to count the number of inputs that follow direct-labeled paths, to compute *channel capacity*, which is an upper bound of the leakage over *all* possible distributions of the secret input.

Biondi et al. [8] use interval Markov chains to compute the channel capacity of deterministic processes. They reduce the channel capacity computation to entropy maximization, a well-known problem in Bayesian statistics.

Chothia et al. [14] have developed LeakWatch to approximate leakage of Java programs. LeakWatch is based on probabilistic point-to-point information leakage, in which the leakage between any given two points in the program from secret to public variables is computed.

Chadha et al. [10] employ symbolic algorithms to quantify the precise leakage from public to secret variables. They use Binary Decision Diagrams (BDDs) to model the relation between the inputs and outputs of the program. To do so, Moped [16], a symbolic model checker, is exploited to construct BDDs. Chadha et al. have implemented their method into a tool called Moped-QLeak.

Klebanov [22] uses symbolic execution in combination with deductive verification [5] and self-composition [4] to measure residual Shannon entropy and min-entropy of the secret input. Exploitation of deductive verification makes the analysis immune to the state-space explosion problem, but also makes it semi-automatic, as user-supplied invariants are needed for the analysis to proceed.

References

1. Alvim, M.S., Andrés, M.E., Chatzikokolakis, K., Palamidessi, C.: Quantitative Information Flow and Applications to Differential Privacy, pp. 211–230. Springer Berlin Heidelberg (2011)
2. Amir-Mohammadian, S.: A semantic framework for direct information flows in hybrid-dynamic systems. In: Proceedings of the 7th ACM Cyber-Physical System Security Workshop (CPSS 2021). pp. 5–15. Association for Computing Machinery (June 2021)
3. Baier, C., Katoen, J.P.: Principles of model checking. MIT press Cambridge (2008)
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE workshop on Computer Security Foundations, CSFW'04. pp. 100–114. IEEE Computer Society (2004)
5. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of object-oriented software: The KeY approach. Springer-Verlag (2007)
6. Biondi, F.: Markovian Processes for Quantitative Information Leakage. Ph.D. thesis, IT University of Copenhagen (2014)
7. Biondi, F., Legay, A., Malacaria, P., Wasowski, A.: Quantifying information leakage of randomized protocols. Theoretical Computer Science **597**, 62–87 (2015)
8. Biondi, F., Legay, A., Nielsen, B.F., Wasowski, A.: Maximizing entropy over markov processes. Journal of Logical and Algebraic Methods in Programming **83**(5), 384–399 (2014)
9. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Forward and backward bisimulations for chemical reaction networks. arXiv preprint arXiv:1507.00163 (2015)
10. Chadha, R., Mathur, U., Schwoon, S.: Computing Information Flow Using Symbolic Model-Checking. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 505–516. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
11. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of cryptology **1**(1), 65–75 (1988)
12. Chen, H., Malacaria, P.: The optimum leakage principle for analyzing multi-threaded programs. In: Proceesings of International Conference on Information Theoretic Security. pp. 177–193. Springer (2009)
13. Chen, H., Malacaria, P.: Quantifying maximal loss of anonymity in protocols. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. pp. 206–217. ACM (2009)
14. Chothia, T., Kawamoto, Y., Novakovic, C.: Leakwatch: Estimating information leakage from Java programs. In: Kutyłowski, M., Vaidya, J. (eds.) Computer Security - ESORICS 2014. pp. 219–236. Springer International Publishing, Cham (2014)
15. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. Journal of the ACM (JACM) **42**(2), 458–487 (1995)
16. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with craig interpolation and symbolic pushdown systems. Journal on Satisfiability, Boolean Modeling and Computation **5**, 27–56 (2008)
17. Högberg, J., Maletti, A., May, J.: Backward and forward bisimulation minimization of tree automata. Theoretical Computer Science **410**(37), 3539–3552 (2009)

18. Jurado, M., Palamidessi, C., Smith, G.: A formal information-theoretic leakage analysis of order-revealing encryption. In: Proceedings of the 34th IEEE workshop on Computer Security Foundations, CSFW'21. IEEE Computer Society (2021)
19. Jurado, M., Smith, G.: Quantifying information leakage of deterministic encryption. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop. pp. 129–139 (2019)
20. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1), 43–68 (1990)
21. Karimpour, J., Isazadeh, A., Noroozi, A.A.: Verifying observational determinism. In: Proceedings of 30th IFIP TC 11 International Conference on ICT Systems Security and Privacy Protection, SEC'15, pp. 82–93. Springer-Verlag (2015)
22. Klebanov, V.: Precise quantitative information flow analysis - a symbolic approach. Theoretical Computer Science **538**, 124–139 (2014)
23. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 286–296. ACM (2007)
24. Köpf, B., Smith, G.: Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In: Proceesings of 23rd IEEE Computer Security Foundations Symposium (CSF). pp. 44–56. IEEE (2010)
25. Noroozi, A.A., Karimpour, J., Isazadeh, A.: Information leakage of multi-threaded programs. Computers & Electrical Engineering **78**, 400–419 (2019)
26. Noroozi, A.A., Salehi, K., Karimpour, J., Isazadeh, A.: Secure information flow analysis using the prism model checker. In: International Conference on Information Systems Security. pp. 154–172. Springer (2019)
27. Phan, Q.S., Malacaria, P., Păsăreanu, C.S., d'Amorim, M.: Quantifying information leaks using reliability analysis. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software. pp. 105–108. ACM (2014)
28. Puterman, M.L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons (1994)
29. Salehi, K., Karimpour, J., Izadkhah, H., Isazadeh, A.: Channel capacity of concurrent probabilistic programs. Entropy **21**(9), 885 (2019)
30. Salehi, K., Noroozi, A.A., Amir-Mohammadian, S.: Quantifying information leakage of probabilistic programs using the prism model checker. In: Emerging Security Information, Systems and Technologies. pp. 47–52. IARIA (2021)
31. Smith, G.: On the foundations of quantitative information flow. In: Proceedings of the 12th International conference on Foundations of Software Science and Computational Structures, FOSSACS 2009. pp. 288–302. Springer (2009)
32. Sproston, J., Donatelli, S.: Backward bisimulation in markov chain model checking. IEEE Transactions on Software Engineering **32**(8), 531–546 (2006)
33. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop, CSFW'03. pp. 29–43. IEEE Computer Society (2003)