FOSAD Summer School 2015

Concurrent Noninterference

Day 2: Noninterference for multi-threaded programs

Heiko Mantel, Computer Science Department, TU Darmstadt

collaborators on this topic
Aslan Askarov, Timo Bähr, Steve Chong, Steffen Lortz, Alexander Lux, Matthias Perner, Andrei Sabelfeld, David Sands, Jens Sauer, David Schneider, Artem Starostin, Henning Sudbrock, Alexandra Weber, …

TECHNISCHE UNIVERSITÄT DARMSTADT

MAIS Modeling and Analysis of Information Systems

RS³

From Sequential to Concurrent Computation

Information-flow security for sequential programs

- ❑ formulation of noninterference-like properties
- ❑ very many analysis techniques and tools
  - ❑ many with soundness proofs for some noninterference-like property
- ❑ tradeoff between precision and efficiency is understood some extent

⇒ Theoretical foundations are sufficiently well developed for applications.

Is information-flow security for concurrent programs more complex?

- ❑ If yes, how much more?
- ❑ Are there any substantial additional difficulties?
  - ❑ How can these additional difficulties be approached?
  - ❑ How mature are the current solutions?

You will be able to answer these questions after this part of the tutorial.

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- ❑ multi-threaded computations
- ❑ information leakage by multi-threaded programs
- ❑ challenges for information-flow security for multi-threaded programs
- ❑ noninterference for multi-threaded programs
- ❑ formalizing noninterference: two traditions with pros and cons
- ❑ lifting local security guarantees to global security guarantees

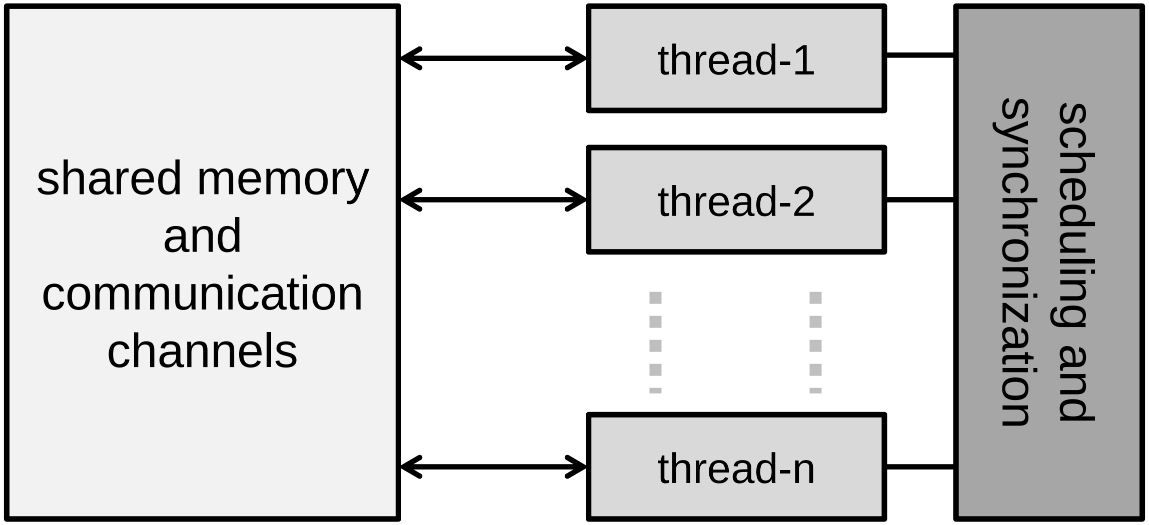
Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

What is a multi-threaded program?

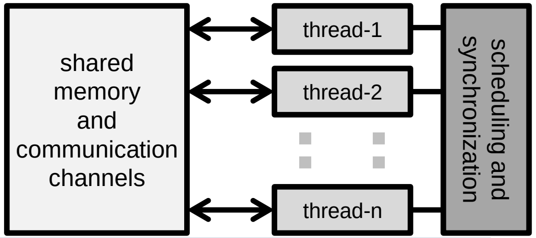
Execution of a multi-threaded program


shared memory and communication channels
thread-1
thread-2
thread-n
scheduling and synchronization

Multiple threads run concurrently:

- Each thread executes a separate program.
- Each thread has read and write access to a shared memory.
- Which thread performs the next step is determined by a scheduler.
- Synchronization between threads can be used for coordination.

Global Configurations


shared memory and communication channels
thread-1
thread-2
thread-n
scheduling and synchronization

We denote a parallel program with n threads by .

$$prog_1 || \cdots || prog_n$$

Global configuration $\langle\langle prog_1 || \cdots || prog_n , mem \rangle\rangle$

- where $prog_i \in Prog$ is a program for each $i \in \{1, \ldots, n\}$
- where $mem\colon Var \to Val$ is a memory states (like in Part 1)

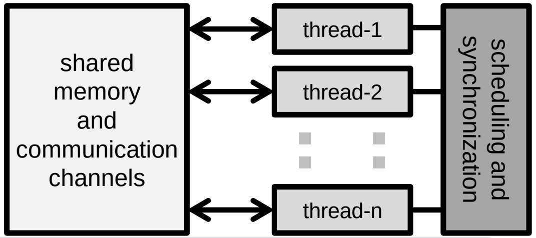
Intuition: $\langle\langle prog_1 || \cdots || prog_n , mem \rangle\rangle$ models a snapshot,

- where $prog_i$ models the program that remains to be executed by the ith thread for each $i \in \{1, \ldots, n\}$ and
- where mem models the current values of all shared variables.

Example $\langle\langle x \coloneqq 42 \;||\; y \coloneqq x * y , [x \mapsto 0, y \mapsto 0] \rangle\rangle$

- Two concurrent assignments remain to be executed.
- Both variables currently have value 0.

Local Configurations


shared memory and communication channels
thread-1
thread-2
thread-n
scheduling and synchronization

Local configuration $\langle prog, mem\rangle \in Prog \times Mem$

- ❑ where $prog \in Prog$ is a program
- ❑ where $Mem = Var \rightarrow Val$ is the set of all memory states (like in Part 1)

Intuition: $\langle prog, mem\rangle$ models a snapshot of a thread's view in a run,

- ❑ where $prog$ models the program to be executed by the thread
- ❑ where mem models the current values of all shared variables.

Notation: Given a global configuration

$$gcnf = \langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle$$

we write

- ❑ $\#gcnf$ for the number of threads, i.e. $\#gcnf = n$
- ❑ $gcnf(i)$ for the local configuration of the ith thread ($i \in \{1, \ldots, \#gcnf\}$).

That is, $\#gcnf = n$ and $gcnf(7) = \langle prog_7, mem\rangle$.

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- ❑ multi-threaded computations
- ❑ information leakage by multi-threaded programs
- ❑ challenges for information-flow security for multi-threaded programs
- ❑ noninterference for multi-threaded programs
- ❑ formalizing noninterference: two traditions with pros and cons
- ❑ lifting local security guarantees to global security guarantees

Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

Formalizing Computation Steps

Labeled transitions capturing steps of a multi-threaded program

$$\langle\langle prog_1 || \cdots || prog_n , mem\rangle\rangle \overset{\alpha}{\rightarrow} \langle\langle prog_1{}' || \cdots || prog_n{}' , mem'\rangle\rangle$$

or

$$gcnf \overset{\alpha}{\rightarrow} gcnf'$$

where $gcnf = \langle\langle prog_1 || \cdots || prog_n , mem\rangle\rangle$

and $gcnf' = \langle\langle prog_1{}' || \cdots || prog_n{}' , mem'\rangle\rangle$ are global configurations.

How does a multi-threaded computation progress?

- The scheduler selects a thread that can perform a computation step and this thread performs a computation step.

OR

- The scheduler selects multiple threads that can synchronize. and these threads synchronize.

UNTIL all threads have terminated.

Scheduling

A derivation rule capturing nondeterministic scheduling

$$\frac{\begin{array}{ll} i \in \{1, \dots, n\} & \langle prog_i, mem\rangle \xrightarrow{\alpha} \langle prog_i', mem'\rangle \\ \alpha \in Ev & \forall j \in \{1, \dots, n\} \backslash \{i\}.\, prog_j{}' = prog_j \end{array}}{\langle\langle prog_1 || \cdots || prog_n\,, mem\rangle\rangle \xrightarrow{\alpha} \langle\langle prog_1{}' || \cdots || prog_n{}'\,, mem'\rangle\rangle}$$

The scheduler chooses an arbitrary thread that can perform a computation step.

this thread performs a computation step

Other scheduling strategies are round-robin and uniform.

no other thread performs a step

More complex scenarios result from allowing computation steps by multiple threads at a time, interrupts during a computation step, …

Leakage via Scheduling (1)

Leakage via scheduling

attacker learns whether initial value of secret was positive

```
output 0 to untrusted-channel || output 1 to untrusted-channel
```

A secret is leaked to an untrusted channel assuming

- variable **secret** contains secret information when the statement is run
- attacker can observe messages sent to **untrusted-channel**
- scheduler firstly selects the first thread for execution given that the value of **secret** is positive and, otherwise, selects the second thread

How does the attacker deduce secret information?

- If the attacker observes the sequence [0,1] on **untrusted-channel** then he knows that the initial value of **secret** must have been positive.

The scheduling choice should not depend on secrets!

Leakage via Scheduling (2)

Leakage via scheduling

```
while  secret > 0  do                          ||   skip;
   secret := secret – 1  od;                   ||   skip;
output  1  to  untrusted-channel               ||   output  0  to  untrusted-channel
```

For a round-robin scheduler that re-schedules after each step:

- The sequence [1,0] would appear on **untrusted-channel** if the initial value of **secret** is at most 1. Otherwise, [0,1] would appear.

For a uniform scheduler that chooses threads with equal probability:

- The sequence [0,1] would appear more likely on **untrusted-channel** than the sequence [1,0] for high initial values of **secret .**

This is a so called **internal timing channel** – this is tricky!

How to take scheduling into account? (1)

If the scheduler's behavior is known then

one can take it into account when verifying information-flow security.

Problems

The scheduling algorithm is usually not even known to the programmer.

- ❑ Semantics of concurrency features in programming languages are often underspecified to create freedom for compiler development.

Even if the scheduling algorithm is known, a scheduler-specific security analysis is tedious. It needs to be redone for each scheduler.

Is this problem specific to software security?

No, it appears when analyzing concurrent programs wrt. any property.

This doesn't look bad. Can we apply a standard solution?

How to take scheduling into account? (2)

What if the scheduling algorithm is not known at analysis time?

The usual solution

Analyze the program under a possibilistic scheduler, i.e., the scheduler that nondeterministically chooses an arbitrary thread (as on slide 9).

Underlying reasoning for why this is suitable (beware!)

The possibilistic scheduler over-approximates scheduling behavior:

- ❑ E.g., every scheduling choice that a round-robin scheduler might make can also be made by the possibilistic scheduler.
- ❑ In contrast, the possibilistic scheduler can make scheduling choices that the round-robin scheduler cannot make.

If one analyses a program's behavior under a possibilistic scheduler then one takes all behaviors possible under other schedulers into account.

Is more (i.e. over-approximation) always better?

How to take scheduling into account? (3)

Over-approximating scheduling behavior by possibilistic scheduling

- When does it work and when not?

If the property can be characterized by a predicate of system runs

recall from Part 1 of this tutorial:

- There is a predicate P on individual runs, i.e. $P(\tau)$ holds or does not hold for a given system run τ .
- A system satisfies the property specified by P if and only if $P(\tau)$ holds for each run τ that is possible for this system.

If such a property is verified assuming possibilistic scheduling

Then the property also holds for more concrete scheduling behaviors.

It works for a large class of relevant properties!

How to take scheduling into account? (4)

Over-approximating scheduling behavior by possibilistic scheduling

- ❑ When does it work and when not?

A more general characterization of when over-approximation is good

The property of interest quantifies over possible system runs using only universal quantifiers.

- ❑ This holds if property is characterized by predicate on runs: $\forall\tau.\, P(\tau)$.
- ❑ This also holds for the definition of noninterference in Part 1 of tutorial.

When does over-approximation not work?

If the property of interest existentially quantifies over possible runs.

- ❑ $\forall\exists$ is a quantifier structure that appears in definitions of noninterference (will be explained later in this tutorial).

There are other solutions than possibilistic scheduling!

Shared Memory

Modifications of the shared memory (same rule as before)

$$\frac{\begin{array}{ll} i \in \{1, \dots, n\} & \langle prog_i, mem\rangle \xrightarrow{\alpha} \langle prog_i', mem'\rangle \\ \alpha \in Ev & \forall j \in \{1, \dots, n\} \backslash \{i\}.\, prog_j{}' = prog_j \end{array}}{\langle\langle prog_1 || \cdots || prog_n\, , mem\rangle\rangle \xrightarrow{\alpha} \langle\langle prog_1{}' || \cdots || prog_n{}'\, , mem'\rangle\rangle}$$

the modification of shared variables are propagated from premise to conclusion

Complete sharing

Every variable can, in principle, be read and written

- ❑ by every thread
- ❑ at all times.

It is the programmer’s obligation to reduce this freedom, if necessary.

More complex scenarios possible, e.g., threads with local memory, non-atomic memory updates, and relaxed consistency guarantees.

Leakage via Shared Memory

None of the following programs, alone causes information leakage:

```
x:=secret;  x:=0
```

```
x:=0;  output  x  to  untrusted-channel
```

Leakage by fine-grained resource sharing (here: program variables)

```
x:=secret;  x:=0   ||   output  x  to  untrusted-channel
```

attacker might learn initial value of secret

How does the attacker deduce secret information?

- ❑ If the attacker observes any value other than 0 on **untrusted-channel** then he knows that this was the initial value of **secret** .

This is tricky! I will get back to this.

Synchronization (1)

A barrier command: $\mathrm{barrier}$

Intuition: Passing a barrier has no effect other than passing the barrier.

However, certain conditions must be fulfilled in order to pass (next slide).

Derivation rule for the barrier command

$$\frac{}{\langle barrier, mem\rangle \xrightarrow{sync} \langle \mathrm{stop}, mem\rangle}$$

event $sync$ is emitted

Programming language after adding the barrier command

- $$\begin{aligned} prog \coloneqq\ & \mathrm{stop} \mid \mathrm{skip} \mid x \coloneqq aexp \mid \mathrm{barrier} \\ & \mid \mathrm{input}\ x\ \mathrm{from}\ ch \mid \mathrm{output}\ aexp\ \mathrm{to}\ ch \\ & \mid prog; prog \mid \mathrm{if}\ bexp\ \mathrm{then}\ prog\ \mathrm{else}\ prog\ \mathrm{fi} \\ & \mid \mathrm{while}\ bexp\ \mathrm{do}\ prog\ \mathrm{od} \end{aligned}$$

Synchronization (2)

Barrier synchronization

$$\frac{\exists\, i \in \{1,\dots,n\}.\ prog_i \neq \mathrm{stop} \qquad \forall\, i \in \{1,\dots,n\}.\ (prog_i = \mathrm{stop} \wedge {prog_i}' = prog_i) \vee \langle prog_i, mem\rangle \xrightarrow{sync} \langle prog_i', mem\rangle}{\langle\langle prog_1 || \cdots || prog_n\, , mem\rangle\rangle \dot{\rightarrow} \langle\langle {prog_1}' || \cdots || {prog_n}'\, , mem\rangle\rangle}$$

at least one thread has not yet terminated

each thread that has not yet terminated performs a step that emits *sync*

There are variants of this barrier command, e.g., only a dedicated subset of threads needs to participate or only a certain number threads needs to participate in passing the barrier jointly.

There are many further synchronization primitives, e.g., locks, …

Leakage via Synchronization (1)

Example

```
if  secret > 0  then barrier          ||  barrier;
                 else skip fi;        ||  output  1  to  untrusted-channel
output  0  to  untrusted-channel      ||
```

What are the possible observations of an attacker during a run?

1. no output on **untrusted-channel** yet
2. [0] has been output on **untrusted-channel** so far
3. [1] has been output on **untrusted-channel** so far
4. [0,1] has been output on **untrusted-channel** so far
5. [1,0] has been output on **untrusted-channel** so far

What can the attacker deduce?

In the 3rd and 5th case above, the attacker learns that the initial value of **secret** must have been positive because that 1 is output first, is only possible if both threads jointly pass the barrier.

Leakage via Synchronization (2)

Example

```
if  secret > 0  then barrier           ||  if  secret > 0  then skip
                 else skip fi;         ||                   else barrier fi;
output  0  to  untrusted-channel       ||  output  1  to  untrusted-channel
```

What are the possible observations of an attacker during a run?

[], [0], [1], [0,1], and [1,0] could be observed on **untrusted-channel**

What can the attacker deduce?

From [1] and [1,0], the attacker learns that the initial value of **secret** must have been positive. If [0] or [0,1] occurs **secret** > 0 was initially false.

Whether a barrier is reached should not depend on secrets!

Synchronization statements need similar care like public outputs!

Runs of Multi-threaded Programs

Labeled transitions capturing runs

$$\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \overset{\tau}{\Rightarrow} \langle\langle prog_1' || \cdots || prog_n', mem'\rangle\rangle$$

Rules

$$\frac{}{\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \overset{[]}{\Rightarrow} \langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle}$$

$$\frac{\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \overset{\alpha}{\rightarrow} \langle\langle prog_1' || \cdots || prog_n', mem'\rangle\rangle \qquad \langle\langle prog_1' || \cdots || prog_n', mem'\rangle\rangle \xRightarrow{[\alpha 1,\ldots,\alpha n]} \langle\langle prog_1'' || \cdots || prog_n'', mem''\rangle\rangle}{\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \xRightarrow{[\alpha,\alpha 1,\ldots,\alpha n]} \langle\langle prog_1'' || \cdots || prog_n'', mem''\rangle\rangle}$$

This lifting of steps to runs is similar as for sequential programs.

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- ❑ multi-threaded computations
- ❑ information leakage by multi-threaded programs
- ❑ challenges for information-flow security for multi-threaded programs
- ❑ noninterference for multi-threaded programs
- ❑ formalizing noninterference: two traditions with pros and cons
- ❑ lifting local security guarantees to global security guarantees

Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

Summary of Observations so far

Information-flow security for multi-threaded programs is tricky

In comparison to sequential programs, further leaks are possible:

- ❑ leakage via scheduling,
- ❑ leakage via fine-grained resource sharing, and
- ❑ leakage via synchronization.

Solutions should fulfill further conditions than avoiding such leaks:

- ❑ Platform-independent semantics of languages should be supported.
- ❑ Precision should be high enough to not
  - ❑ reject too many programs as potentially insecure (for analyses)
  - ❑ interfere with too many benign program behaviors (for enforcement)

Obtaining such solutions is subject to current and future research!

Some Exercises (Homework 3)

1. Create all possible derivations of

   $$\langle\langle \mathrm{x} := \textbf{secret};\ \ \mathrm{x}{:=}0 \ \|\ \mathrm{output}\ \ \mathrm{x}\ \ \mathrm{to}\ \ \textbf{untrusted−channel}\ , [\mathrm{x} \mapsto 0]\ \rangle\rangle$$
   $$\overset{\tau}{\Rightarrow} \langle\langle\ \mathrm{stop}\ \|\ \mathrm{stop}, mem' \rangle\rangle$$

   - For which values of τ and mem' exists a derivation?
2. Develop a formal definition of noninterference for multi-threaded programs that faithfully captures our informal definition
   - if the initial values of all variables in **high** $\subseteq$ Var are the secrets
   - for attackers that can only observe the initial and final values of variables in the set **low** = Var\high .
3. Add the command $\mathrm{spawn}\, prog$ to the syntax of our programming language. This command terminates while creating a new thread that shall execute the program $prog$. Formalize the semantics of this command by adding rules.
4. Augment the language by further synchronization commands.

End of presentation on September 1

break & time for homework

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- multi-threaded computations
- information leakage by multi-threaded programs
- challenges for information-flow security for multi-threaded programs
- noninterference for multi-threaded programs
- formalizing noninterference: two traditions with pros and cons
- lifting local security guarantees to global security guarantees

Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

Formalizing Noninterference (1)

Noninterference informally (like in Part 1 of the tutorial)

A program is noninterferent if the observations that an attacker makes during runs of this program do not depend on secrets in any way.

A formal definition of noninterference (like in Part 1)

A program $prog$ is noninterferent if and only if

$$\forall mem1, mem2, mem1', mem2' : Var \to Val \,.\, \forall \tau, \tau' \in Ev^* .$$
$$[\, \langle prog, mem1 \rangle \overset{\tau}{\Rightarrow} \langle stop, mem2 \rangle \wedge mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\, \langle prog, mem1' \rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2' \rangle \;\Rightarrow\; mem2' =_{\mathbf{low}} mem2 \,]$$

Let us now modify this definition of noninterference such that we obtain a definition of noninterference for an attacker who can messages output on some channels.

Formalizing Noninterference (2)

Choice: What is secret? (like in Part 1)

There is a dedicated set of variables **high** $\subseteq$ Var . The initial values of these variables must be kept confidential.

Choice: What can the attacker observe? (difference to Part 1)

There is a dedicated set of variables **low** $=$ Var\high . The initial and final values of these variables are what the attacker observes.

In addition, there is a dedicated set of channels **L** $\subseteq$ Ch. The attacker can observe all messages output on these channels.

To proceed, we need a definitions of indistinguishability for events and for event sequences.

Formalizing Noninterference (3)

Indistinguishability of memories for the attacker (like in Part 1)

Two memories $mem, mem' \colon Var \to Val$ are indistinguishable (denoted by $mem =_{\mathbf{low}} mem$’) if and only if

$$\forall x \in \mathbf{low}.\, mem(x) = mem'(x)$$

Indistinguishability of events for the attacker (new)

Two events $\alpha, \alpha' \in Ev$ are indistinguishable (denoted by $\alpha =_{\mathbf{L}} \alpha'$) iff

$$\alpha, \alpha' \in \{\, \cdot, input(ch, n), output(ch', n) \mid n \in Val, ch \in Ch, ch' \in Ch \setminus \mathbf{L}\}$$
$$\vee\, \exists ch_l \in \mathbf{L}.\, \exists n \in Val.\, (\alpha = output(ch_l, n) \wedge \alpha' = output(ch_l, n))$$

Two output events on an observable channel are indistinguishable only if they agree on both, the channel and the value.

We next lift this to indistinguishability on sequence of events.

Formalizing Noninterference (4)

Traces

A **trace** τ is a list of events, i.e. $\tau \in Ev^*$.

Projecting a trace $\tau \in Ev^*$ to the set L (denoted $\tau \downarrow \mathsf{L}$)

- ❑ $[] \downarrow \mathsf{L} = []$
- ❑ $([\alpha].\tau) \downarrow \mathsf{L} = \tau \downarrow \mathsf{L}$

  if $\alpha \in \{\, \cdot, input(ch,n), output(ch',n) \mid n \in Val, ch \in Ch, ch' \in Ch \setminus \mathsf{L} \,\}$
- ❑ $([output(ch,n)].\tau) \downarrow \mathsf{L} = [output(ch,n)].(\tau \downarrow \mathsf{L})$ if $ch \in \mathsf{L}$

Indistinguishability of traces

Two traces $\tau, \tau' \in Ev$ are indistinguishable (denoted by $\tau =_{\mathsf{L}} \tau'$) iff

$$(\tau \downarrow \mathsf{L}) = (\tau' \downarrow \mathsf{L})$$

Formalizing Noninterference (5)

A formal definition of noninterference (like in Part 1)

A program $prog$ is noninterferent if and only if

$$\forall mem1, mem2, mem1', mem2' : Var \rightarrow Val\,.\, \forall \tau, \tau' \in Ev^*.$$
$$[\,\langle prog, mem1\rangle \overset{\tau}{\Rightarrow} \langle stop, mem2\rangle \wedge mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\,\langle prog, mem1'\rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2'\rangle \;\Rightarrow\; mem2' =_{\mathbf{low}} mem2\,]$$

A definition of noninterference if outputs on **L** are observable

A program $prog$ is noninterferent if and only if

$$\forall mem1, mem2, mem1', mem2' : Var \rightarrow Val\,.\, \forall \tau, \tau' \in Ev^*.$$
$$[\,\langle prog, mem1\rangle \overset{\tau}{\Rightarrow} \langle stop, mem2\rangle \;\wedge\; mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\,\langle prog, mem1'\rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2'\rangle$$
$$\Rightarrow\; [\,mem2' =_{\mathbf{low}} mem2 \;\wedge\; \tau' =_{\mathbf{L}} \tau]$$

Note: You just learnt how to adapt a definition of noninterference!

Formalizing Noninterference (6)

A definition of noninterference if outputs on L are observable

A program $prog$ is noninterferent if and only if

$$\forall mem1, mem2, mem1', mem2' : Var \rightarrow Val \,.\, \forall \tau, \tau' \in Ev^{*}.$$
$$[\, \langle prog, mem1 \rangle \overset{\tau}{\Rightarrow} \langle stop, mem2 \rangle \;\wedge\; mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\, \langle prog, mem1' \rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2' \rangle \Rightarrow \; [\, mem2' =_{\mathbf{low}} mem2 \;\wedge\; \tau' =_{\mathbf{L}} \; \tau]$$

Lifting this definition of noninterference to multi-threaded programs:

A multi-threaded program $prog_1 || \cdots || prog_n$ is noninterferent iff

$$\forall mem1, mem2, mem1', mem2' : Var \rightarrow Val \,.\, \forall \tau, \tau' \in Ev^{*}.$$
$$[\langle\langle prog_1 || \cdots || prog_n, mem1 \rangle\rangle \overset{\tau}{\Rightarrow} \langle\langle \mathrm{stop} || \cdots || \mathrm{stop}, mem2 \rangle\rangle \wedge mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\langle\langle prog_1 || \cdots || prog_n, mem1' \rangle\rangle \overset{\tau'}{\Rightarrow} \langle\langle \mathrm{stop} || \cdots || \mathrm{stop}, mem2' \rangle\rangle$$
$$\Rightarrow \; [\, mem2' =_{\mathbf{low}} mem2 \wedge \tau' =_{\mathbf{L}} \; \tau]$$

Oops!... I did It again! [Britney Spears 2000]

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- ❑ multi-threaded computations
- ❑ information leakage by multi-threaded programs
- ❑ challenges for information-flow security for multi-threaded programs
- ❑ noninterference for multi-threaded programs
- ❑ formalizing noninterference: two traditions with pros and cons
- ❑ lifting local security guarantees to global security guarantees

Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

Another Observation

Lifted definition of noninterference for multi-threaded programs:

A multi-threaded program $prog_1 || \cdots || prog_n$ is noninterferent iff

$$\forall mem1, mem2, mem1', mem2' : Var \to Val\,.\, \forall \tau, \tau' \in Ev^{*}.$$
$$[\langle\langle prog_1 || \cdots || prog_n, mem1 \rangle\rangle \overset{\tau}{\Rightarrow} \langle\langle \mathrm{stop} || \cdots || \mathrm{stop}, mem2 \rangle\rangle \wedge mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\langle\langle prog_1 || \cdots || prog_n, mem1' \rangle\rangle \overset{\tau'}{\Rightarrow} \langle\langle \mathrm{stop} || \cdots || \mathrm{stop}, mem2' \rangle\rangle$$
$$\Rightarrow [\, mem2' =_{\mathbf{low}} mem2 \wedge \tau' =_{\mathbf{L}} \tau]$$

Consider the following program

output 0 to **untrusted-channel** || output 1 to **untrusted-channel**

Intuitively, this program does not leak secrets for a round-robin scheduler.

But this example program violates our definition of noninterference!

Maybe, we did an easy-to-correct mistake in the previous steps???

Formalizing Noninterference - Again

There are two traditions in formalizing noninterference

- The ∀∀-tradition (example instance)

$$\forall mem1, mem2, mem1', mem2' : Var \to Val \,.\, \forall \tau, \tau' \in Ev^*.$$
$$[\,\langle prog, mem1\rangle \overset{\tau}{\Rightarrow} \langle stop, mem2\rangle \land mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\,\langle prog, mem1'\rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2'\rangle \;\Rightarrow\; mem2' =_{\mathbf{low}} mem2\,]$$

- The ∀∃-tradition (example instance)

$$\forall mem1, mem2, mem1' : Var \to Val \,.\, \forall \tau \in Ev^*.$$
$$[\,\langle prog, mem1\rangle \overset{\tau}{\Rightarrow} \langle stop, mem2\rangle \land mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow \exists mem2' : Var \to Val. \exists \tau' \in Ev^*.$$
$$[\,\langle prog, mem1'\rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2'\rangle \;\Rightarrow\; mem2' =_{\mathbf{low}} mem2\,]$$

There is no silver bullet: Both traditions have their disadvantages.

Disadvantages of the ∀∀-Tradition

Such definitions of noninterference are restrictive wrt.

- ❑ programs in which nondeterminisms is intentionally used
  - ❑ e.g., nondeterministic choice of a random number
- ❑ programs in which nondeterminism appears as a side effect
  - ❑ e.g., distributed computations without a strict, global scheduler
  - ❑ e.g., the order in which output by concurrent threads appears

There are also effects on compositional reasoning about programs.

- ❑ Black-box reasoning about the security of programs in terms of security certificates for their program components is difficult because by composing programs, one might introduce nondeterminism.
- ❑ Compositional reasoning will be covered later in this tutorial.

Scheduler-independence in the ∀∀-Tradition

The ∀∀-tradition (example instance, like before)

$$\forall mem1, mem2, mem1', mem2' : Var \to Val\,.\,\forall \tau, \tau' \in Ev^*.$$
$$[\,\langle prog, mem1\rangle \overset{\tau}{\Rightarrow} \langle stop, mem2\rangle \wedge mem1' =_{\mathbf{low}} mem1]$$
$$\Rightarrow [\,\langle prog, mem1'\rangle \overset{\tau'}{\Rightarrow} \langle stop, mem2'\rangle \;\Rightarrow\; mem2' =_{\mathbf{low}} mem2\,]$$

Observation

- If the above property is true, then the attacker's observations are deterministically determined by the initial values of the non-secret variables if the program is executed under a possbilistic scheduler.
- Since the possibilistic scheduler overapproaximates the possibile behaviors of more concrete schedulers, the attacker's observations are also deterministic under more concrete schedulers.

The above properties remain true if the scheduler is refined.

- In the ∀∀-tradition, one gets scheduler independence for free.

Disadvantages of the ∀∃-Tradition

Closure property

A predicate $\phi : \mathcal{P}(\mathcal{A}) \rightarrow \mathrm{Bool}$ is a closure property if for each $A \subseteq \mathcal{A}$, there exists an $A' \subseteq \mathcal{A}$ such that $\phi(A')$ holds.

An obvious fact

If $\phi(A)$ holds then $\phi(B)$ need not hold for $B \subseteq A$.

Noninterference-definitions in ∀∃-tradition are closure properties.

Consequently, if a program that satisfies such a noninterference-definition is constrained by a mechanism making some program runs impossible then the resulting system might not satisfy the noninterference-definition.

- e.g., by refining a possibilistic scheduler by a round-robin scheduler

This problem is known under the name “refinement paradox”.
It was first pointed out in [Jacob’89].

Scheduler-independence in the ∀∃-Tradition

Strong Security [Sabelfeld/Sands'00]

A definition of a noninterference-like security condition that is based on a partial equivalence relation that requires lock-step execution.

$$\begin{tikzcd}
\langle prog1, mem1 \rangle \arrow[r] \arrow[d, dash, "\sim"', "=_{\text{low}}"] & \langle prog2, mem2 \rangle \arrow[d, dash, dotted, "\sim"', "=_{\text{low}}"] \\
\langle prog1', mem1' \rangle \arrow[r, dotted] & \langle prog2', mem2' \rangle
\end{tikzcd}$$

FSI Security [Mantel/Sudbrock'10]

A definition of a noninterference-like security condition that is based on a partial equivalence relation that does not require lock-step execution.

- ❑ allows one to use triangle- in addition to quadrangle-diagrams

Message: A scheduler-independent security analysis is possible, even when following the ∀∃-tradition, but it requires some care.

Roadmap

Part 1: An Introduction to Noninterference

Part 2: Noninterference for Multi-threaded Programs

- multi-threaded computations
- information leakage by multi-threaded programs
- challenges for information-flow security for multi-threaded programs
- noninterference for multi-threaded programs
- formalizing noninterference: two traditions with pros and cons
- lifting local security guarantees to global security guarantees

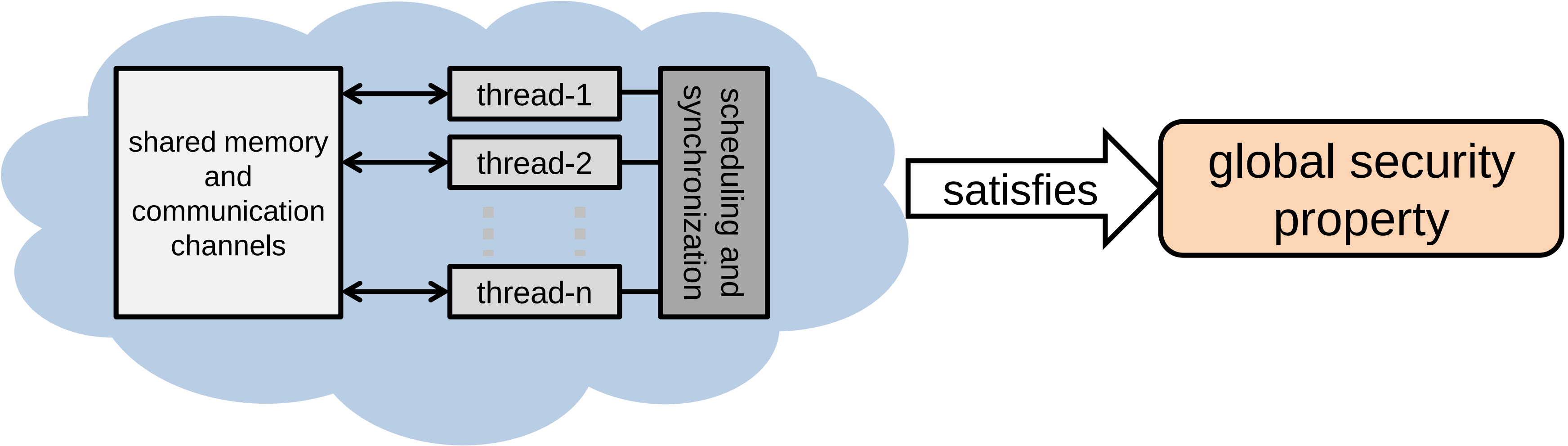
Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

End-to-End Security Guarantees

Security of a program together with its environment (global security)


shared memory and communication channels
thread-1
thread-2
thread-n
scheduling and synchronization
satisfies
global security property

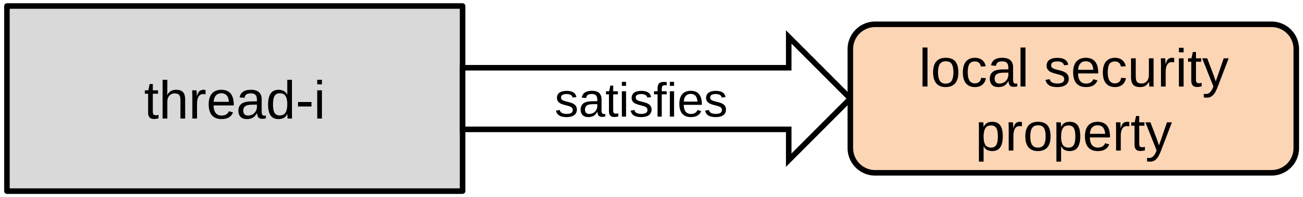
- A security guarantee for the system in form that it is run

End-to-end security is also called global security.

Analyzing global security is difficult for complex systems.

Thread-Local Security Guarantees

Security of an individual thread



- A security guarantee for the program run by an indvidual thread.

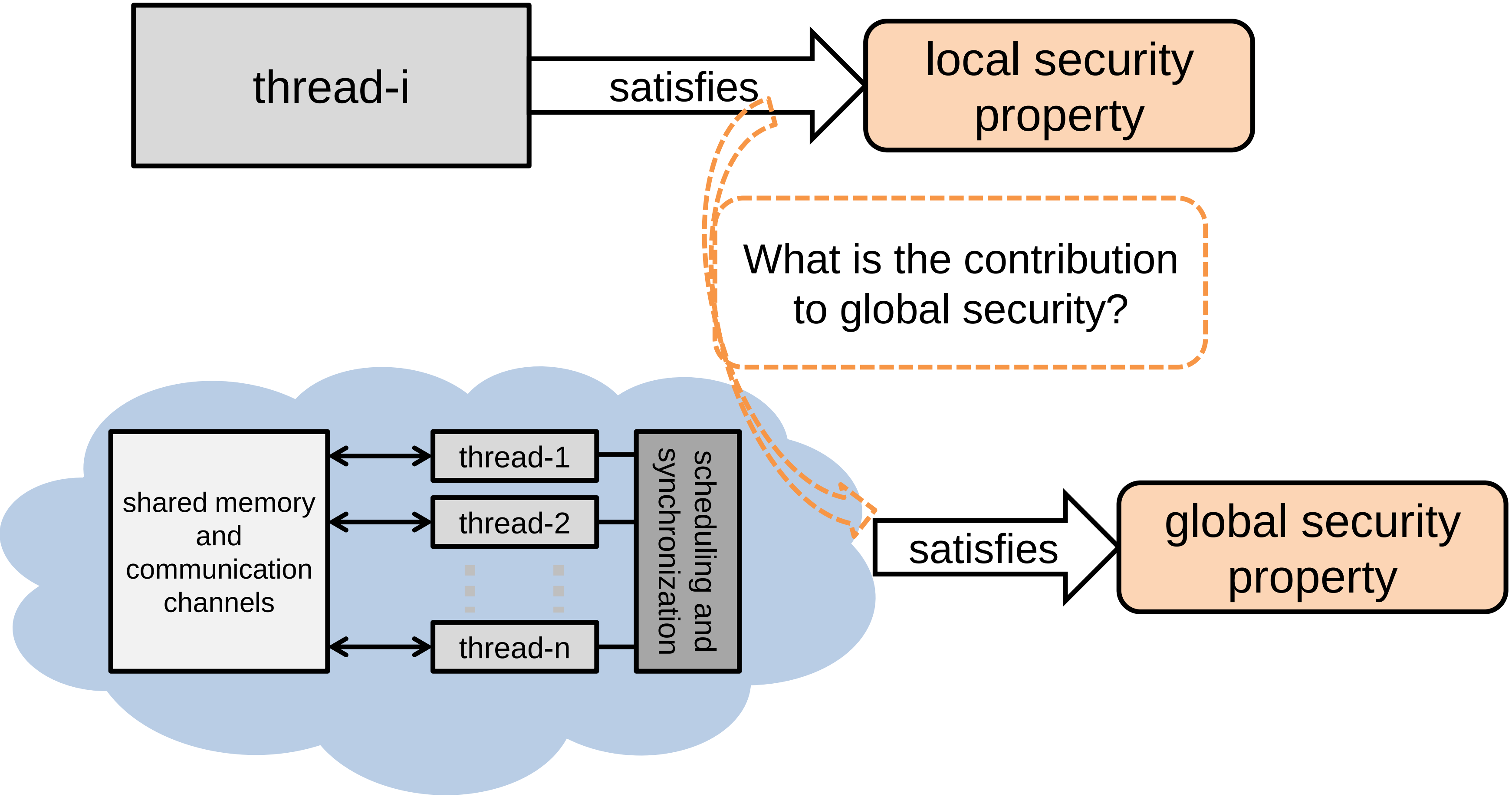
Observations

- Verifying local security for one thread is similar to a verifying security for a sequential program.
- Verifying local security for one thread is conceptually less complex than verifying global security.

How to derive global security from local security?

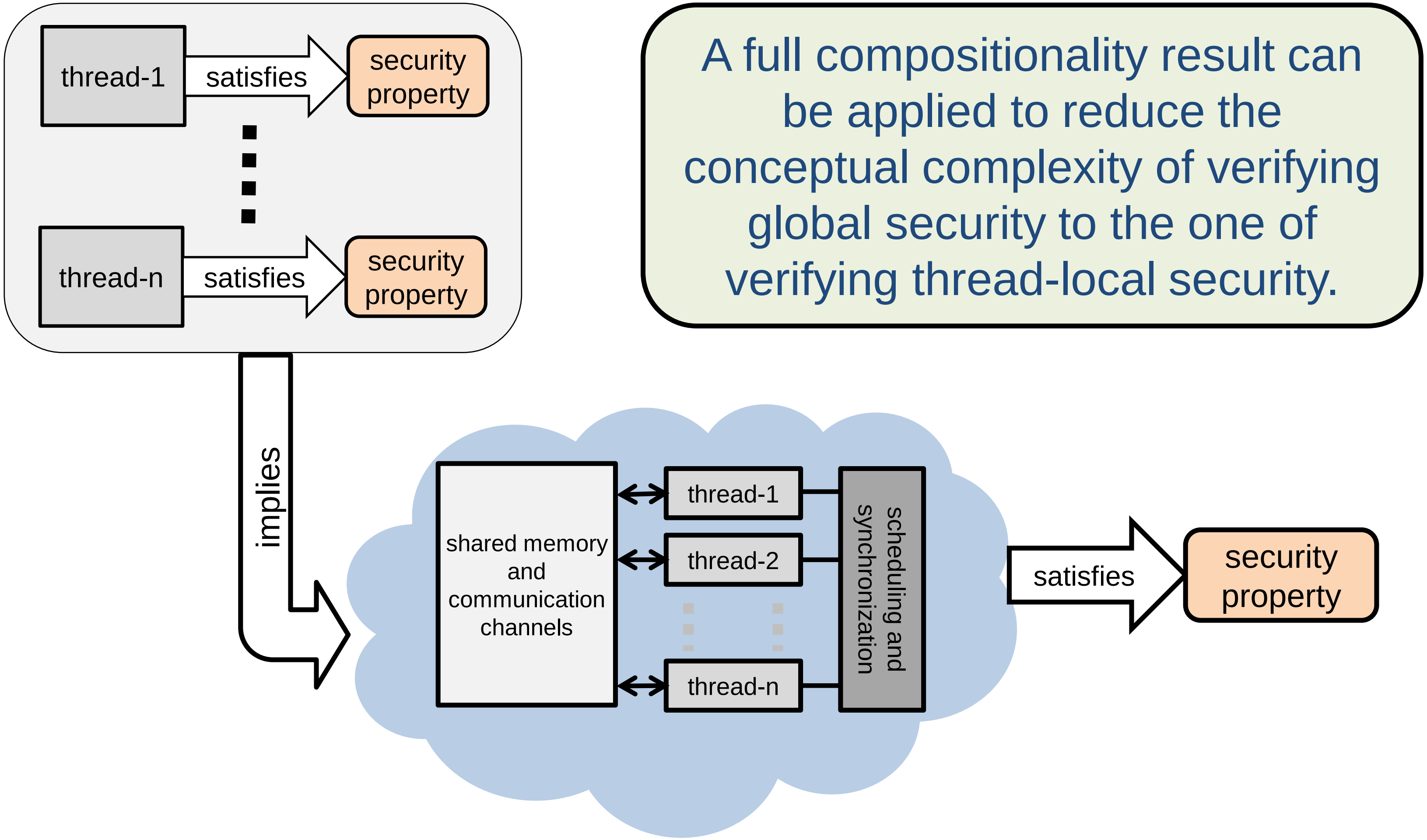
From Local to Global Security (1)

Lifting local security guarantees to global ones


thread-i
satisfies
local security property
What is the contribution to global security?
shared memory and communication channels
thread-1
thread-2
thread-n
scheduling and synchronization
satisfies
global security property

From Local to Global Security (2)

Full compositionality



A full compositionality result can be applied to reduce the conceptual complexity of verifying global security to the one of verifying thread-local security.

Compositionality Results of this Flavor

Strong security [Sabelfeld/Sands 2000]

- ❑ The first proposal of a noninterference-like security property with a scheduler-independence result and a full compositionality result.
- ❑ This security property was shown to be optimal in [Sabelfeld 2003]

Strong security for distributed programs [Mantel/Sabelfeld2003]

- ❑ Strong security is adapted for a programming language that supports message-passing communication between distributed programs.

WHAT_1 [Mantel/Reinhard 2007, Lux/Mantel/Perner 2012]

- ❑ Strong security relaxed to support controlled declassification.

FSI security [Mantel/Sudbrock 2010]

- ❑ Scheduler-independence result for the class of robust schedulers.
- ❑ FSI security is less restrictive than strong security.

Full compositionality results allow one to reduce conceptual complexity of verifying security substantially, but limits precision

Some Exercises (Homework 4)

1. Create a formal definition of noninterference for sequential programs. Assume:
    - Initial values of variables do not need to be kept confidential.
    - There is a dedicated set of channels **H** ⊆ Ch. All values input on these channels need to be kept confidential.
    - The attackers can only observe initial and final values of variables in **low** ⊆ Var .

    Argue why your definition is faithful under these conditions.
2. Generalize your definition from Exercise 4.1 to multi-threaded programs.
3. Create a formal definition of noninterference for sequential programs. Assume:
    - Initial values of variables do not need to be kept confidential.
    - There is a dedicated set of channels **H** ⊆ Ch. All values input on these channels need to be kept confidential. **Moreover, it must be kept confidential whether and how often such inputs have occurred,**
    - The attackers can only observe initial and final values of variables in **low** ⊆ Var .

    Argue why your definition is faithful under these conditions.

    difference to Exercise 4.1
4. Formulate a full compositionality results formally. Use the intuition provided by the pictures on Slide 45 and the noninterference definition from Slide 33.

Bonus Challenge

Choose a scenario from you research background

- ❑ in which confidentiality requirements are relevant
- ❑ which is as simple/small as possible while still making sense for you

Describe the scenario using text, pictures, and/or formalism

- ❑ try to limit your description to ½ a page

Develop a suitable noninterference definition in a stepwise fashion

1. choose one secret and one attacker and describe them in informal terms (at most ½ a page)
2. choose a suitable notion of system configuration and formalize it
3. choose a suitable notion of computation step and formalize it
4. characterize your secret in terms of the concepts under Steps 3 and 4
5. characterize the attackers observations in these terms
6. define noninterference formally (at most ½ a page)

Argue why your definition is faithful for this scenario (at most ½ a page)

submission deadline: **September 3, 2015 15:00**
page limit: **2 pages**; submit to **mantel@cs.tu-darmstadt.de**