FOSAD Summer School 2015

Concurrent Noninterference

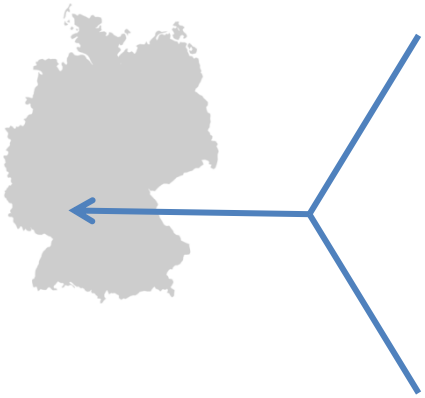
Part 1: An Introduction to Noninterference

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, …

Where am I from?

at TU Darmstadt since 2007



the MAIS group

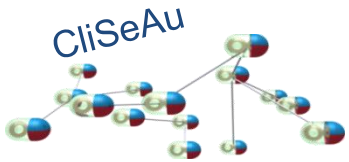
- securing distributed systems
- side channels
- secure software engineering
- security analysis

MAIS

What are my research interests?

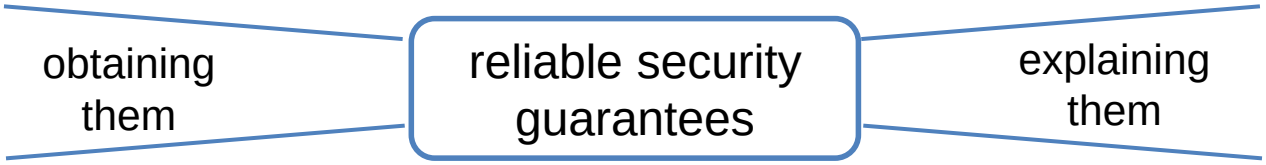
Current research interests

- reliable guarantees for software systems
  - focus: information-flow security and secure usage
  - analysis techniques and tools for deriving security guarantees
  - engineering techniques and tools for establishing security by design
  - languages for expressing security guarantees



SCF:
Side-Channel Finder

- enable security-preserving abstraction, refinement, and composition
  - improving the understanding interplay between security and a, r, and c
  - side channel detection, analysis, and mitigation



My Current Research Projects

CROSSING

CASED

EC SPRIDE
EUROPEAN CENTER FOR
SECURITY AND PRIVACY BY DESIGN

we are hiring

open post-doc positions
- ❑ concurrent program security
- ❑ information-flow security by design

open PhD positions
- ❑ concurrent program security
- ❑ mobile security

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

Facets of Information Security

Security is CIA

- **C : C**onfidentiality
- *the nonoccurrence of unauthorized disclosure of information*
- **I : I**ntegrity
- *the nonoccurrence of unauthorized modification of data or resources*
- **A : A**vailability
- *the degree to which a system or component is operational and accessible when required for use*

Other facets of security can be expressed using CIA

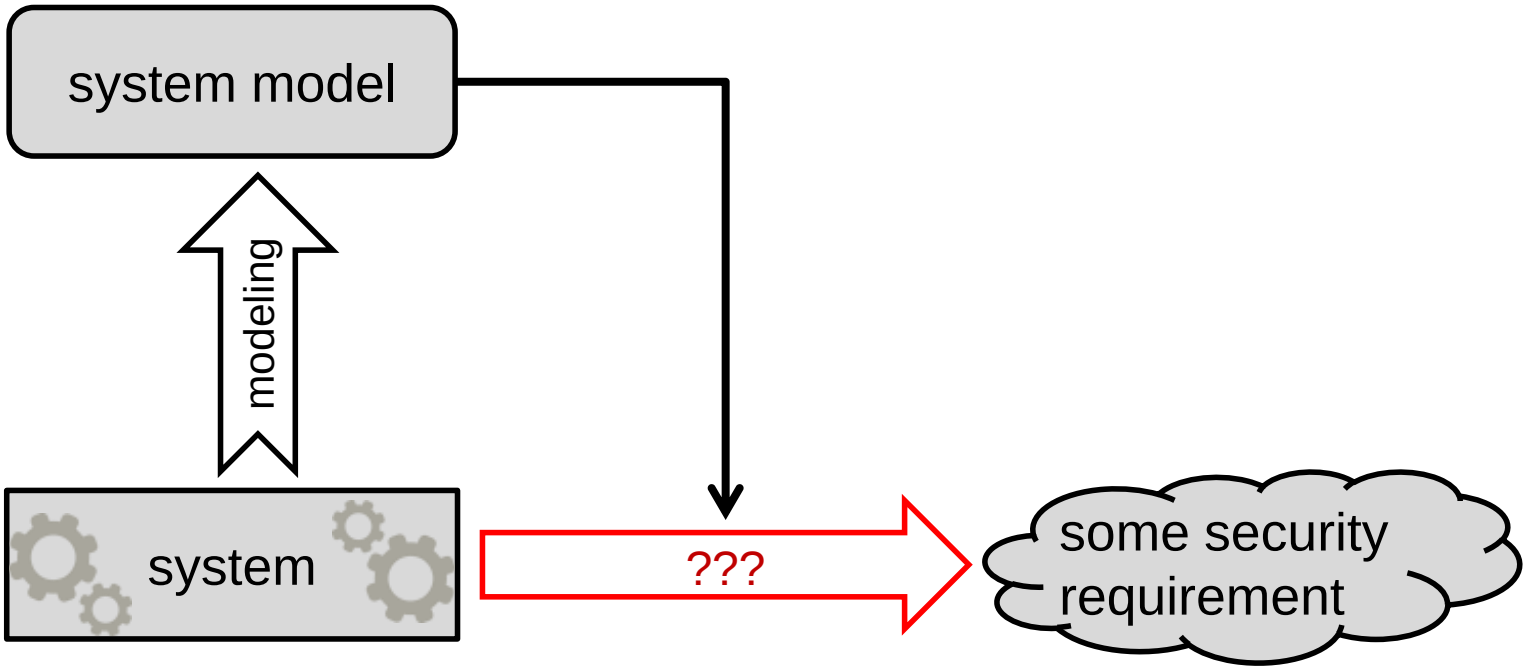
- e.g. anonymity, authenticity, non-reputability, privacy, …

For instance, privacy can be expressed using CIA

- confidentiality of information you don't want to share means: You choose what you let other people know.

Mechanism-centric Security (1)

Ensuring security

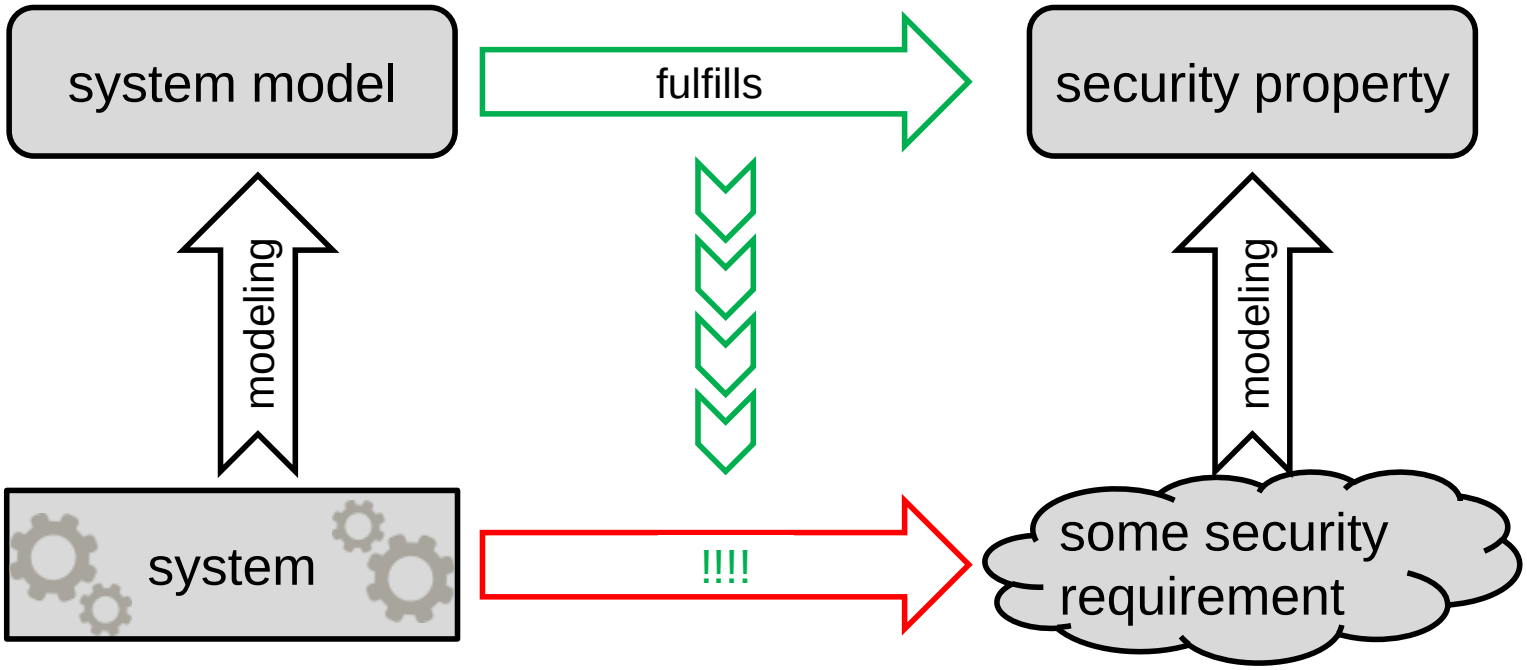


Is mechanism-centric security alone enough?

Mechanism-centric Security (2)

Property-centric Security

Modeling security as a property and then ensuring its satisfaction



Property-centric view should complement mechanism-centric one!

What is a property?

A property is an essential or distinctive attribute or quality of a thing.

Satisfaction of a property

- A system either has a given property or does not have it.
- If a system does not have a property, the system violates this property.

Example

A horse might satisfy the properties "… is fast.", "… is brown.", "… is big.".

How to formulate a property?

- "… incorporates an access control mechanism" is a property, but the property-centric view does not provide added value for such a property
- "… is secure" nicely abstracts from security mechanism, but it is not a property, as a program might be "secure" for a user, but not for another

How to characterize conditions that are properties?

Characterizing Properties (1)

Properties can often be characterized by predicates on system runs.

Convention: Such a property is satisfied if the predicate holds for each run that this system could possibly perform.

How to characterize a property by a predicate on system runs?

- One defines 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.

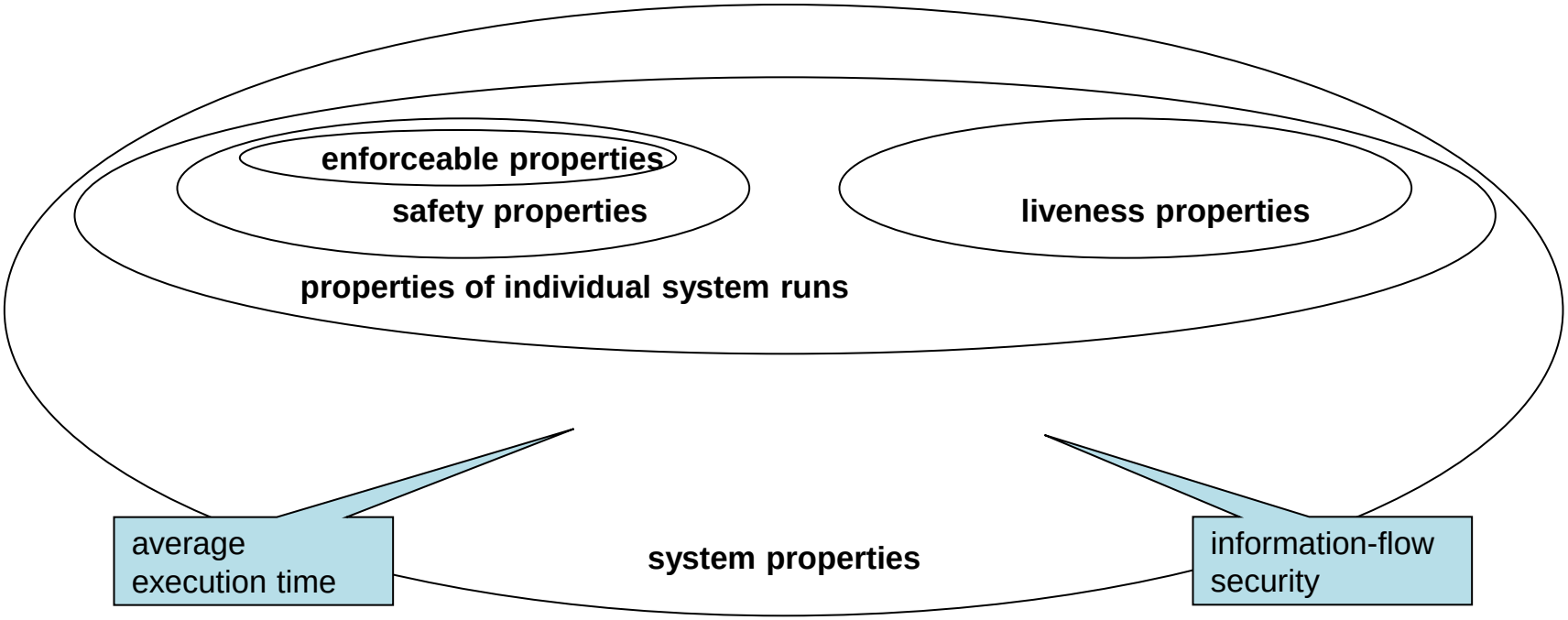
Example

- A system is terminating if each possible system run is finite

Many properties of interest in Computer Science can be characterized by predicates on system runs, but there are also properties that cannot be characterized in this way.

Characterizing Properties (2)

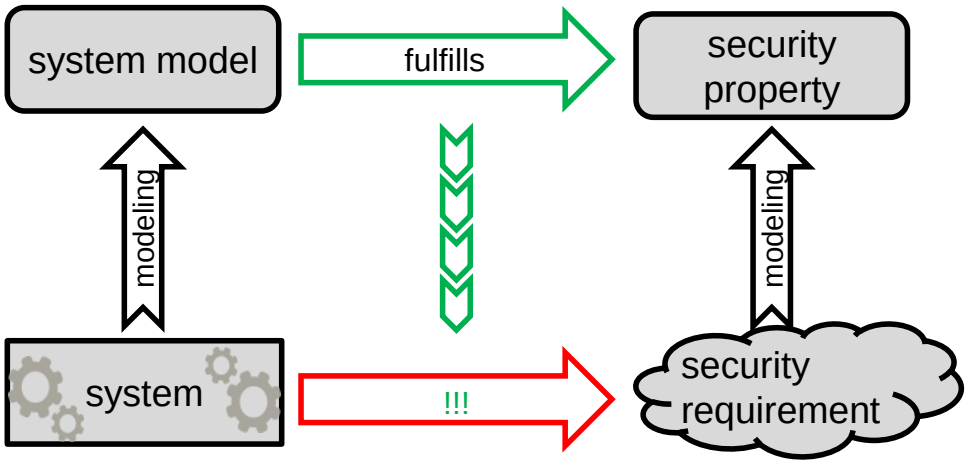
Properties can be classified according to their characterizations



Predicates on individual system runs are not expressive enough to characterize all security properties! (more by Michael Clarkson)

From Mechanisms to Properties

Property-centric security



Using security mechanisms to establish security properties

- access control ⇛ authorized accesses only
- usage control ⇛ secure usage
- information-flow control ⇛ information-flow security

The focus of this tutorial will be on information-flow security.

Why Information-Flow Security?

Example

- You install an app on your cell phone.
- How can you be sure that this app does not leak
  - your calendar,
  - your contacts,
  - your call history, or
  - your physical location?

Even if you feel OK with that an app leaks some of your private data, are you still OK if it leaks all of your private information in all cases?

- The purpose of information-flow security is to limit what data is leaked.

Note that access control is insufficient if the app's functionality needs access to your private data and also to information sinks where you don't want your private data to go.

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

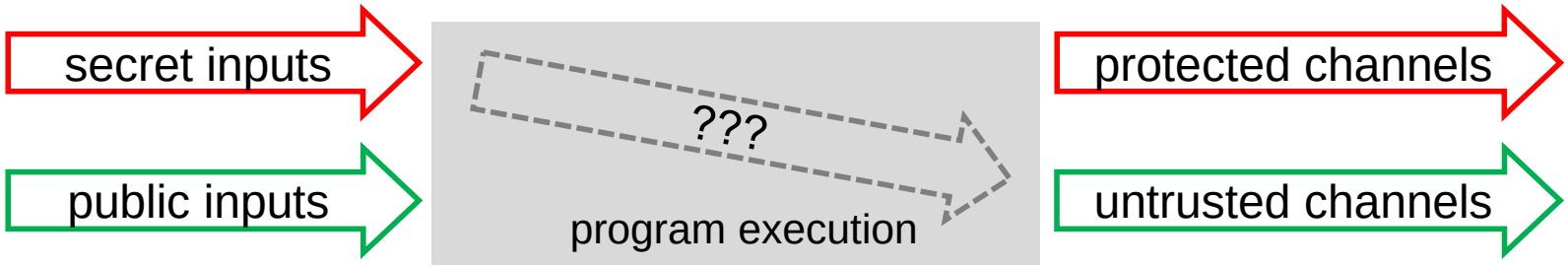
Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

What is information-flow security? (1)

Is there any danger that secrets are leaked to untrusted sinks?



Information leakage

An attacker makes observations during a program run that allow him to deduce secret information.

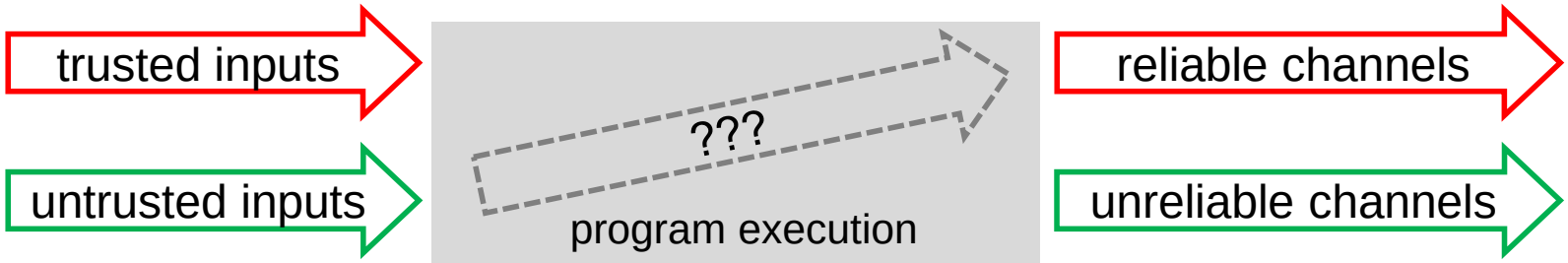
Information-flow security [for confidentiality]

There is no danger of information leakage.

Note: Information-flow security can also be understood as integrity.

What is information-flow security? (2)

Is there any danger of corruption?



Corruption

An attacker provides untrusted input that affects the output on reliable channels.

Information-flow security [for integrity]

Output on reliable channels is no less trustworthy than trusted input.

In this tutorial, I will focus on confidentiality (previous slide).

Information Leakage

Might running this program leak secret information?

```
…
x := exp
…
```

Yes, if

- the value of the target x can be observed by the attacker

and

- the value of the expression exp depends on secrets
  or whether the assignment is executed or not depends on secrets

Hence, for establishing information-flow security, one needs

- access control ⇛ Knowing what the attacker can observe.
- data-flow analysis ⇛ … if values of expressions depend on secrets.
- control-flow analysis ⇛ … if reachability of statements depends on ..

How might information leak? (1)

Explicit leakage (or: direct leakage)

output **secret** to **untrusted-channel**

attacker learns initial value of **secret**

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**

How does the attacker deduce secret information?

- When this program is run, the attacker observes some message **msg** .
- From this message and the program code, he deduces that the initial value of **secret** must have been the same as the value of **msg** .

How might information leak? (2)

Implicit leakage (or: indirect leakage)

attacker learns whether initial value of secret was positive

```
if secret > 0 then output 1 to untrusted-channel
              else output 0 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**

How does the attacker deduce secret information?

- When this program is run, the attacker observes some message **msg** .
- If value of **msg** is 1 then, he deduces that the initial value of **secret** must have been greater than zero. If value of **msg** is 0 then, the initial value of **secret** must have been smaller or equal than zero.

Conservative assumption: The attacker knows the program code.

How might information leak? (3)

Information leakage via non-termination

```
while secret ≥ 0 do skip od;
output 1 to untrusted-channel
```

if attacker sees 1 then he learns that the initial value of **secret** was negative

Information leakage via non-progress

```
public := 0;
while true do
      output public to untrusted-channel;
      if public < secret
         then public := public+1
         else while true do skip od
      fi; od
```

attacker learns initial value of **secret** from the last output that he sees in a run

How might information leak? (4)

Information leakage via array look-up

```
output  public-array[secret]  to  untrusted-channel ;
for  i := 0 to length(public-array) do
   output  public-array[i]  to  untrusted-channel ;
od
```

attacker can narrow down the initial value of **secret** to those positions where the value output equals the first output

Information leakage via array modification

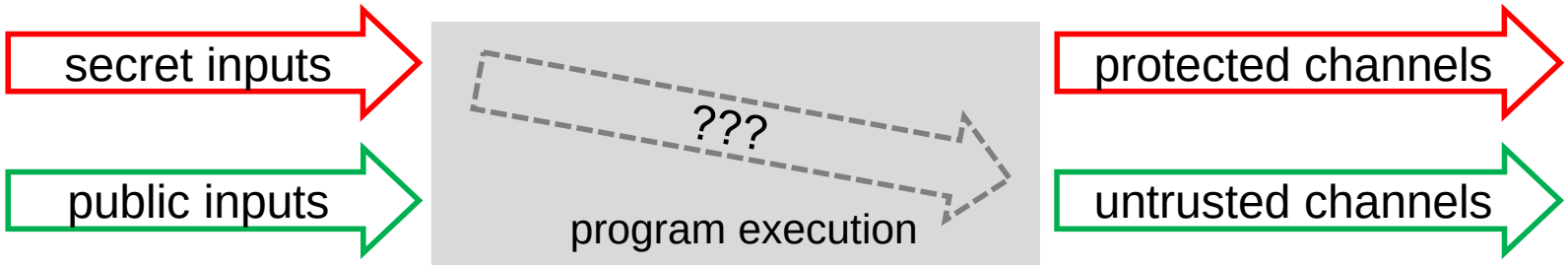
```
public-array[secret] := 42;
for  i := 0 to length(public-array) do
   output  public-array[i]  to  untrusted-channel ;
od
```

attacker can narrow down the initial value of **secret** to those positions where the value 42 is output

Many further possibilities for leaking information exist, e.g., via dynamic dispatch, via exceptions, due to concurrency, … .

What is information-flow security? (3)

Is there any danger that secrets are leaked to untrusted sinks?



Information leakage

An attacker makes observations during a program run that allow him to deduce secret information.

Information-flow security [for confidentiality]

There is no danger of information leakage.

Definition of information-flow security depends on what is secret, what the attacker can observe, and what the attacker can deduce.

Confidentiality and Attacker Models

Information that one might want to keep confidential

- initial values of dedicated program variables
- input on dedicated channels provided during a program run
- strategies used to determine the next input on dedicated channels
- …

Capabilities of attackers that an attacker model could cover:

- attacker knows the program code

and

- attacker observes final result of a program run
- attacker observes output occurring during a program run
- attacker observes intermediate values of variables during a run

More powerful attacker models are possible (e.g. timing, power).

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

What is noninterference? (1)

Noninterference informally

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

Why does noninterference characterize information-flow security?

When a noninterferent program is run, the attacker makes observations. Since his observations do not depend on secrets, the attacker does not know more secrets after the run than before, i.e., no information is leaked.

Definition of noninterference depends on what is secret, what the attacker can observe, and how “dependence” is defined.

What is noninterference? (2)

Noninterference informally (like before)

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

Example

Is the following program noninterferent?

```
if secret > 0 then output 1 to untrusted-channel
              else output 0 to untrusted-channel
```

Better: Is the program noninterferent if the initial value of **secret** is a secret and the attacker can observe messages on **untrusted-channel** .

Answer: Under these conditions, the program is **not** noninterferent.

Argument: Which value the attacker observes on **untrusted-channel** during a program run depends on the initial value of **secret** .

That is, the attackers observations depend on a secret.

Here, semantics of programming language is relevant

Ensuring Noninterference

Possibilities for verifying that a program is noninterferent

- direct verification using the unwinding technique
- dedicated program analysis techniques (different traditions exist: type systems, program dependence graphs, abstract interpretation, …)
- dedicated program logics
- general-purpose program logics (using self composition)

Possibilities for enforcing noninterference

- program transformations
- dynamic program analysis techniques (attention: some pitfalls)
- hybrid analysis techniques (combine static and dynamic analysis)

For verifying the soundness of such verification and enforcement techniques our definition of noninterference is too imprecise.

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

Syntax

Syntax of a programming language

- A formally defined language.

Example

- $prog$ ::= stop | skip | $x := aexp$
  | input x from ch | output $aexp$ to ch
  | $prog; prog$ | if $bexp$ then $prog$ else $prog$ fi
  | while $bexp$ do $prog$ od
- $aexp \in AExp$ "arithmetic expressions"
- $bexp \in BExp$ "boolean expressions"
- $x \in Var$ "program variables"
- $ch \in Ch$ "communication channels"

You will have a good intuition about the meaning of such programs.

Operational semantics gives a precise meaning to programs.

Operational Semantics (1)

Memory state

A memory state is a function $mem: Var \to Val$.

$mem(x)$ is the value of the variable $x \in Var$ in the memory state mem.

Evaluation of arithmetic expressions

$\langle aexp, mem \rangle \Downarrow n$ models that the arithmetic expression $aexp$ evaluates to the number $n \in Val$ in the memory state mem.

Evaluation of boolean expressions

$\langle bexp, mem \rangle \Downarrow b$ models that the boolean expression $bexp$ evaluates to the boolean value $b \in \{true, false\}$ in the memory state mem.

We leave the syntax and semantics of arithmetic expressions and of boolean expressions otherwise unspecified for now.

Operational Semantics (2)

Configuration

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

- where $Prog$ is the set of all programs and
- where $Mem = Var \to Val$ is the set of all memory states

Intuition

A configuration $\langle prog, mem \rangle$ models a snapshot during a program run,

- where $prog$ models the program that remains to be executed and
- where mem models the current values of all program variables.

Example

$$\langle x := 42;\ y := x * y\ , [x \mapsto 0, y \mapsto 0] \rangle$$

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

Operational Semantics (3)

Labeled transitions capturing computation steps

$$\langle prog, mem\rangle \xrightarrow{\alpha} \langle prog', mem'\rangle$$

- where $\langle prog, mem\rangle$ and $\langle prog', mem'\rangle$ are configurations and
- where $\alpha \in Ev$ is an event (Ev remains unspecified for now)

Intuition

A transition $\langle prog, mem\rangle \xrightarrow{\alpha} \langle prog', mem'\rangle$ captures a computation step

- where $\langle prog, mem\rangle$ is the configuration before the step,
- where $\langle prog', mem'\rangle$ is the configuration after the step, and
- where the event α captures additional information, e.g. a value output

Example

$$\langle x := 42;\ y := x * y\ , [x \mapsto 0, y \mapsto 0]\rangle \dot{\rightarrow} \langle y := x * y, [x \mapsto 42, y \mapsto 0]\rangle$$

Operational Semantics (4)

Form of derivation rules

$$\frac{premise1 \cdots premise\ n}{conclusion}$$

Intuition

If each premise is true then the conclusion is also true.

A derivation rule for assignments

$$\frac{\langle aexp, mem\rangle \Downarrow n}{\langle x := aexp, mem\rangle \dot{\rightarrow} \langle \mathrm{stop}, mem[x \mapsto n]\rangle}$$

the arithmetic expression $aexp$ evaluates to some number n

nothing remains to be executed

the new memory state differs from the old one only for variable x

Operational Semantics (5)

Derivation rules for sequential composition

$$\frac{\langle prog1, mem\rangle \xrightarrow{\alpha} \langle \text{stop}, mem'\rangle}{\langle prog1; prog2, mem\rangle \xrightarrow{\alpha} \langle prog2, mem'\rangle}$$

program $prog1$ terminates, results in memory state mem', and causes event α

event α is propagated from the premise

$prog2$ remains to be executed

the memory state mem' is propagated from the premise

$$\frac{\langle prog1, mem\rangle \xrightarrow{\alpha} \langle prog1', mem'\rangle}{\langle prog1; prog2, mem\rangle \xrightarrow{\alpha} \langle prog1'; prog2, mem'\rangle}$$

program $prog1$ performs a step resulting in memory state mem' and causing event α without terminating

event α is propagated from the premise

$prog1'; prog2$ remains to be executed

the memory state mem' is propagated from the premise

Operational Semantics (6)

A derivation

$$\dfrac{\dfrac{\dfrac{}{\langle 42, [x \mapsto 0, y \mapsto 0]\rangle \Downarrow 42}}{\langle x := 42, [x \mapsto 0, y \mapsto 0]\rangle \dot{\rightarrow} \langle \text{stop}, [x \mapsto 42, y \mapsto 0]\rangle}}{\langle x := 42;\ \ y := x * y\ , [x \mapsto 0, y \mapsto 0]\rangle \dot{\rightarrow} \langle y := x * y, [x \mapsto 42, y \mapsto 0]\rangle}$$

The rules applied in this derivation

$$\dfrac{\langle prog1, mem\rangle \xrightarrow{\alpha} \langle \text{stop}, mem'\rangle}{\langle prog1; prog2, mem\rangle \xrightarrow{\alpha} \langle prog2, mem'\rangle}$$

$$\dfrac{}{\langle n, mem\rangle \Downarrow n}$$

a rule for evaluating arithmetic expressions that are constants

$$\dfrac{\langle aexp, mem\rangle \Downarrow n}{\langle x := aexp, mem\rangle \dot{\rightarrow} \langle \text{stop}, mem[x \mapsto n]\rangle}$$

Operational Semantics (7)

Rules for conditionals and loops

$$\frac{\langle bexp, mem\rangle \Downarrow false}{\langle \text{if } bexp \text{ then } prog1 \text{ else } prog2 \text{ fi}, mem\rangle \dot{\rightarrow} \langle prog2, mem\rangle}$$

$$\frac{\langle bexp, mem\rangle \Downarrow true}{\langle \text{if } bexp \text{ then } prog1 \text{ else } prog2 \text{ fi}, mem\rangle \dot{\rightarrow} \langle prog1, mem\rangle}$$

$$\frac{\langle bexp, mem\rangle \Downarrow false}{\langle \text{while } bexp \text{ do } prog \text{ od}, mem\rangle \dot{\rightarrow} \langle stop, mem\rangle}$$

$$\frac{\langle bexp, mem\rangle \Downarrow true}{\langle \text{while } bexp \text{ do } prog \text{ od}, mem\rangle \dot{\rightarrow} \langle prog; \text{while } bexp \text{ do } prog \text{ od}, mem\rangle}$$

Operational Semantics (8)

Rule for skip

$$\frac{}{\langle \text{skip}, mem\rangle \dot{\rightarrow} \langle \text{stop}, mem\rangle}$$

Rules for output

$$\frac{\langle aexp, mem\rangle \Downarrow n}{\langle \text{output } aexp \text{ to } ch, mem\rangle \xrightarrow{output(ch,n)} \langle \text{stop}, mem\rangle}$$

$$\frac{}{\langle \text{input } x \text{ from } ch, mem\rangle \xrightarrow{input(ch,n)} \langle \text{stop}, mem[x \mapsto n]\rangle}$$

Operational Semantics (9)

Labeled transitions capturing runs

$$\langle prog, mem\rangle \overset{\tau}{\Rightarrow} \langle prog', mem'\rangle$$

- where $\langle prog, mem\rangle$ and $\langle prog', mem'\rangle$ are configurations and
- where $\tau \in Ev^*$ is a sequence of events

Rules

$$\frac{}{\langle prog, mem\rangle \overset{[]}{\Rightarrow} \langle prog, mem\rangle}$$

$$\frac{\langle prog, mem\rangle \overset{\alpha}{\rightarrow} \langle prog', mem'\rangle \qquad \langle prog', mem'\rangle \overset{[\alpha 1,..,\alpha n]}{\Longrightarrow} \langle prog'', mem''\rangle}{\langle prog, mem\rangle \overset{[\alpha,\alpha 1,...,\alpha n]}{\Longrightarrow} \langle prog'', mem''\rangle}$$

Some Exercises (Homework 1)

1. Add the command $\text{repeat}\, prog\, \text{until}\, bexp$ to the syntax of our programming language. Formalize the semantics of this command by adding 2 rules.
2. Define sets of arithmetic and boolean expressions (i.e. $AExp$ and $BExp$) and rules for $\langle aexp, mem\rangle \Downarrow n$ and $\langle bexp, mem\rangle \Downarrow b$.
   - Remark 1: We declared the sets $AExp$ and $BExp$ (on Slide 30), but we did not define these two sets so far.
   - Remark 2: You are free to choose these languages as you like.

End of presentation on August 31

break & time for homework

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

Formalizing Noninterference (1)

Noninterference informally (like before)

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

Choice: What is secret?

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?

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

How to formally define noninterference for these choices?

Formalizing Noninterference (2)

Choice: What is secret? (like before)

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

Choice: What can the attacker observe? (like before)

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

Indistinguishability of memories for the attacker

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

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

Formalizing Noninterference (3)

Choice: What is secret? (like before)

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

Choice: What can the attacker observe? (like before)

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

A formal definition of noninterference for these choices

A program $prog$ is noninterferent if and only if

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

Definition of noninterference is based on counter-factual reasoning.

Roadmap

Part 1: An Introduction to Noninterference

- property-centric security vs mechanism-centric security
- information-flow security and information leakage
- noninterference: an informal definition
- a simple introduction to operational semantics
- noninterference: a formal definition
- example system: Cassandra
- exercises

Part 2: Noninterference for Multi-threaded Programs

Part 3: Recent Results on Concurrent Noninterference

Bibliography

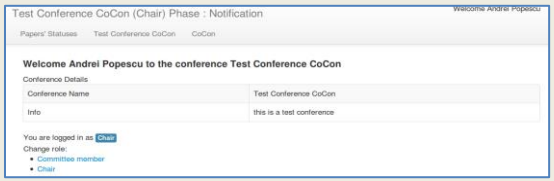
Nice Theory – Can it be applied?

Reference scenarios in the DFG priority program RS³:

www.spp-rs3.de

security in web-based workflow management systems

CoCon



software security for mobile devices

the RS³ Certifying Appstore

see talk by David Schneider today

security in E-voting

select secure election

Some Exercises (Homework 2)

3. Argue why the program from Slide 27 does not satisfy our formal definition of noninterference.
4. Does the formal definition of noninterference on Slide 44 faithfully capture our informal definition of noninterference if
   a. the attacker can observe the number of steps,
   b. the attacker can observe interactions on the channel $open \in Ch$.
5. If your answer to 3a and/or 3b is NO, then modify the formal definition of noninterference such that it faithfully captures our informal definition of noninterference under the given conditions.
6. How to modify the operational semantics if the program environment chooses inputs on a channel based on prior interactions on this channel?

Some Lessons Learned

- How to complement the mechanism-centric view on security by a property-centric view in a useful way?
- What is information-flow security and what is it good for?
- How to capture information-flow security by noninterference?
- How to define noninterference formally?
  - based on the operational semantics of a programming language
- Which definition of noninterference is suitable depends on
  - which secrets need to be kept confidential,
  - what the attacker can observe, and
  - the model of execution.

Bibliography

Early definitions of noninterference-like properties (selected)

[Feiertag et al 1977] R.J. Feiertag, K.N. Levitt, L. Robinson: *Proving Multi-level Security of a System Design*. In: Proceedings of ACM Symposium on Operating Systems Principles. pp. 35-41 (1977)

[Cohen 1977] E. Cohen: *Information Transmission in Computational Systems*. In: Proceedings of ACM Symposium on Operating Systems Principles. pp. 133-139 (1977)

[Goguen/Meseguer 1982] J.A. Goguen, J. Meseguer: *Security Policies and Security Models*. In: Proceedings of IEEE Security and Privacy. pp. 11-20 (1982)

[Sutherland 1986] D. Sutherland: *A Model of Information*. In: Proceedings of National Computer Security Conference. (1986)

[McCullough 1987] D. McCullough: *Specifications for Multi-level Security and a Hook-Up Property*. In: Proceedings of IEEE Security and Privacy. pp. 161-166 (1987)

[Mantel 2011] H. Mantel: *Information Flow and Noninterference*. In: Encyclopedia of Cryptography and Security, 2nd ed. (2011)

Early information-flow analysis with or w/o soundness results (selected)

[Denning/Denning 1977] D.E. Denning, P.J. Denning: *Certification of Programs for Secure Inforamtion Flow.* In: Communications of the ACM 20(7). pp. 504-513 (1977)

[Goguen/Meseguer 1984] J.A. Goguen, J. Meseguer: *Unwinding and Inference Control*. . In: Proceedings of IEEE Security and Privacy,: pp. 75-87 (1984)

[Rushby 1992] J. Rushby: *Noninterference, Transitivity, and Channel-Control Security Policies*. TR CSL-92-02, SRI International. (1992)

[Volpano/Smith/Irvine 1996] D. Volpano, G. Smith, C. Irvine*: A Sound Type System for Secure Flow Analysis*. In: Journal of Computer Security 4(3). pp. 1-21 (1996)

[Myers/Liskov 1997] A.C. Myers, B. Liskov: *A Decentralized Model for Information Flow Control*. In: Proceedings of ACM Symposium on Operating Systems Principles. pp. 129-142 (1997)

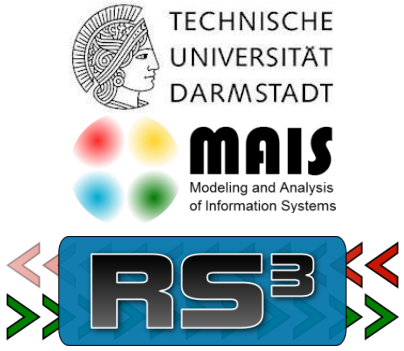
FOSAD Summer School 2015

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, …



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

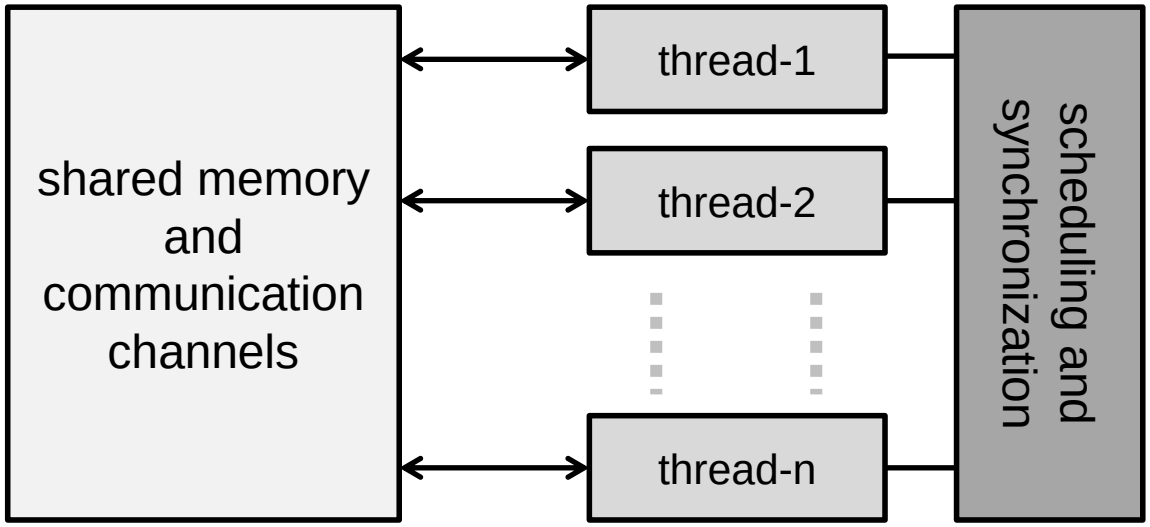
Part 3: Recent Results on Concurrent Noninterference

Exercises

Bibliography

What is a multi-threaded program?

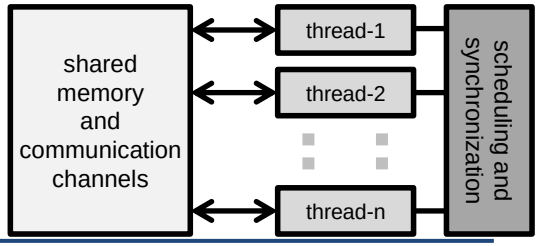
Execution of a multi-threaded program



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



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: $Var \to Val$ is a memory states (like in Part 1)

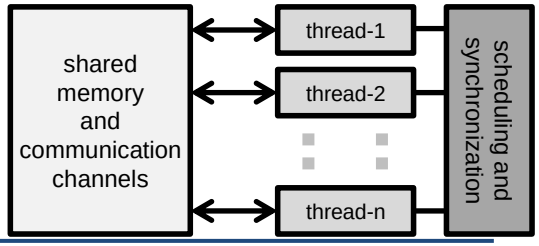
Intuition: $\langle\langle prog_1 || \cdots || prog_n , mem\rangle\rangle$ models a snapshot,

- 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 $\langle\langle x := 42 \;||\; y := x * y , [x \mapsto 0, y \mapsto 0] \rangle\rangle$

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

Local Configurations



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

- where $prog \in Prog$ is a program
- where $Mem = Var \to Val$ 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

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

or

$$gcnf \xrightarrow{\alpha} gcnf'$$

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

$$\frac{i \in \{1, \ldots, n\} \quad \alpha \in Ev \quad \langle prog_i, mem\rangle \xrightarrow{\alpha} \langle prog_i', mem'\rangle \quad \forall j \in \{1, \ldots, n\}\backslash\{i\}.\, prog_j{}' = prog_j}{\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \xrightarrow{\alpha} \langle\langle prog_1{}' || \cdots || prog_n{}', mem'\rangle\rangle}$$

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)

$$\frac{i \in \{1,\ldots,n\} \quad \alpha \in Ev \quad \langle prog_i, mem\rangle \xrightarrow{\alpha} \langle prog_i', mem'\rangle \quad \forall j \in \{1,\ldots,n\}\backslash\{i\}.\, prog_j{}' = prog_j}{\langle\langle prog_1 || \cdots || prog_n, mem\rangle\rangle \xrightarrow{\alpha} \langle\langle prog_1{}' || \cdots || prog_n{}', mem'\rangle\rangle}$$

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: 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

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

event $sync$ is emitted

Programming language after adding the barrier command

- $prog$ ::= stop | skip | $x := aexp$ | barrier
  | input x from ch | output $aexp$ to ch
  | $prog; prog$ | if $bexp$ then $prog$ else $prog$ fi
  | while $bexp$ do $prog$ od

Synchronization (2)

Barrier synchronization

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

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
              else skip fi;
output 0 to untrusted-channel
```

||

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

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

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

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

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

2. Which values can τ and mem' take?
3. 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** .
4. Add the command $\text{spawn}\ prog$ 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.
5. Augment the language by further synchronization commands.

End of presentation on September 1

break & time for homework