Noninterfering Schedulers

When Possibilistic Noninterference Implies Probabilistic Noninterference

Andrei Popescu, Johannes Hölzl, Tobias Nipkow

Technische Universität München

Background: Language-Based Security

- Program state separated into a private part and a public part
- Variables partitioned into low (public) and high (private)

  In this talk: low = public, high = private
- Noninterference (Denning and Denning 1977, Volpano, Smith and Irvine 1996):

  Running the program and inspecting the low variables during/after the execution does not reveal anything about the initial value of the high variables

  In this talk: Security = Noninterference

Background: Variants of Concurrent Language-Based Security

Assuming states s and s' equal on low variables:

- Possibilistic (nondeterministic) security:

  $\forall$ e $\in$ Exec(s). $\exists$ e' $\in$ Exec(s').

  low_obs(e) = low_obs(e')

- Probabilistic security:

  distrib { low_obs(e). e $\in$ Exec(s) } =

  distrib { low_obs(e'). e' $\in$ Exec(s') }

Examples

Is the whole program secure?

l is a low variable and h is a high variable

Thread1: while h > 0 do {h := h − 1} ; l := 2 Thread2: l := 1

Secure possibilistically

Not secure for the uniform probabilistic scheduler

Prob (final l = 2) increases with the initial value of h

Examples

Is the whole program secure?

l is a low variable and h is a high variable

Thread1: h := 1 ; l := 2 Thread2: l := 1

Secure for all schedulers

No interesting computation with high variables

Examples

Is the whole program secure?

l is a low variable and h is a high variable

Thr1: while $h > 0$ do $\{h := h - 1\}$ Thr2: $l := 2$ Thr3: $l := 1$

Secure for some schedulers: uniform, round robin, ...

Not secure for all schedulers

Depending on termination of Thr1, Thr2 or Thr3 is taken first

Problem Setting

- Concurrent multi-threaded language
- Arbitrary probabilistic scheduler

What properties of the individual threads and of the scheduler guarantee that the multi-threaded program as a whole is secure?

Solutions from the Literature

thread
freedom observational
determinism
probabilities
allowed Sands
large depend
strong time Operational synchronization
primitives Zdancewic all
class creation run
race assumed
semantics Mantel
runs Sudbrock
Myers private
Boudol independently
terminating Sabelfeld dynamic
scheduler
Threads
Nonstandard Russo
robust Castellani
data customized
Security
schedulers

Word It Out

Limitations of Existing Solutions

Do not cope with dynamic thread creation

or

Do not allow thread running times to depend on private data

or

Offer weak security guarantees on the whole program

or

Rely on race freedom or termination analysis

or

Use non-standard thread-level synchronization primitives

Our Solution

Do not cope with dynamic thread creation

or

Do not allow thread running times to depend on private data

or

Offer weak security guarantees on the whole program

or

Rely on race freedom or termination analysis

or

Use non-standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Do not allow thread running times to depend on private data

or

Offer weak security guarantees on the whole program

or

Rely on race freedom or termination analysis

or

Use non-standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Allows thread running times to depend on private data

Offer weak security guarantees on the whole program

or

Rely on race freedom or termination analysis

or

Use non-standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Allows thread running times to depend on private data

Offers stronger security guarantees on the whole program

– security during execution, not only on the final state

Rely on race freedom or termination analysis

or

Use non-standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Allows thread running times to depend on private data

Offers stronger security guarantees on the whole program

- security during execution, not only on the final state

Relies on no race freedom or termination analysis

- nonterminating systems (web servers, operating systems)

Use non-standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Allows thread running times to depend on private data

Offers stronger security guarantees on the whole program

- security during execution, not only on the final state

Relies on no race freedom or termination analysis

- nonterminating systems (web servers, operating systems)

Uses standard thread-level synchronization primitives

Our Solution

Copes with dynamic thread creation

Allows thread running times to depend on private data

- but does not cover all schedulers

Offers stronger security guarantees on the whole program

- security during execution, not only on the final state

Relies on no race freedom or termination analysis

- nonterminating systems (web servers, operating systems)

Uses standard thread-level synchronization primitives

Overview of our Solution

- Study the scheduler behavior in isolation from the concrete operational semantics of the threads
- Identify for the scheduler a form of system security in the style of Goguen and Meseguer

Theorem

secure scheduler

+

possibilistically secure program

Overview of our Solution

- Study the scheduler behavior in isolation from the concrete operational semantics of the threads
- Identify for the scheduler a form of system security in the style of Goguen and Meseguer

Theorem

secure scheduler

+

possibilistically secure program

$\Rightarrow$

probabilistically secure program

Possibilistically Secure Program

- Thread semantics:

  $\text{step} : \text{State} \times \text{Thread} \rightarrow \text{State} \times \text{Thread}_{\bot} \times \text{List (Thread)}$

- A thread is low if it may still modify the low part of the state
- Possibilistic security of the program: If starting in alternative states with the same low part, the executions proceed in the same way, spawning equivalent low threads
  - Defined as a bisimilarity (Mantel and Sudbrook 2010)
  - Verifiable compositionally using type systems

Overview of our Solution

- Study the scheduler behavior in isolation from the concrete operational semantics of the threads
- Identify for the scheduler a form of system security in the style of Goguen and Meseguer

Theorem

secure scheduler

+

possibilistically secure program

$\Rightarrow$

probabilistically secure program

Probabilistically Secure Program

- Assume transitions have probabilities
- Program assigns to each state a Markov chain of configurations having states as components

Probabilistic security of the program: If starting in alternative states with the same low part, the executions modify the low part of the state in the same ways with the same probabilities

  - Defined as an adaptation of weak probabilistic bisimilarity (Smith 2003)

Overview of our Solution

- Study the scheduler behavior in isolation from the concrete operational semantics of the threads
- Identify for the scheduler a form of system security in the style of Goguen and Meseguer

Theorem

secure scheduler

+

possibilistically secure program

$\Rightarrow$

probabilistically secure program

Scheduler Behavior

- A history is a list or threads “taken so far” together with the sets of available threads at each moment.
- A scheduler is an assignment of a probability distribution to the set of current threads of each history

$$sched : \Pi_{H \in History} Distrib\ (Currently_Available\ (H))$$

- E.g., the uniform scheduler:
$sched_H = 1 / |Currently_Available\ H|$

System Security

- System with multiple users who can act and observe
- Goguen and Meseguer, 1982:
  - Given a group B of users, the system is B-secure if, for any $a \notin B$, what B users do does not affect what a can observe
  - a would make the same observations if the users in B were completely removed from the system
- Users = Threads
- Actions = Single Execution Steps
- a-Observations = …

  "Exit probabilities" of taking B-steps followed by a a-step

Secure Scheduler

- A scenario is a fair tree of histories: it branches on all available threads
- A scheduler sch induces a probability space on each scenario
- sch is called secure if, for all scenarios Sc, histories $H \in Sc$,

$B \subseteq$ Currently_Available(H), and $a \in$ Currently_Available(H) - B

$$Prob_{sch,Sc,H}((\text{Takes } B) \text{ Until } (\text{Takes } a)) = sch_{Remove(B,Sc),H}(a)$$

where Remove(B,Sc) is the scenario obtained from Sc by removing any trace of B

Secure Scheduler

- A scenario is a fair tree of histories: it branches on all available threads
- A scheduler sch induces a probability space on each scenario
- sch is called secure if, for all scenarios Sc, histories $H \in Sc$,

$B \subseteq$ Currently_Available(H), and $a \in$ Currently_Available(H) - B

$Prob_{sch,Sc,H}$((Takes B) Until (Takes a)) = Removal of B

where Remove(B,Sc) is the scenario obtained from Sc by removing any trace of B

Secure Scheduler

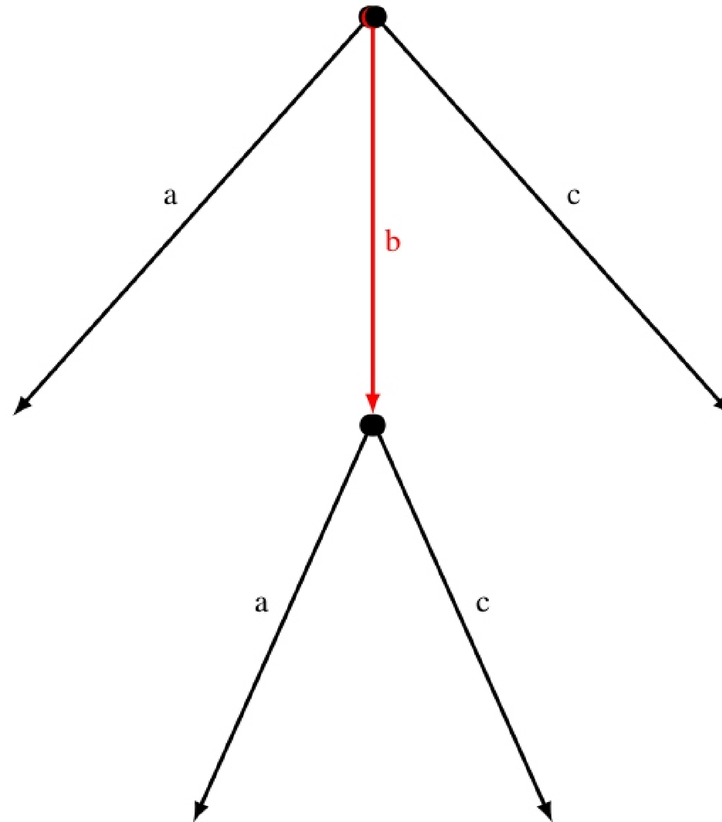
- A scenario is a fair tree of histories: it branches on all available threads
- A scheduler sch induces a probability space on each scenario
- sch is called secure if, for all scenarios Sc, histories $H \in Sc$,

$B \subseteq$ Currently_Available(H), and $a \in$ Currently_Available(H) - B

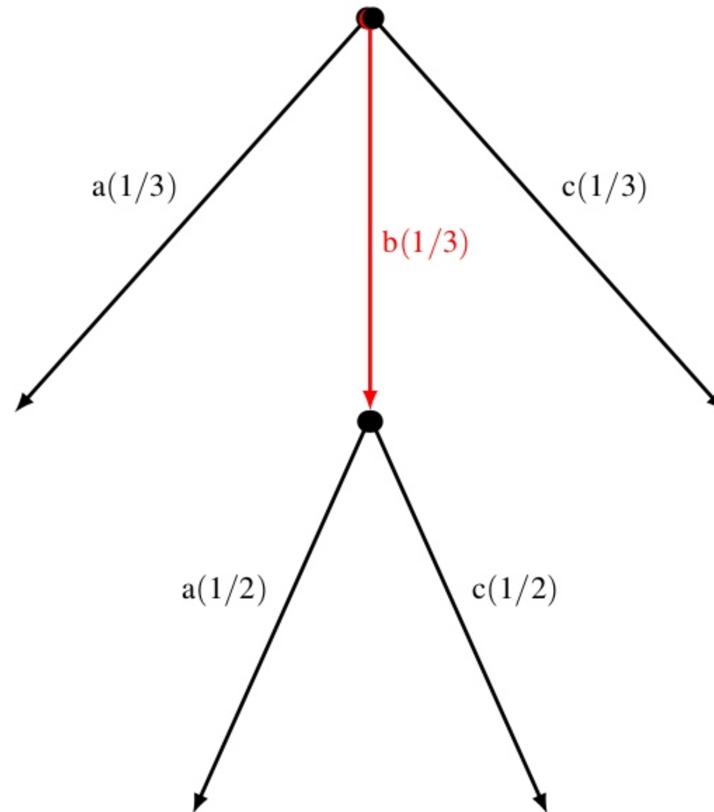
Hiding of B = Removal of B

where Remove(B,Sc) is the scenario obtained from Sc by removing any trace of B

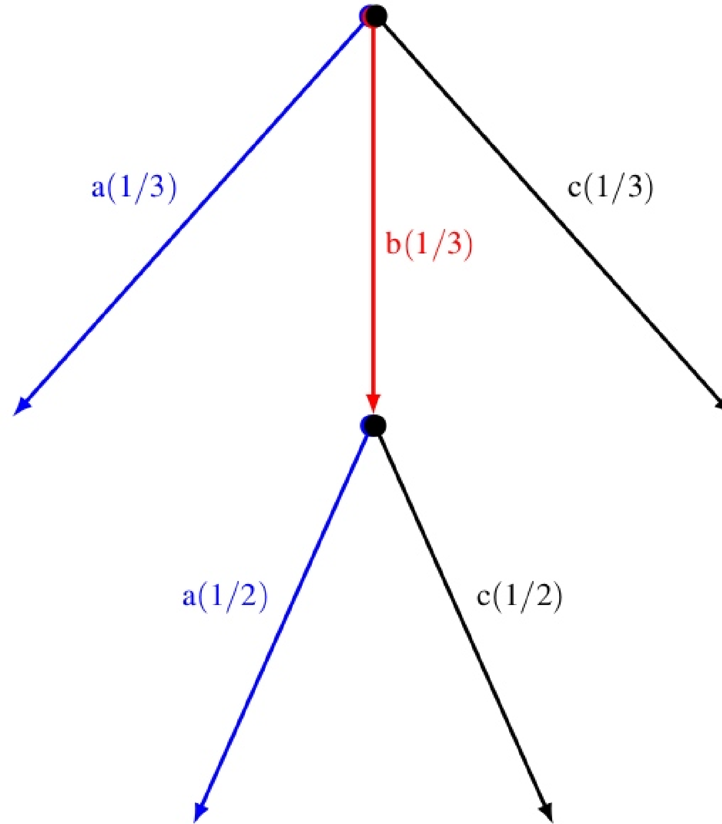
Nondeterministic Scenario



Scenario with Uniform Scheduler

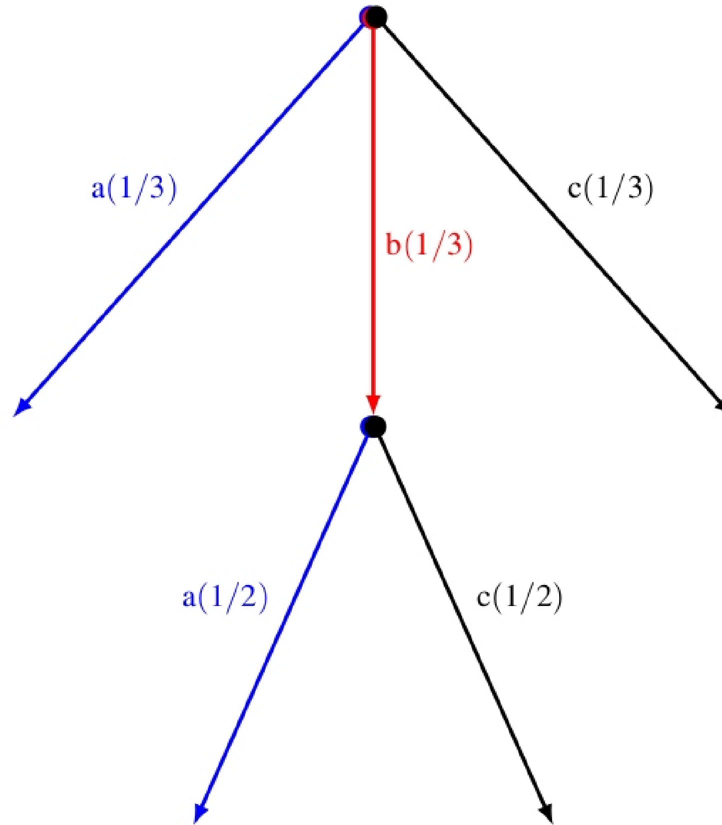


Hiding b

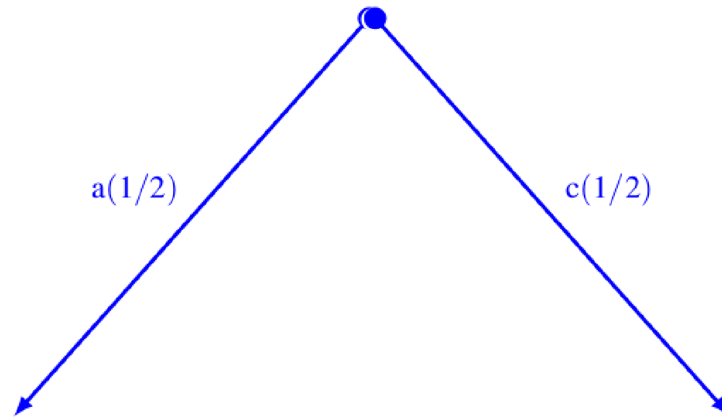


Hiding b

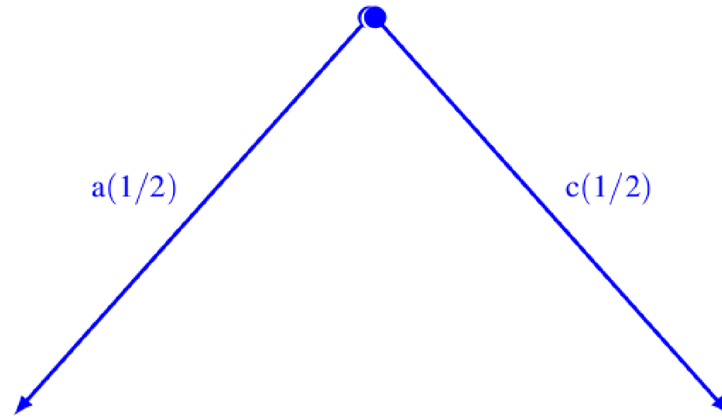
1/3 + 1/3 * 1/2 = 1/2



Hiding b



Hiding = Removal



Back to the Examples

Thr1: while h > 0 do {h := h − 1} ; l := 2 Thr2: l := 1

Thr1: h := 1 ; l := 2 Thr2: l := 1;

Thr1: while h > 0 do {h := h − 1} Thr2: l := 2 Thr3: l := 1

Back to the Examples

Thr1: while h > 0 do {h := h − 1} ; l := 2 Thr2: l := 1 ✕

Thr1: h := 1 ; l := 2 Thr2: l := 1;

Thr1: while h > 0 do {h := h − 1} Thr2: l := 2 Thr3: l := 1

Back to the Examples

Thr1: while h > 0 do {h := h − 1} ; l := 2 Thr2: l := 1 ✕

Thr1: h := 1 ; l := 2 Thr2: l := 1; ✔

Thr1: while h > 0 do {h := h − 1} Thr2: l := 2 Thr3: l := 1

Back to the Examples

Thr1: while h > 0 do {h := h − 1} ; l := 2 Thr2: l := 1 ✕

Thr1: h := 1 ; l := 2 Thr2: l := 1; ✔✔

Thr1: while h > 0 do {h := h − 1} Thr2: l := 2 Thr3: l := 1

Back to the Examples

Thr1: while h > 0 do {h := h − 1} ; l := 2 Thr2: l := 1 ✕

Thr1: h := 1 ; l := 2 Thr2: l := 1; ✔✔

Thr1: while h > 0 do {h := h − 1} Thr2: l := 2 Thr3: l := 1 ✔

Conclusion

- Class of schedulers that
  cut down nondeterminism in a secure way
- Characterized by the equation

  Hiding = Removal

- Includes interesting schedulers such as
  uniform and round robin
- Insight from system security into
  language-based security

Open Questions

- Is there an operational semantics for probabilistic schedulers allowing for a syntactic security criterion?
- From a review of this paper: “So my clear impression is that this is very fine work. That said, I cannot figure out why they have submitted the paper to this conference.”

  “My best guess is that it is because small-step operational semantics is fundamental to them, and that has been a focus of work on coalgebra in the past.”

  We study the relationship between nondeterministic and probabilistic notions of bisimilarity

Codatatypes Meet Isabelle

codatatype thread =

Step (state → state × thread option × thread list)

codatatype sched =

Sched ((threadID × real × sched) finite_set)

coinductive well_formed_sched : sched → bool

where ...

Codatatypes Meet Isabelle

codatatype thread =

Step (state $\rightarrow$ state $\times$ thread option $\times$ thread list)

compositional nesting of (co)datatypes

codatatype sched =

Sched ((threadID $\times$ real $\times$ sched) finite_set)

coinductive well_formed_sched : sched $\rightarrow$ bool

where ...

Codatatypes Meet Isabelle

codatatype thread =

Step (state → state × thread option × thread list)

codatatype sched =

Sched ((threadID × real × sched) finite_set)

(co)recursion through permutative types

coinductive well_formed_sched : sched → bool

where ...

Isabelle Embraces Codatatypes

(joint work with Jasmin Blanchette and Dmitriy Traytel)

- Based on a class of functors accessible functors on Set
- Good support for primitive corecursion
- Beta version of incremental up-to corecursion and mixed recursive/corecursive definitions
- What can we improve w.r.t. expressiveness, convenience, conciseness, etc.?

  Lutz Schröder and Stefan Milius helped us eliminate one cardinality invariant

Noninterfering Schedulers

When Possibilistic Noninterference Implies Probabilistic Noninterference

Andrei Popescu, Johannes Hölzl, Tobias Nipkow

Technische Universität München

Solutions from the Literature

- Sabelfeld and Sands 1999: strong security
  - Threads run independently on private data
  - Security w.r.t. all schedulers
- Zdancewic and Myers 2003: observational determinism
  - Data race freedom
  - Security w.r.t. all schedulers
- Russo and Sabelfeld 2006:
  - Nonstandard synchronization primitives
  - Security w.r.t. customized schedulers
- Mantel and Sudbrock 2010: robust schedulers
  - Thread runs allowed to depend on time, but assumed terminating
  - Security w.r.t. a large class of schedulers
- Boudol and Castellani

  Operational semantics for scheduler
  - No probabilities, no dynamic thread creation