Verifying Weak Probabilistic Noninterference

Ali A. Noroozi
Department of Computer Science
University of Tabriz
Tabriz, Iran

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

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

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

***Abstract*—Weak probabilistic noninterference is a security property for enforcing confidentiality in multi-threaded programs. It aims to guarantee secure flow of information in the program and ensure that sensitive information does not leak to attackers. In this paper, the problem of verifying weak probabilistic noninterference by leveraging formal methods, in particular algorithmic verification, is discussed. Behavior of multi-threaded programs is modeled using probabilistic Kripke structures and formalize weak probabilistic noninterference in terms of these structures. Then, a verification algorithm is proposed to check weak probabilistic noninterference. The algorithm uses an abstraction technique to compute quotient space of the program with respect to an equivalence relation called weak probabilistic bisimulation and does a simple check to decide whether the security property is satisfied or not. The progress made is demonstrated by a real-world case study. It is expected that the proposed approach constitutes a significant step towards more widely applicable secure information flow analysis.**

***Keywords*—Confidentiality; secure information flow; noninterference; algorithmic verification; bisimulation**

I. Introduction

A. Motivation

In information security, a *confidentiality policy* prevents the unauthorized disclosure of information. Confidentiality policies are defined in terms of confidentiality mechanisms, which are approaches to enforce the policies [1]. Cryptography and access control are examples of confidentiality mechanisms. But they do not restrict the flow of information inside a program. For example, when an android application grants permission to access contacts, there is no cryptography or access control mechanism to verify legal use of the contacts by the application. This is where secure information flow comes to the rescue.

Secure information flow controls the way information flows throughout a program. Information flow properties are designed to prevent the information from flowing to an unauthorized user, i.e., attacker or low-observer [2]. Typically, it is supposed that there are two security levels, *high* (*H*) and *low* (*L*), corresponding to higher and lower confidentiality for program variables respectively. An information flow property is defined in such a way that it prevents data in *H* from flowing to *L*. More complex hierarchies of security levels can be defined via a security structure [3]. Information flow properties are of paramount significance for guaranteeing confidentiality of data. Because of this, it is desirable to establish an automatic and efficient verification approach for secure information flow.

B. Background

In most of researches done on secure information flow, a security property specifying the confidentiality policy is formally defined and then a verification method is proposed to check the property. *Noninterference* [4] is a long-established information flow property, stipulating that high data may not interfere with low data. The absence of interference requires *indistinguishability* of program behavior, as secret inputs are varied.

Probabilistic noninterference is a widely-used security property for multi-threaded programs, proposed by Volpano and Smith [5], and extended by Sabelfeld and Sands [6]. It is a timing- and probabilistic-sensitive property, defined over a simple imperative language with dynamic thread creation. Sabelfeld and Sands define a timing-sensitive partial probabilistic bisimulation to characterize indistinguishability of the executions of the program. The intuition is that low-equivalent states must produce executions that run in lock-step, affect the shared memory in the same way, and the probability of stepping to the states from the same equivalence class be the same [6].

Smith [7] shows that probabilistic bisimulation is too strict regarding time. To address this problem, Smith defines probabilistic noninterference in terms of *weak probabilistic bisimulation*, allowing probabilistic systems to be regarded as equivalent when they do not run at the same time. The resultant property is called *weak probabilistic noninterference*, which requires low-equivalent states to produce executions that visit the same sequence of equivalence classes, but some executions may remain in a class longer that the other executions.

Verifying secure information flow is mostly done via *information flow type systems*. A type system is a formal system of type inference rules for reasoning about properties of programming languages [8]. In information flow type systems,

the property of interest is a property of secure information flow, e.g., probabilistic noninterference. Many information flow type systems have been proposed to enforce probabilistic noninterference. Sabelfeld and Sands [6] define a type system to verify probabilistic noninterference. Smith [9] proposes a new type system to enforce probabilistic noninterference for multi-threaded programs running under a uniform probabilistic scheduler. In [7], Smith applies weak probabilistic bisimulation to prove that the type system proposed by him in [9] guarantees the probabilistic noninterference.

Type systems are automated and compositional, but they are not extensible, as each new feature added to the programming language, or variation of the information flow property requires a redefinition of the type system and its soundness proof [10]. Consequently, *algorithmic verification* has been favored recently, which is the application of rigorous, mathematically sound, and fully automatic techniques to the analysis of systems. These techniques are more flexible than type systems, and give a precise and efficient mechanism to verify a variety of security properties, without the need to prove soundness repeatedly [11].

Algorithmic verification techniques have been mostly developed for trace properties, which describe single executions of programs. But, most security properties, including weak probabilistic noninterference, are *2-safety* properties. 2-safety properties predicate over two executions of a program and consequently, verification requires establishing relationships between two different executions [12]. For example, weak probabilistic noninterference is not a property of individual executions and hence not a trace property, because whether an execution is allowed by the property depends on whether another execution is also allowed. 2-safety properties are an important subset of *relational* properties, which describe multiple executions of one or more programs [13]. As most classical verification techniques are not adequate to reason about relational properties, recently, many new techniques have been developed for secure information flow [12], [14]-[19], but none for weak probabilistic noninterference.

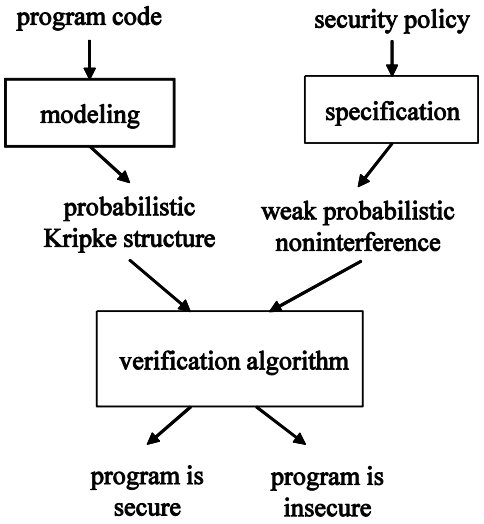

Fig. 1. Components of the proposed approach.

C. Foreground

In this paper, an algorithm is developed to verify weak probabilistic noninterference for multi-threaded programs running under an arbitrary scheduler. The program to be verified is modeled by a probabilistic state transition system, called probabilistic Kripke structure. Weak probabilistic noninterference is formally defined in terms of semantics of the probabilistic Kripke structure. In the proposed analysis, a program satisfies weak probabilistic noninterference, if and only if all executions with low-equivalent initial states visit the same sequence of equivalent classes with respect to weak probabilistic bisimulation. The verification algorithm computes the quotient space, i.e., the set of all equivalence classes of the probabilistic Kripke structure and does a simple check to decide the satisfaction of the security property. The quotient space is an abstraction of the concrete model of a program and allows obtaining enormous state-space reductions, possibly avoiding sate explosion problem. It is shown that the proposed verification algorithm runs in polynomial time. A case study is provided to show the feasibility of the verification algorithm. Fig. 1 gives a clear picture of the proposed approach.

D. Structure of the Paper

The paper starts by an informal overview of the approach in Section II. The program model assumed throughout the paper is presented in Section III. Weak probabilistic noninterference is defined in Section IV, using weak probabilistic bisimulation. The verification algorithm, time complexity, and application of the algorithm to a case study are addressed in Section V. Discussing related work and comparisons are done in Section VI. Finally, Section VII concludes the paper and discusses some future work.

II. Overview of Approach

In this section, a tour of the proposed work is given. To build intuition for the proposed approach, the key idea is illustrated using an example.

For clarity, some informal definitions are discussed. Suppose an *attacker* has full knowledge of source code of a multi-threaded program, can choose a scheduler for its execution, and observe the program behavior under the chosen scheduler. By observing behavior, we mean the attacker can see values of *public* variables *during* the program execution. For example, she can print public values. If the attacker can infer information about secret (high) values of the program by observing public (low) values, the program is said to have a *leak (or channel)*. Depending on the ability of the attacker, programs may have different leaks; e.g., explicit, implicit, or probabilistic leaks. Explicit leaks occur when a high value is assigned to a low variable; e.g., `l:=h`, assuming `l` is a low variable and `h` is a high variable. Implicit flows happen because of the control structure of a program; e.g., `if h=1 then l:=1 else l:=0`. Probabilistic leaks occur as a result of probabilistic behavior of the program. An example of this leak will be given in the following.

Secure information flow to the rescue. Secure information flow analysis aims to detect and consequently avoid information leaks in a program. Usually, it involves three main steps: 1) The program behavior is defined using a

program model; 2) The absence of leaks is defined using a security property; 3) A verification technique is developed to check the satisfaction ability of the property in the given program. In this paper, probabilistic Kripke structure (definition 1) is used to model the program behavior. Weak probabilistic noninterference (definition 8) of Smith [7] is reformulated in terms of the program model, and finally an algorithmic verification technique (Algorithm 1) is developed to check the property.

A program satisfies weak probabilistic noninterference, if each *pair* of program executions with *low-equivalent* initial values are *indistinguishable*. Smith defines indistinguishability via weak probabilistic bisimulation $\approx_p$ (definition 6), an equivalence relation relating executions that change low values in the *same order*, with the *same probability*. Thus, an attacker observing pairs of weak probabilistic bisimilar executions with low-equivalent initial values (and probably different initial high values), will not be able to distinguish these executions and consequently infer secret information.

For further clarity, consider the following example program:

l:=0; l:=h mod 2; (l:=h || (l:=0 || l:=1))

Where, || is the parallel operator and h can have values 0 or 1. Suppose a uniform scheduler S, where each statement of || is chosen with probability ½. Then, final value of l will reveal h with probability of ¾. This is a probabilistic leak. Fully probabilistic Kripke structure of the program $\mathrm{K_S}$ induced by the uniform scheduler is shown in Fig. 2. In this figure, nodes and edges represent states and transitions between states of the program, respectively. Edge labels show transition probabilities. Probability of transitions without a label is 1. Each state label shows the value of l in the corresponding state. The set of states, initial states, and executions are $S = \{s_0, s_1, \ldots, s_{24}\}$, $\mathrm{I} = \{s_0\}$, and $Execs(\mathrm{K_S}) = \{\sigma_0 = s_0 s_1 s_2 s_3 s_4^{\omega}, \sigma_1 = s_0 s_1 s_2 s_5 s_6^{\omega}, \ldots, \sigma_7 = s_0 s_{13} s_{22} s_{23} s_{24}^{\omega}\}$, respectively.

Verification. According to weak probabilistic noninterference, executions with low-equivalent initial values should be weak probabilistic bisimilar. A key idea of the proposed technique is to break down the executions of the program into various *groups*, depending on low-equivalency of initial states, and in each group check $\approx_p$ between the executions. To do this, the initial states are partitioned, based on the low-equivalence relation, into $\mathrm{I\,B}_0, \ldots, \mathrm{I\,B}_m$ so that $\approx_p$ can be checked between every $\sigma \in Execs(\mathrm{I\,B}_i)$.

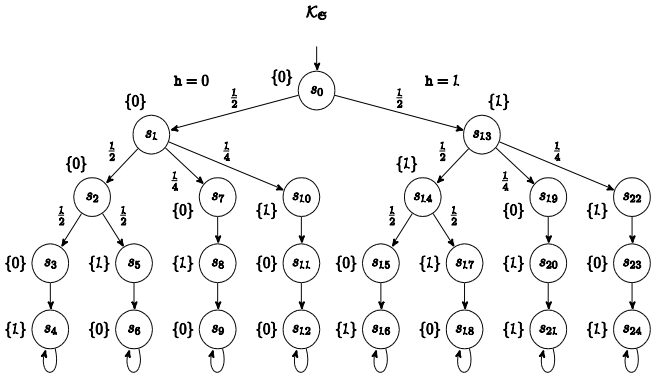

Fig. 2. Model of the example program.

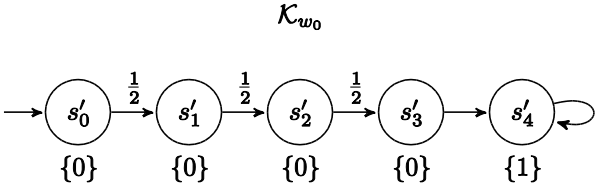

Fig. 3. Witness execution for the example program.

Another key idea is that a witness execution w_i is chosen for each group of executions $Execs(\mathrm{I\,B}_i)$ and check $\approx_p$ between w_i and every $\sigma \in Execs(\mathrm{I\,B}_i)$. According to lemma 1 (Section V), w_i and σ have relation $\approx_p$ if and only if their initial states, i.e., $w_i[0]$ and $\sigma[0]$ have relation $\approx_p$. This means the set of equivalence classes (quotient space) of combination of $Execs(\mathrm{I\,B}_i)$ and w_i with respect to $\approx_p$ can be computed. Then, $w_i[0]$ and every $\sigma[0]$ should belong to the same equivalence class. If not, then the program does not satisfy weak probabilistic noninterference and is not secure.

Back to the example, the set of initial states are partitioned. As there is only 1 initial state, just 1 block is obtained: $\mathrm{I\,B}_0 = \{s_0\}$. The execution $s_0 s_1 s_2 s_3 s_4^{\omega}$ is chosen as the witness and the state names are renamed, so that they are not confused with the states of $\mathrm{K_S}$. Thus, the witness execution is $w_0 = s'_0 s'_1 s'_2 s'_3 s'^{\omega}_4$ (Fig. 3). The quotient space of the combination of $\mathrm{K_S}$ and w_0 is computed. The quotient space has 12 blocks: $\{s_0\}$, $\{s_1\}$, $\{s_{13}\}$, $\{s_2\}$, $\{s_7\}$, $\{s_5, s_8, s_{10}, s_{17}\}$, $\{s_{14}\}$, $\{s_3, s'_3, s_{15}, s_{19}, s_{23}\}$, $\{s_{22}\}$, $\{s_6, s_9, s_{11}, s_{12}, s_{18}\}$, , $\{s_4, s'_4, s_{16}, s_{20}, s_{21}, s_{24}\}$, $\{s'_0, s'_1, s'_2,\}$. As s_0 and s'_0 do not belong to the same equivalence class, the verification algorithm returns insecure. This is what was expected.

III. PROGRAM MODEL

In this section, the program model assumed throughout the paper is introduced. Furthermore, some basic concepts concerning probability distributions, partitions, and equivalences are recalled.

A *probability distribution* μ over a set X is a function $\mu : X \to [0,1]$, such that $\sum_{x \in X} \mu(x) = 1$. The set of all probability distributions over X is denoted by $\mathrm{D}(X)$. The support of a probability distribution $\mu \in \mathrm{D}(X)$ is the set of all elements with a positive probability, i.e., $supp(\mu) = \{x \in X \mid \mu(x) > 0\}$.

A *partition* of a finite set S of states is a set $\{B_1, B_2, \ldots, B_n\}$ such that $B_i \neq \varnothing$, $\bigcup_i B_i = S$, and B_i are pairwise disjoint. B_i are called *blocks*. An equivalence relation R on S partitions S into the set of *equivalence classes*. The equivalence class of $s \in S$ w.r.t. R, denoted $[s]_R$, is defined as $[s]_R = \{s' \mid (s, s') \in R\}$. The set of equivalence classes of S w.r.t. R is called *quotient space*, denoted S / R.

Probabilistic Kripke structures are used to model operational semantics of probabilistic programs. Probabilistic Kripke structures are state transition systems that permit both probabilistic and nondeterministic choices. A state of a probabilistic Kripke structure indicates the current value of all low variables (shared memory of the multi-threaded program) together with the current value of the program counter that indicates the next program statement to be executed.

Definition 1 (Probabilistic Kripke Structure (PKS)): A *probabilistic Kripke structure* is a tuple $\mathrm{K} = (S, \to, \zeta, AP, La)$ where,

- S is a set of states,
- $\to \subseteq S \times \mathrm{D}(S)$ is a transition relation,
- ζ is an initial distribution such that $\sum_{s \in S} \zeta(s) = 1$,
- AP is a set of atomic propositions,
- $La : S \to 2^{AP}$ is a labeling function.

Here, atomic propositions are possible values of the low variables. K is called *finite* if S and AP are finite. The set I containing states s with $\zeta(s) > 0$ is considered as the set of *initial states*. The set of successor distributions of a state s is defined as $Post_D(s) = \{\mu \in \mathrm{D}(S) \mid s \to \mu\}$. The set of successor states of a state s is defined as $Post(s) = \cup_{\mu \in Post_D(s)} supp(\mu)$. A state s is called *terminal* if $Post(s) = \varnothing$.

Executions in a PKS K are alternating sequences of states that may arise by resolving both nondeterministic and probabilistic choices in K. More precisely, a finite *execution fragment* $\hat{\sigma}$ of K is a finite state sequence $s_0 s_1 \ldots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$. An *infinite* execution fragment σ is an infinite state sequence $s_0 s_1 s_2 \ldots$ such that $s_i \in Post(s_{i-1})$ for all $0 < i$. An execution fragment is called *initial* if it starts in an initial state, i.e., if $\zeta(s_0) > 0$. An *execution* of K is either an initial finite execution fragment that ends in a terminal state, or an initial infinite execution fragment.

Let $\sigma = s_0 s_1 s_2 \ldots$ be an execution and let $\sigma[0] = s_0$. $Execs(s)$ denotes the set of executions starting in s and $Execs(\mathrm{K})$ the set of executions of the initial states of K: $Execs(\mathrm{K}) = \bigcup_{s \in \mathrm{I}} Execs(s)$. Let $\mathrm{I}' \subseteq \mathrm{I}$; Then, $Execs(\mathrm{I}') = \bigcup_{s \in \mathrm{I}'} Execs(s)$.

A PKS with no non-determinism is called a fully probabilistic Kripke structure.

Definition 2 (Fully Probabilistic Kripke Structure (FPKS)): A PKS is called *fully probabilistic* if for each state there is at most one outgoing transition, i.e., $\forall s \in S : s \to \mu$ and $s \to \mu'$ implies $\mu = \mu'$.

FPKSs are state transition systems with probability distributions for transitions of each state. That is, the next state is chosen probabilistically, not non-deterministically. In the definition of FPKS, for convenience the transition relation $\to$ is replaced with a transition probability function $\mathbf{P} : S \times S \to [0,1]$. The function $\mathbf{P}$ determines for each state s the probability $\mathbf{P}(s, s')$ of a single transition from s to s'. The probability $\mathbf{P}(s, T)$ is defined as the probability of moving from s to some state $t \in T$ in a single step, i.e., $\mathbf{P}(s, T) = \sum_{t \in T} \mathbf{P}(s, t)$.

Reasoning about probabilities of sets of executions of a PKS relies on the resolution of the possible non-determinism in the PKS. This resolution is performed by a *scheduler*. A scheduler takes a finite execution (history of computation) as input and chooses the next transition to execute. Let $S^+ = \{s_1 s_2 \ldots s_k \mid k > 0 \text{ and each } s_i \in S\}$. Formally,

Definition 3 (Scheduler): Let $\mathrm{K} = (S, \to, \zeta, AP, La)$ be a PKS. A *scheduler* for K is a function $\mathrm{U} : S^+ \to \mathrm{D}(S)$, such that for all $\sigma = s_0 s_1 \ldots s_n \in S^+$, $\mathrm{U}(\sigma) \in Post_D(s_n)$.

A *finite-memory* scheduler denotes a scheduler that can be described by a deterministic finite automaton (DFA). Formally,

Definition 4 (Finite-memory scheduler): Let K be a PKS with state space S. A *finite-memory scheduler* S for K is a tuple $\mathsf{S} = (Q, \Delta, de, st)$ where,

- Q is a finite set of *modes*,
- $\Delta : Q \times S \to Q$ is a transition function,

- $de: Q \times S \to \mathrm{D}(S)$ is a decision function that selects the next transition $de(q,s)$ for any mode $q \in Q$ and state s of K ,
- $st: S \to Q$ is a function that selects a starting mode for state s of K .

The behavior of a PKS $\mathrm{K} = (S, \to, \zeta, AP, La)$ under a finite-memory scheduler $\mathsf{S} = (Q, \Delta, de, st)$ is as follows. At the beginning, an initial state s_0 is randomly chosen such that $\zeta(s_0) > 0$ and the DFA S is initialized to the mode $q_0 = st(s_0) \in Q$. Assuming that K is in state s and the current mode of S is q , the next transition is given by the decision function, i.e., $de(q,s) = \mu \in Post_D(s)$. Subsequently, the PKS randomly moves to the next state according to the distribution μ , while S changes mode to $\Delta(q,s)$.

As all nondeterministic choices in a PKS K are resolved by a finite-memory scheduler S , a fully probabilistic Kripke structure K_{S} is induced. The states in K_{S} are pair $< s, q >$ where s is a state in K and q a mode of S . Formally,

Definition 5: Let $\mathrm{K} = (S, \to, s_0, Var, La_{\mathrm{h}}, La_{\mathrm{l}})$ be a PKS and $\mathsf{S} = (Q, \Delta, de, st)$ be a finite-memory scheduler on K . **The FPKS of** K **induced by** S is given by

$$\mathrm{K}_{\mathsf{S}} = (S \times Q, \mathbf{P}, s_0, Var, La_{\mathrm{h}}', La_{\mathrm{l}}')$$

Where, $La_{\mathrm{h}}'(< s, q >) = La_{\mathrm{h}}(s)$, $La_{\mathrm{l}}'(< s, q >) = La_{\mathrm{l}}(s)$, and

$$\mathbf{P}(< s, q >, < s', q' >) = \begin{cases} \mu(s') & \text{if } s' \in Post(s),\ q' = \Delta(q,s), \\ & \text{and } \mu = de(q,s) \in Post_{\mathrm{D}}(s) \\ 0 & \text{otherwise.} \end{cases}$$

If K is a finite PKS, then K_{S} is finite too [20]. If K_{S} has a terminal state s_n , a transition $\mathbf{P}(s_n, s_n) = 1$ is included, ensuring that K_{S} has no terminal state. Therefore, all executions of K_{S} are infinite. It is assumed that the state space of the model of the multi-threaded program and the shared memory used by the threads are finite.

A *combination* operator $\oplus$ is defined to combine two FPKSs in a single FPKS. Let $\mathrm{K}_i = (S_i, \mathbf{P}_i, \zeta_i, AP, La_i)$, $i = 1, 2$ be two FPKSs. The combination of K_1 and K_2 is defined as $\mathrm{K}_1 \oplus \mathrm{K}_2 = (S_1 \hat{a}\ S_2, \mathbf{P}_1 \hat{a}\ \mathbf{P}_2, \zeta_1 \hat{a}\ \zeta_2, AP, La)$ where â stands for disjoint union and $La(s) = La_i(s)$ if $s \in S_i$.

IV. Specifying Weak Probabilistic Non-interference

A multi-threaded program is secure when a *variation* of the values of the high variables does not influence the low-observable behavior of the program [6]. Thus, low-observable behavior of the program should be *indistinguishable* as high variables are varied. Variation of the values of high variables is represented by low-equivalence relation. Two states s_1 and s_2 are *low-equivalent*, denoted $s_1 =_L s_2$, if $La(s_1) = La(s_2)$. Smith [7] uses the notion of *weak probabilistic bisimulation* to represent the indistinguishability of low-observable behavior of the program.

Weak probabilistic bisimulation abstracts from steps that remain inside the equivalence classes, i.e., it does not care which state within the equivalence class the system is in [21]. Let $\mathrm{K} = (S, \mathbf{P}, \zeta, AP, La)$ be an FPKS and $R \subseteq S \times S$ be an equivalence relation. State s is *silent* with respect to R , if $\mathbf{P}(s, [s]_R) = 1$, i.e., s does not have a successor state outside $[s]_R$. Any state that is not silent with respect to R , may leave its equivalence class by a single transition with positive probability. Let S_{silent}^R denote the set of silent states with respect to R . For any state $s \notin S_{silent}^R$ and $C \in S / R$ with $C \neq [s]_R$

$$\mathbf{P}_c(s, C) = \frac{\mathbf{P}(s, C)}{1 - \mathbf{P}(s, [s]_R)}$$

denotes the conditional probability for non-silent state s to reach block C under the condition that being in s the system does not make a move inside $[s]_R$.

Definition 6 (Weak probabilistic bisimulation) [21], [22]: Let $\mathrm{K} = (S, \mathbf{P}, \zeta, AP, La)$ be an FPKS. A *weak probabilistic bisimulation* for K is an equivalence relation R on S such that for all $s_1\ R\ s_2$:

- $s_1 =_L s_2$.
- If $\mathbf{P}(s_i, [s_i]_R) < 1$ for $i = 1, 2$ then for each equivalence class $C \in S / R$, $C \neq [s_1]_R = [s_2]_R$:
- $\mathbf{P}_c(s_1, C) = \mathbf{P}_c(s_2, C)$.
- s_1 can reach a state outside $[s_1]_R$, iff s_2 can reach a state outside $[s_2]_R$.

States s_1 and s_2 are weak probabilistic bisimilar, denoted as $s_1 \approx_p s_2$, if there exists weak probabilistic bisimulation R for K such that $s_1 R s_2$.

Condition (1) asserts that states s_1 and s_2 are low-equivalent, and condition (2) ensures that their conditional probability to move to another equivalence class is the same. According to condition (3) for any equivalence class C , either for all $s \in C$: $s \in S_{silent}^R$ or for all $s \in C$ there is an execution fragment $\hat{\sigma} = s_0 s_1 \ldots s_n$ starting in $s = s_0$ with $n \geq 0$, $s_i \in C$ for $i = 1, \ldots, n-1$ and $s_n \notin C$.

Weak probabilistic bisimulation for pairs of executions is defined as follows:

Definition 7 (Weak probabilistic bisimilar executions): For infinite execution fragments $\sigma_i = s_{0,i}s_{1,i}s_{2,i}\ldots$, $i=1,2$ in K, σ_1 is weak probabilistic bisimilar to σ_2, denoted $\sigma_1 \approx_p \sigma_2$ if and only if there exists an infinite sequence of indices $0=j_0<j_1<j_2<\ldots$ and $0=k_0<k_1<k_2<\ldots$ with:

$s_{j,1} \approx_p s_{k,2}$ for all $j_{r-1} \le j < j_r$ and $k_{r-1} \le k < k_r$ with $r=1,2,\ldots$

In other words, two executions are weak probabilistic bisimilar, if they run through the same sequence of equivalence classes under $\approx_p$.

Smith [7] states that if a secure program is run starting from two low-equivalent states, then two executions must pass through the same sequence of equivalence classes. This is captured formally by the definition of weak probabilistic noninterference.

Definition 8 (Weak probabilistic noninterference): Given a finite-memory scheduler S, a multi-threaded program *MT* satisfies *weak probabilistic noninterference*, iff

$$\forall \sigma,\sigma' \in Execs(\mathrm{K}_\mathsf{S}).\ \sigma[0] =_L \sigma'[0] \ \Rightarrow\ \sigma \approx_p \sigma'$$

Where, K_S denotes an FPKS, modeling the executions of the program *MT* under the scheduler S, $=_L$ is low-equivalence relation between states, and $\approx_p$ is weak probabilistic bisimulation relation.

The intuition is that low-equivalent executions must visit the same sequence of equivalence classes of $\approx_p$, but some executions may run slowly than the others.

V. Verification

In this section an algorithm is developed to verify weak probabilistic noninterference. In what follows, a finite fully probabilistic Kripke structure $\mathrm{K}_\mathsf{S}=(S,\mathbf{P},\zeta,AP,La)$ is fixed which models the executions of a multi-threaded program *MT* under a scheduler S. Let I denote the set of initial states of K_S, i.e., set of states s with $\zeta(s)>0$.

A. The Algorithm

Weak probabilistic noninterference requires that all executions of the program with low-equivalent initial states must be weak probabilistic bisimilar. To verify this, the set of initial states I of K_S is partitioned into initial state blocks $\mathrm{I\,B}_0,\ldots,\mathrm{I\,B}_m$. Each initial state block contains all low-equivalent initial states. Then, an arbitrary witness execution $w_i \in Execs(\mathrm{I\,B}_i)$ is chosen for each $\mathrm{I\,B}_i$ ($i\in\{0,1,\ldots,m\}$) and FPKS K_{w_i} is created from w_i. K_S and all K_{w_i} are combined to form FPKS $\mathrm{K}'=(S',\mathbf{P}',\zeta',AP,La')$: $\mathrm{K}'=\mathrm{K}_\mathsf{S}\oplus\mathrm{K}_{w_0}\oplus\ldots\oplus\mathrm{K}_{w_m}$. Now, K_S satisfies weak probabilistic noninterference if and only if $w_i[0]$ and all states of $\mathrm{I\,B}_i$ belong to the same equivalence class in the quotient space $\mathrm{K}'/\approx_p$, i.e., $\mathrm{I\,B}_i \subseteq [\,w_i[0]\,]_{\approx_p}$, where $[\,w_i[0]\,]_{\approx_p}$ denotes the equivalence class of $w_i[0]$ w.r.t. $\approx_p$.

The main steps of the verification algorithm are sketched in Algorithm 1. The algorithm takes a finite FPKS as input and returns secure if the FPKS satisfies weak probabilistic noninterference, and insecure if it does not. In the sequel, some steps of the algorithm are explained in more detail.

Taking a witness execution: As pointed out earlier, all executions of the input FPKS are infinite and hence form a cycle. To take a witness execution, a cycle detection algorithm based on depth-first search, called colored DFS, is used. The algorithm initially marks all states white. It then proceeds by moving to successor states and coloring them, and terminates when a colored state (i.e. a state that was encountered before) is visited. The sequence of states remains in the stack of the depth-first search form the witness execution.

Algorithm 1. Verification of Weak Probabilistic Non-interference

Input: finite FPKS K_S
Output: *secure*, if the program satisfies weak probabilistic noninterference;
insecure, otherwise;

Partition I into $\mathrm{I\,B}_0,\ldots,\mathrm{I\,B}_m$;
Take a witness execution $w_i \in Execs(\mathrm{I\,B}_i)$ ($i\in\{0,1,...,m\}$);
Build FPKS K_{w_i} for each w_i;
Build FPKS $\mathrm{K}'=\mathrm{K}_\mathsf{S}\oplus\mathrm{K}_{w_0}\oplus\ldots\oplus\mathrm{K}_{w_m}$;
Compute the quotient space $\mathrm{K}'/\approx_p$;
for each $\mathrm{I\,B}_i$ and the corresponding witness w_i do
  if $\mathrm{I\,B}_i \not\subseteq [\,w_i[0]\,]_{\approx_p}$ **then**
    return *insecure*;
  end if
end for
return *secure*;

Computing the quotient space w.r.t. $\approx_p$: Equivalence classes w.r.t. $\approx_p$ are computed using an approach similar to that of Baier and Hermanns [21]. The general idea of the computation algorithm is to use an iterative partition refinement technique. It starts from a trivial initial partition, where each block of the partition contains all low-equivalent states (condition (1) of the definition 6). It then successively refines the given partition by splitting any block of the partition into sub-blocks, eventually resulting in the set of weak probabilistic bisimulation equivalence classes. A general schema of the iterative refinement is depicted in Fig. 4.

The main idea for splitting each block B of the partition is to isolate non-silent states $s,s'\in B$ with equivalent conditional probability to some other block C, i.e. $\mathbf{P}_c(s,C)=\mathbf{P}_c(s',C)$, in order to ensure condition (2) of the definition 6. By condition (3) of the definition, each such non-silent isolated subblock $A\subseteq B$ has to be enriched with those silent states of B, which produce execution fragments that remain inside B and end up in A. Fig. 5 shows how B is refined into two subblocks.

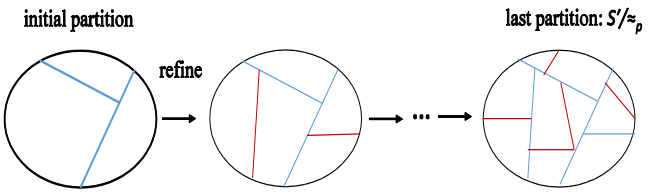

Fig. 4. Successive partition refinement to compute the quotient space.

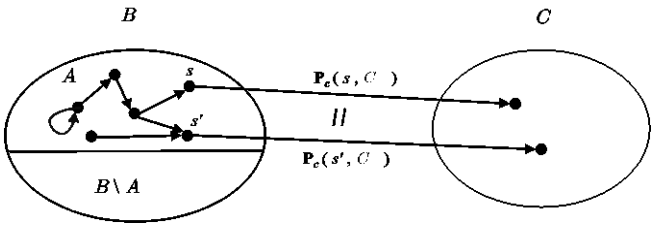

Fig. 5. Refinement of the block *B* into *A* and *B\A*.

B. Correctness of the Algorithm

Before diving into proving correctness of the proposed algorithm, a lemma is presented, which will be used in the correctness proof. This lemma asserts that $\approx_p$ can be lifted from states to executions and vice versa.

Lemma 1. Weak probabilistic bisimilar states have weak probabilistic bisimilar executions and vice versa:

$$s_1 \approx_p s_2 \text{ iff } \forall \sigma_1 \in Execs(s_1), \sigma_2 \in Execs(s_2).\ \sigma_1 \approx_p \sigma_2$$

Proof: "if": Let $\sigma_1 = s_{0,1}s_{1,1}s_{2,1}\ldots \in Execs(s_1)$ starting in $s_1 = s_{0,1}$ and $\sigma_2 = s_{0,2}s_{1,2}s_{2,2}\ldots \in Execs(s_2)$ starting in $s_2 = s_{0,2}$. By definition 7, if two executions are weak probabilistic bisimilar, then their initial states are weak probabilistic bisimilar too. Thus, $s_1 \approx_p s_2$.

"only if": Let $s_1 \approx_p s_2$ and $\sigma_1 = s_{0,1}s_{1,1}s_{2,1}\ldots \in Execs(s_1)$. Successively, a weak probabilistic bisimilar execution σ_2 starting in s_2 is defined by lifting the transitions from $s_{i,1}$ to $s_{i+1,1}$ with $s_{i,1} \not\approx_p s_{i+1,1}$ to finite execution fragments $s_{i,2}u_{i,1}\ldots u_{i,n_i}s_{i+1,2}$ (Fig. 6) such that:

$$s_{i,1} \approx_p s_{i,2},\ s_{i+1,1} \approx_p s_{i+1,2}, \text{ and } s_{i,2} \approx_p u_{i,1} \approx_p \ldots \approx_p u_{i,n_i}.$$

The proof is by induction on i. The base case $i = 0$ is straightforward and omitted. Assume $i \geq 0$ and that the execution fragment

$$\sigma_2 = s_{0,2}u_{0,1}\ldots u_{0,n_0}s_{1,2}u_{1,1}\ldots u_{1,n_1}s_{2,2}\ldots s_{i,2}$$

is already constructed. Distinguish two cases:

1) $s_{i,1} \not\approx_p s_{i+1,1}$. Since $s_{i,1} \approx_p s_{i,2}$ and $\mathbf{P}(s_{i,1}, s_{i+1,1}) > 0$, there exists a finite execution fragment $\hat{\sigma}_2' = s_{i,2}u_{i,1}\ldots u_{i,n_i}s_{i+1,2}$ such that: $s_{i+1,1} \approx_p s_{i+1,2}$ and $s_{i,2} \approx_p u_{i,1} \approx_p \ldots \approx_p u_{i,n_i}$. Concatenating the execution fragment $\hat{\sigma}_2$ with the execution fragment $\hat{\sigma}_2'$ yields an execution fragment that fulfills the desired conditions.

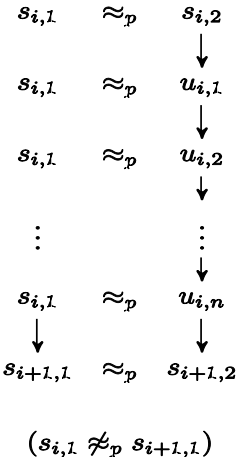

Fig. 6. Construction of a weak probabilistic bisimilar execution.

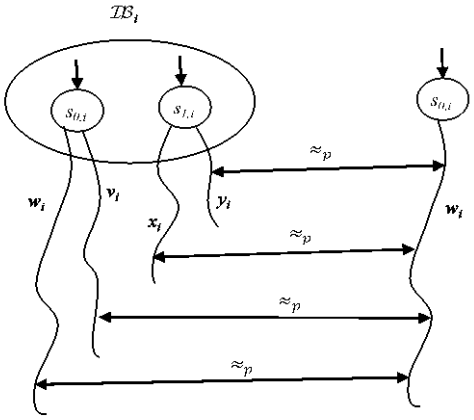

Fig. 7. Relation between the witness execution w_i and other executions of $\mathrm{I}\,\mathrm{B}_i$.

2) $s_{i,1} \approx_p s_{i+1,1}$. Distinguish between $s_{i,1}$ can reach outside $[s_{i,1}]_{\approx_p}$ and cannot reach outside $[s_{i,1}]_{\approx_p}$:

a) $s_{i,1}$ can reach outside $[s_{i,1}]_{\approx_p}$, i.e., there exists an index $j > i+1$ with $s_{i,1} \not\approx_p s_{j,1}$. Without loss of generality, assume that j is minimal, i.e., $s_{i,1} \approx_p s_{i+1,1} \approx_p \ldots \approx_p s_{j-1,1}$ $s_{j-1,1} \not\approx_p s_{j,1}$. As $s_{i,2} \approx_p s_{j-1,1}$ and $\mathbf{P}(s_{j-1,1}, s_{j,1}) > 0$, there exists a finite execution fragment $\hat{\sigma}_2' = s_{i,2}u_{i,1}\ldots u_{i,n_i}s_{i+1,2}$ such that $s_{j,1} \approx_p s_{i+1,2}$ and $s_{i,2} \approx_p u_{i,1} \approx_p \ldots \approx_p u_{i,n_i}$. Concatenation of the execution fragment $\hat{\sigma}_2$ with the execution fragment $\hat{\sigma}_2'$ yields an execution fragment that fulfills the desired conditions.

b) $s_{i,1}$ cannot reach outside $[s_{i,1}]_{\approx_p}$, i.e., $s_{i,1} \approx_p s_{j,1}$ for all $j \geq i$. As $s_{i,1} \approx_p s_{i,2}$, and $s_{i,1}$ cannot reach outside $[s_{i,1}]_{\approx_p}$, $s_{i,2}$ cannot reach outside $[s_{i,2}]_{\approx_p}$ (see condition (3) of definition 6), i.e., there is an execution $s_{i,2}s_{i+1,2}s_{i+2,2}\cdots$ with $s_{i,2} \approx_p s_{i+1,2} \approx_p s_{i+2,2} \approx_p \ldots$. Concatenating the execution

fragment $\hat{\sigma}_2$ with the execution fragment $s_{i,2}s_{i+1,2}s_{i+2,2}\ldots$ yields an execution that fulfills the desired conditions. Consequently, the resulting execution σ_2 is weak probabilistic bisimilar to σ_1.

The following theorem proves correctness of the Algorithm 1.

Theorem 1. Algorithm 1 returns secure if and only if the input FPKS K_S satisfies weak probabilistic non-interference.

Proof: The algorithm starts by partitioning I into low-equivalent sets of states $IB_0,\ldots,IB_m$. Then, a witness execution $w_i \in Execs(IB_i)$ is chosen and Kripke structures $K_{w_0},\ldots,K_{w_m}$, and K' are created. Now, the problem of K_S satisfying weak probabilistic noninterference is reduced to checking weak probabilistic bisimilarity between w_i and each $\sigma \in Execs(IB_i)$. For example, in Fig. 7, the relation $\approx_p$ should be established between the witness execution w_i and all executions in $Execs(IB_i)$, i.e., w_i, v_i, x_i, y_i.

According to lemma 1, w_i and σ are weak probabilistic bisimilar if and only if their initial states, i.e., $w_i[0]$ and $\sigma[0]$ are weak probabilistic bisimilar:

$$w_i \approx_p \sigma \text{ iff } w_i[0] \approx_p \sigma[0]$$

Given that $\approx_p$ was defined as an equivalence relation, $w_i[0] \approx_p \sigma[0]$ if and only if $w_i[0]$ and $\sigma[0]$ belong to the same equivalence class in the quotient space $K'/\approx_p$. Thus, each $\sigma[0]$, i.e., all states of IB_i, should belong to $[w_i[0]]_{\approx_p}$. In other words, for each initial state block IB_i and the corresponding witness execution w_i, it should be $IB_i \subseteq [w_i[0]]_{\approx_p}$.

C. Complexity of the Algorithm

For computing the initial state blocks, *HashMap* class of Java was used. The worst case complexity of inserting a key-value pair to the hash map is $O(|AP|)$. Hence, the time complexity of computing the initial state blocks is $O(|I|.|AP|)$.

Let t be the number of transitions of K_S. A witness execution can be extracted in time $O(t+|S|)$. Thus, the time complexity of extracting all witness executions is $O(|I|.(t+|S|))$.

The quotient space $K'/\approx_p$ can be constructed in time $O(|S'|^3)$ [21]. Assuming $|S| \leq t$ and considering the fact that $\frac{|S'|}{2} \leq |S|$, verification of weak probabilistic noninterference can be implemented in time $O(|S|^3 + |I|.(t+|AP|))$.

D. Case Study

The algorithm proposed in this paper has been implemented as part of *SCT* (Security Certifying Tool), which has been developed in JAVA to verify secure information flow for multi-threaded programs. SCT gets a probabilistic Kripke structure as model of the program and checks whether the program satisfies weak probabilistic noninterference. To our knowledge, no other algorithmic verification technique for weak probabilistic noninterference has been published, so it is not possible to compare the implementation to other algorithms.

As a case study, consider the problem of *dining cryptographers*. The problem is borrowed from [11] to show how an attacker can deduce secret information through probabilistic leaks. David Chaum first proposed this problem in 1988 as an example of anonymity and identity hiding [23]. In this problem, three 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 an unbiased coin and only informs the cryptographer on the right of the outcome. The situation is illustrated in Fig. 8. In this figure, c1, c2, and c3 are identities of cryptographer 1, cryptographer 2, and cryptographer 3 respectively.
- 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.

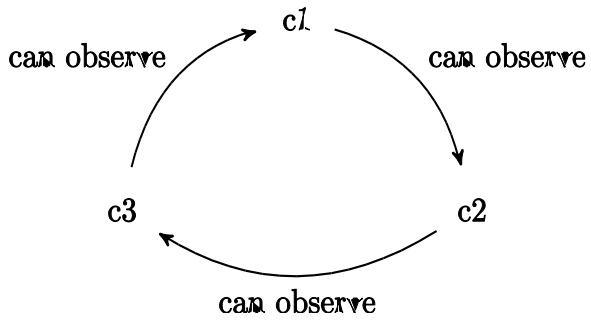

Fig. 8. Dining cryptographers. c1 can observe c2's coin, and c2 can observe c3's coin.

An even number of 'agree's implies that none of the cryptographers paid (the master paid), while an odd number implies that one of the cryptographers paid. David Chaum names this protocol as *Dining Cryptographers network* or DC-net. DC-net is secure, since it does not leak the identity of the paying cryptographer (in case one of the cryptographers made arrangement to pay for the meal). Following Ngo [11], to make this protocol leak information, a slight change is done: coins are *biased*, i.e., with probability 0.6 it comes up heads, and with probability 0.4 it comes up tails.

To model the case study, *PRISM* has been used. PRISM is a tool for formal modeling and analysis of probabilistic systems [24]. PRISM describes models using the PRISM language, a simple, state-based language with a guarded command notation. The program is implemented in PRISM and its model is built. Then, export the explicit-state model, containing the set of reachable states and their labels, along with the transition matrix. Then, the model is given to SCT to compute the quotient space and check the security property. SCT was run on a PC with a Core i3 2.53 GHz CPU and 6 GB RAM.

Without lack of generality, suppose one of the cryptographers has made arrangement for the meal, and the other one is the attacker, i.e., the one who tries to find out the payer's identity. The FPKS K_S of the model built by PRISM has 285 states and 582 transitions. K_S has just 3 initial states. All initial states have the same label value of $\{0\}$ (label values are explained in the next paragraph). Thus, a witness execution w_0 is extracted from K_S and $K' = K_S \oplus K_{w_0}$ is built. K' has 292 states and 589 transitions. The quotient space $K'/\approx_p$ is computed in 1.672 seconds and has 13 equivalence classes. As expected, the initial states and $w_i[0]$ do not belong to same equivalence class and hence SCT correctly recognizes the model as insecure.

To see how an attacker can infer the identity of the payer, consider an example scenario where cryptographer 2 is the attacker and aims to find out which one of the cryptographers 1 or 3 is the payer. Suppose cryptographer 2 and cryptographer 3 both toss tail. Cryptographer 2 can observe the coin of cryptographer 3, and thus announces 'agree'. Assume cryptographer 2 observes that cryptographer 1 announces 'agree' and cryptographer 3 announces 'disagree' for the values of the coins. Two situations corresponding to this case are shown in Fig. 9 and executions of these situations are outlined in Fig. 10. In Fig. 10, each state is represented as 10-tuples listing the current values of the variables (*pay, agree1, agree2, agree3, coin1, s1, coin2, s2, coin3, s3*) and labeled with the current value of parity: 0 for even number of 'agree's, and 1 for odd number of 'agree's. The variable *pay* contains the number of the cryptographer who is actually the payer. Variables *agree1, agree2,* and *agree3* contain the announcements of cryptographer 1, 2, and 3, respectively: 0 for 'disagree', and 1 for 'agree'. Variables *coin1, coin2,* and *coin3* contain the coin values for cryptographer 1, 2, and 3, respectively: 1 for head, and 2 for tail. Finally, variables *s1, s2,* and *s3* contain the status values for the three cryptographers: 0 for 'not done', and 1 for 'done'.

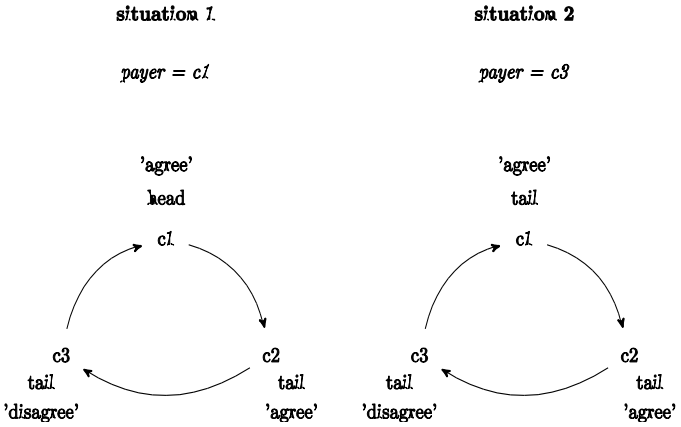

Fig. 9. Two situations corresponding to the case where c2 and c3 both toss tail.

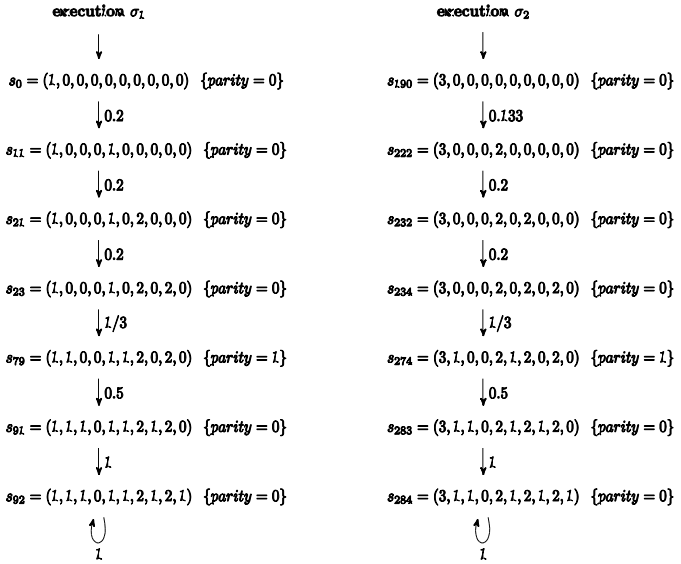

Fig. 10. Two executions corresponding to the situations 1 and 2.

Execution σ_1 occurs when cryptographer 1 is the payer and tosses head. Therefore, cryptographer 1 announces 'agree' and cryptographer 3 announces 'disagree'. Execution σ_2 occurs when cryptographer 3 is the payer and tosses tail. Thus, cryptographer 3 announces 'disagree' and cryptographer 1 announces 'agree'. As seen in Fig. 10, the probability of σ_1 (i.e. cryptographer 1 tossing head) is more than the probability of σ_2 (i.e. cryptographer 1 tossing tail) and hence the attacker can deduce that cryptographer 1 is more likely to be the payer. This is a probabilistic leak.

VI. Related Work

In the following, some related approaches from the literature are discussed and the proposed approach is compared with them.

Barthe et al. [10] propose the idea of self-composition for logical characterization of information flow properties. Self-composition reduces the problem of verifying information flow property for a program *P* to a safety property for a program derived from *P*, by composing *P* with a renaming of itself. Then, standard model checking and algorithmic verification techniques can be used to verify secure information flow. Terauchi and Aiken [14] introduce 2-safety properties, which can be refuted by observing two executions. They show that termination insensitive secure information flow problem is a 2-safety problem. They further generalize the idea of self-composition and show that it can be used to verify 2-safety properties. Huisman et al. [15] use the idea of self-composition to characterize secure information flow in CTL* and modal μ-calculus temporal logics. They specify secure information flow using observational determinism, an information flow property proposed by Zdancevic and Myers [25] for concurrent programs. Van der Meyden and Zhang [16] employ a self-composition-like method to reason about noninterference properties and develop algorithmic verification techniques for these properties. They characterize the computational complexity of the developed verification techniques and discuss some possible heuristics for optimizing the verification. Verification methods that use the idea of self-composition suffer from the state-space explosion problem, i.e., space needed to store the states and transitions of the program exceed the available memory. This occurs because in self-composition a program model is composed with a copy of itself. In the proposed algorithm, the program model is composed with only a small part of the model (witness execution). Furthermore, security analysis is done on the abstract model (quotient space), not on the concrete model.

Ngo et al. [26] propose scheduler-specific probabilistic observational determinism as a property to specify secure information flow for probabilistic multi-threaded programs. They define the property based on two conditions. First condition requires that all traces of each public variable starting in the same initial state are stuttering equivalent. A trace of an execution is a mapping of states of the execution to the corresponding state labels. Two traces are stuttering equivalent if they become the same after removing repeating adjacent labels. Second condition requires that for all traces of an initial state s_i, there exists a trace of an initial state $s_i^{'}$ low-equivalent to s_i, that is stuttering equivalent to each one of the traces of s_i and the probabilities of the traces are the same. Condition 2 of this property is closest in semantics to our definition of weak probabilistic noninterference. Of course, weak probabilistic noninterference requires weak probabilistic bisimulation between executions, which is different from stuttering equivalence. To verify condition 2 of their property, Ngo et al build two FPKSs for each pair of initial states s_i and $s_i^{'}$. Then, they transform the FPKSs to stuttering-free ones and check equivalence of the probabilistic languages arising from executions of the two FPKSs using an off-the-shelf algorithm. The time complexity of the algorithm is $O\left(n^3\right)$ for each pair of initial states s_i and $s_i^{'}$, where n is the number of states of each FPKS. The deficiency of this verification algorithm is that it builds two copies of the program for each pair of initial states. It is clear that if the input program has enormous state space, then the algorithm would suffer from the state explosion problem.

A trending field in security verification is proof-based verification, in which mathematical logic is used to describe the program, specify the property of interest, and *prove* satisfiability of the property. Hoare logic [27] is one of the most widely-used logics for proof-based verification of software. Variants of Hoare logic have been proposed for verifying relational, and in particular, k-safety properties [28-30]. An advantage of these techniques is that they avoid the state-space explosion problem, because they do not check the whole state space of the program. Consequently, they are suitable for verifying programs with huge, and even infinite, state space. A disadvantage with these techniques is that they are semi-automatic. Although many of the proof steps are done mechanically, some steps need expert user intervention. This contrasts with algorithmic verification, which is fully automatic.

VII. Conclusions and Future Work

In this paper, the problem of verifying weak probabilistic noninterference was discussed. Weak probabilistic noninterference is a notion of confidentiality for multi-threaded programs. The behavior of multi-threaded programs running under the control of a scheduler was modeled by probabilistic Kripke structures. Weak probabilistic noninterference was formalized in terms of executions of the probabilistic Kripke structure. Then, a verification algorithm was proposed to check the property.

As future work, we plan to use the proposed algorithm to verify other information flow properties. We believe the applicability of the algorithm can be extended and it can be used to verify many security properties, such as strong security [6] and probabilistic noninterference [6]. In an earlier paper [31], we used a similar algorithm to verify observational determinism.

A disadvantage of the proposed verification algorithm is that it works on explicit model of the program, which may be too huge for real-world programs. This harms scalability of the approach. To solve this problem, one can change the algorithm in such a way that it works on abstract models of the program, such as binary decision diagrams.

We also aim to modify the algorithm to support compositional verification, thereby reducing conceptual complexity and making the analysis scale.

References

[1] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” IEEE Journal on Selected Areas in Communications, vol. 21, 2003, pp. 5–19.

[2] T. M. Ngo and M. Huisman, "Complexity and information flow analysis for multi-threaded programs," The European Physical Journal Special Topics, 2017, pp. 1-18.

[3] D. E. Denning, "A lattice model of secure information flow," Communications of the ACM, vol. 19, 1976, pp. 236–243.

[4] J. A. Goguen and J. Meseguer, "Security policies and security models," in IEEE Symposium on Security and Privacy, 1982, pp. 11–20.

[5] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," Journal of Computer Security, vol. 7, 1999, pp. 231–253.

[6] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in Proceedings of the 13th IEEE workshop on Computer Security Foundations, CSFW'00, 2000, pp. 200–214.

[7] G. Smith, "Probabilistic noninterference through weak probabilistic bisimulation," in Proceedings of the 16th IEEE workshop on Computer Security Foundations, CSFW'03, 2003, pp. 3–13.

[8] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," Journal of Computer Security, vol. 4, 1996, pp.:167–187.

[9] G. Smith, "A new type system for secure information flow," in Proceedings of the 14th IEEE workshop on Computer Security Foundations, CSFW'01, 2001, pp. 115–125.

[10] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by selfcomposition," in Proceedings of the 17th IEEE workshop on Computer Security Foundations, CSFW'04, 2004, pp. 100–114.

[11] T. M. Ngo. Qualitative and quantitative information flow analysis for multi-thread programs, PhD thesis, University of Twente, 2014.

[12] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. "Decomposition Instead of Self-Composition for k-Safety," To appear in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017.

[13] N. Grimm, K. Maillard, C. Fournet, C. Hritcu, M. Maffei, J. Protzenko, A. Rastogi, N. Swamy, and S. Zanella-Beguelin, "A Monadic Framework for Relational Verification (Functional Pearl)," arXiv preprint arXiv:1703.00055, 2017.

[14] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in Proceedings of the 12th International Static Analysis Symposium, SAS'05, 2005, pp. 352–367.

[15] M. Huisman, P. Worah, and K. Sunesen, "A temporal logic characterisation of observational determinism," in Proceedings of the 19th IEEE workshop on Computer Security Foundations, CSFW'06, 2006.

[16] R. van der Meyden and C. Zhang, "Algorithmic verification of noninterference properties," in Proceedings of the Second International Workshop on Views on Designing Complex Architectures, VODCA'06, 2007, pp. 61–75.

[17] P. Cerny and R. Alur, "Automated analysis of java methods for confidentiality," in Proceedings of the 21st International Conference on Computer Aided Verification, CAV'09, 2009, pp. 173–187.

[18] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, "Checking Probabilistic Noninterference Using JOANA," IT - Information Technology, Vol. 56, 2014, pp. 280-287.

[19] D. D'Souza and K. R. Raghavendra, "Model-checking trace-based information flow properties for infinite-state systems," Journal of Computer Security, (Preprint), 2016, pp. 1–27.

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

[21] C. Baier and H. Hermanns, "Weak bisimulation for fully probabilistic processes," in Proceedings of the 9th International Conference on Computer Aided Verification, CAV'97, 1997, pp. 119–130.

[22] C. Baier, J. P. Katoen, H. Hermanns, and V. Wolf, "Comparative branching-time semantics for markov chains," Information and computation, vol. 200, 2005, pp. 149–214.

[23] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," Journal of cryptology, vol. 1, 1988, pp. 65–75.

[24] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11, vol. 6806 of LNCS, 2011, pp. 585–591.

[25] S. Zdancewic and A. C Myers, "Observational determinism for concurrent program security" in Proceedings of the 16th IEEE Computer Security Foundations Workshop, CSFW'03, 2003, pp. 29–43.

[26] T. M. Ngo, M. Stoelinga, and M. Huisman, "Confidentiality for probabilistic multi-threaded programs and its verification," in Proceedings of the 5th international conference on Engineering Secure Software and Systems, ESSoS'13, 2013, pp. 107–122.

[27] C. A. R. Hoare, "An axiomatic basis for computer programming," Communications of the ACM, vol. 12, 1969, pp. 576–580.

[28] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016, pp. 57–69.

[29] G. Barthe, J. M. Crespo, and C. Kunz, "Product programs and relational program logics," Journal of Logical and Algebraic Methods in Programming, vol. 85.5, 2016, pp. 847-859.

[30] A. Banerjee, D. A. Naumann, and M. Nikouei, "Relational Logic with Framing and Hypotheses," in Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2016, pp. 11:1-11:16.

[31] J. Karimpour, A. Isazadeh, and A. A. Noroozi, "Verifying Observational Determinism," in Proceedings of the 30th IFIP International Information Security Conference, SEC'15, 2015, pp. 82-93.