



Formal methods for software security (invited talk)

Thomas Jensen

► **To cite this version:**

Thomas Jensen. Formal methods for software security (invited talk). Journées Nationales 2017 Pré-GDR Sécurité Informatique, Jun 2017, Paris, France. pp.1-31. <hal-01658835>

HAL Id: hal-01658835

<https://hal.inria.fr/hal-01658835>

Submitted on 7 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



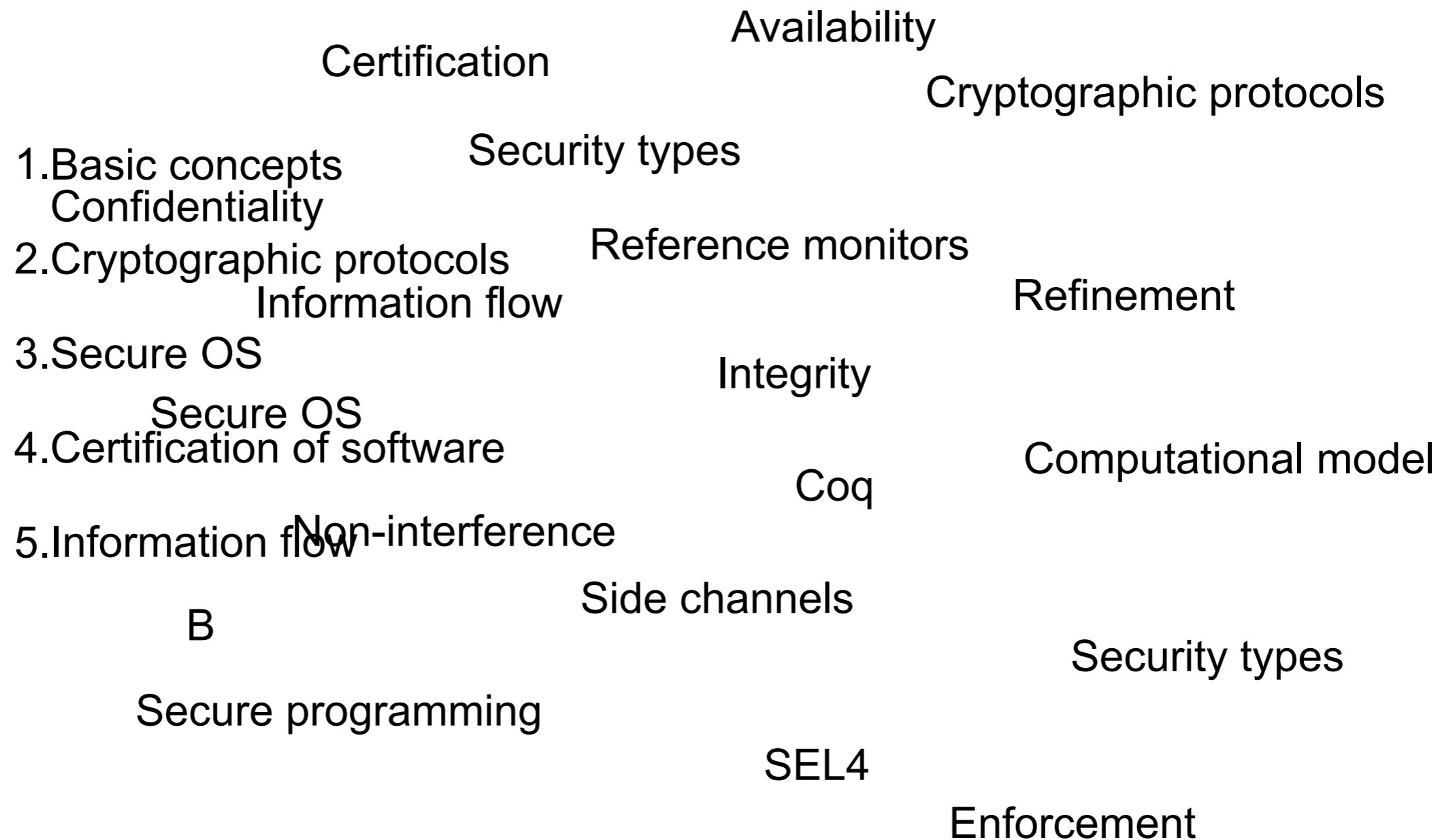
Formal methods for software security

Thomas Jensen, INRIA

GDR Sécurité

Paris, 1 Juin 2017

Formal methods for software security



Basic security concepts

Confidentiality

- the software will not disclose my secrets ... at least not more than I'm willing to accept.

Integrity

- data and decisions are not influenced by intruders.

Availability

- software and services are there when I need them.

Security \neq Safety

... but they are strongly related

Attacker model

Security is open-ended!

The question

Is my software secure?

must be complemented by an **attacker model**, stating the threats we are up against.

Specify the attackers

- observational power (output, network messages, time,...)
- actions (code insertion, message injection,...)
- access to machine (physical, through network,...)

Enforcement mechanisms

Certification of applications

- Common Criteria,
- Formal methods for reaching upper levels.

Security-enhancing software development

- secure programming guidelines,
- secure compilation.

Static code analysis

- eg, Java's byte code verifier, information flow analysis.

Reference monitors and run-time analysis.

Cryptographic protocols

Models of cryptographic protocols

Symbolic models

- specified as a series of exchanges of messages
- assuming perfect cryptography

Example : two agents A, B

1. $A \hookrightarrow B : \{N_A, A\}_{K_B}$
2. $B \hookrightarrow A : \{N_A, N_B, B\}_{K_A}$
3. $A \hookrightarrow B : \{N_B\}_{K_B}$

Attackers may

- intercept and re-send messages,
- encrypt and decrypt messages (with available keys).

Verification

Model

- state = current message + state of A,B, and attacker
- rewriting rules defining protocol and attacker

$$(\{msg\}_{key}, \dots, key, \dots) \rightarrow (msg, \{msg\}_{key}, \dots, key, \dots)$$

Security properties

- secrecy ("no state where attacker has the secret")
- authentication, re-play, ...
- specific properties ("key may not be used on stored content", "vote has been counted")

Tools

A variety of mature tools

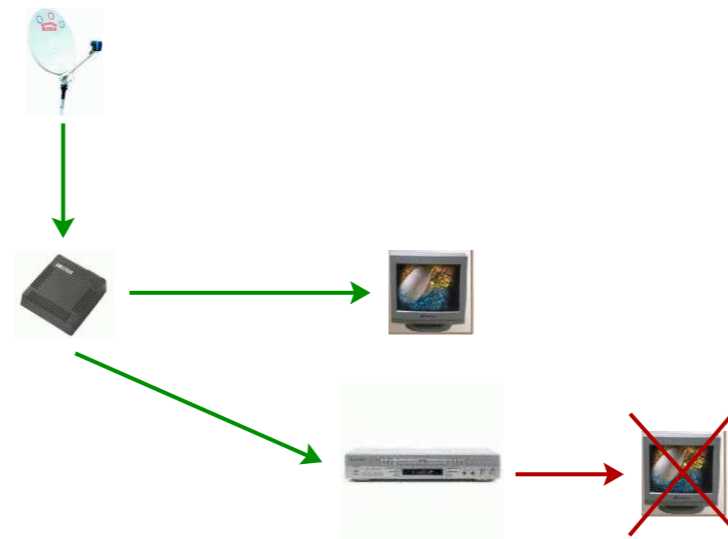
- AVISPA, Tamarin, ProVerif, APTE, ...

based on solid theory

- term and multi-set rewriting, Horn clauses, π -calculus, ...

Interfaces for writing and animating protocols

- eg as Message Sequence Charts (SPAN).



Computational models

A model closer to reality:

- Messages: bit strings,
- Crypto primitives: functions on bit strings,
- Attacker : any probabilistic poly-time Turing machine.

Properties proved for all traces, **except** for a set of traces of negligible probability.

Secrecy: attacker can distinguish secret from random number with only infinitesimal probability.

Proofs by refinement of models.

See eg. the `cryptoverif` tool

Implementations of crypto protocols

Security concerns with **implementations** of protocols and basic operations of cryptography.

Implementations of cryptographic primitives are prone to **side channel** attacks:

- leaking secrets via timing or energy consumption,
- a challenge for implementors

Implementations of entire protocols are prone to programming errors:

- see the Verified TLS project for building a formally verified implementation of TLS.

Secure operating systems

Security and OS

Organized Sharing of resources between processes

- using the same memory
- communicating via IPC

and still guarantee **isolation properties**.



Large, complex software - long history of security alerts.

The SEL4 project

Project run at NICTA 2004-2014.

Formal verification of Liedtke's L4 micro-kernel.

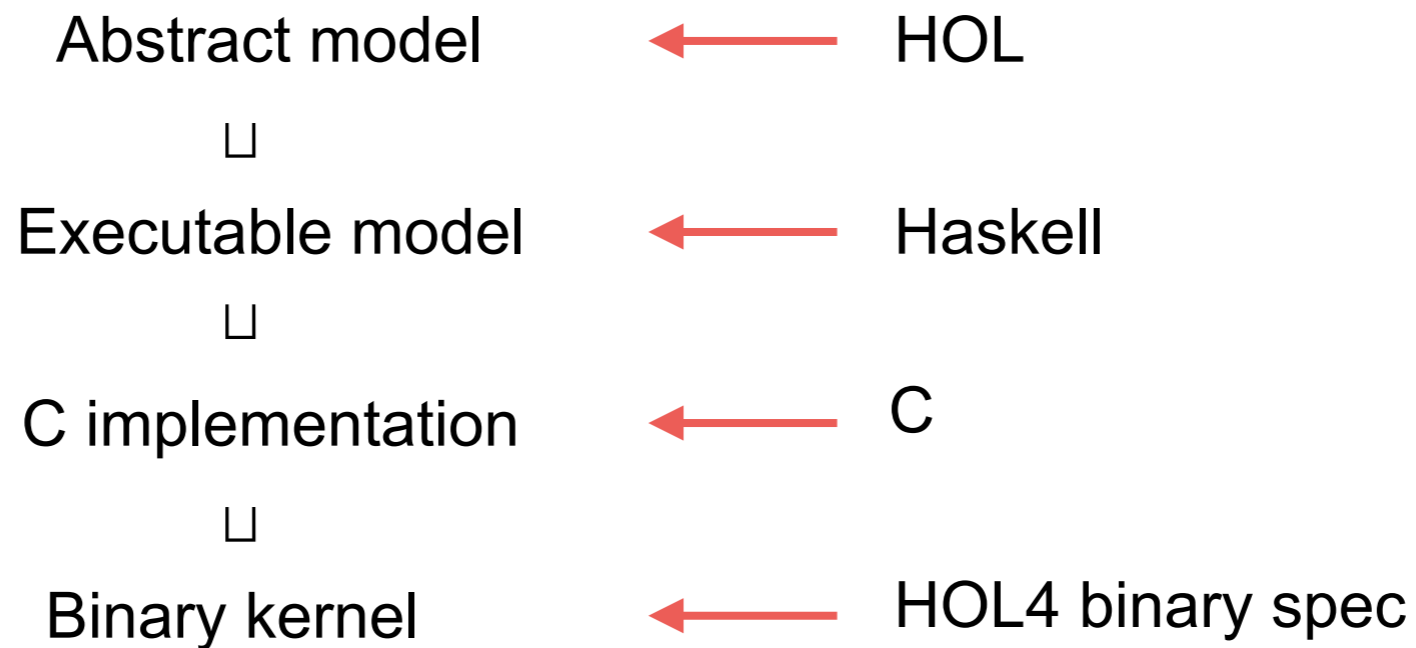
- small code base (9 K Loc),
- threads, memory management, IPC, interrupts, capability-based access control,
- running on ARM,
- verified using the Isabelle/HOL theorem prover.

Prove:

- Functional correctness (and a lot of safety properties)
- Non-interference

SEL4: proof structure

Proof by refinement



On the "Abstract model", build

- access control model,
- integrity and confidentiality proof

200 000 lines of Isabelle/HOL proof

25 person-years

Prove & Run's ProvenCore

SEL4 uses Isabelle/HOL and Haskell

- higher-order logic and lazy functional programming is still not main-stream development tools.

Prove & Run has developed a formally verified microkernel
ProvenCore

- refinement proof method,
- isolation properties.

using their SMART development framework:

- functional, executable specification,
- closer to programmer's intuition,
- equipped with a dedicated prover.

Certification of Java Card applications

Java Card certification

Java Card

- reduced dialect of Java for bank cards and SIM,
- no dynamic loading, reflection, floating points, threads,...
- "resource-constrained" programming practice.

Industrial context:

- Applications developed by third-parties and put on an app store.
- Must be certified according to industry norms (eg, AFSCM* norms for NFC applications).
- Need "light-weight" certification techniques.

*Association Française du Sans Contact Mobile

AFSCM norms/guidelines

Enforce good programming practice and resource usage

- catch exceptions, call methods with valid args,
- no recursion and almost no dynamic allocation,
- don't call method `xxx`.

Avoid exceptions due to

- null pointers, array indexing, class casts,
- illegal applet interaction through the firewall.

The Java Card analyser

A combination of numeric and points-to analysis

- tailored to the application domain,
- take advantage of imposed restrictions,
- precise (flow-sensitive, inter-proc, trace partitioning).

Major challenge: modelling the Java Card API.

Outcome: an abstract model of execution states

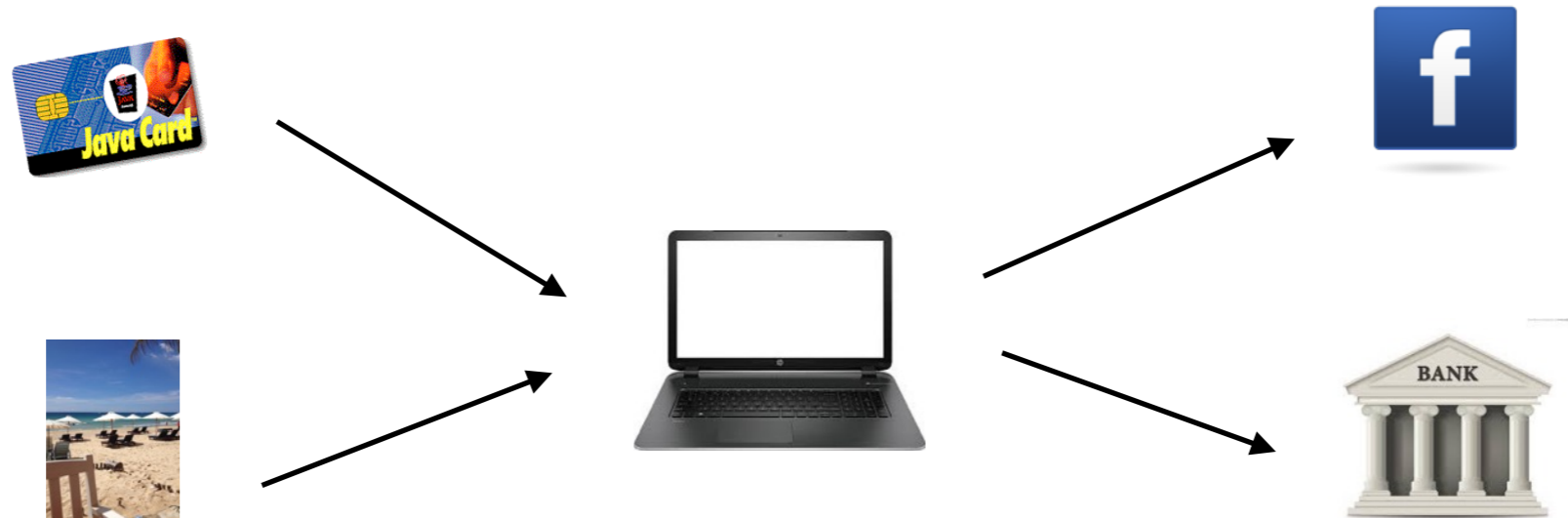
- mined by queries formalising the AFSCM norms.

	A1	A2	A3	A4	A5	A6	A7	A8
Alarms								
ClassCastException	✓	✓	✓	✓	✓	✓	✓	✓
NegativeArraySize	✓	✓	✓	✓	✓	✓	✓	✓
ArrayStoreException	✓	✓	✓	✓	✓	✓	✓	✓
SecurityException	✓	✓	✓	✓	✓	✓	✓	✓
AppletInStaticFields	✓	✓	✓	✓	✓	✓	✓	✓
ArrayConstantSize	✓	✓	✓	✓	✓	✓	✓	✓
InitMenuEntries	✓	✓	✓	✓	✓	✓	✓	✓

	A1	A2	A3	A4	A5	A6	A7	A8
Alarms								
NullPointerException	94	98	99	99	97	98	97	99
ArrayOutOfBounds	71	88	92	87	92	98	90	98
CatchIndividually	46	23	82	31	32	67	57	53
CatchNonISOException	x	x	x	x	x	x	x	x
HandlerAccess	x	✓	x	x	x	✓	✓	✓
AllocSingleton	✓	✓	✓	✓	✓	x	✓	✓
SDOrGlobalRegPriv	x	✓	✓	✓	✓	✓	✓	✓
SWValid	?	✓	✓	✓	✓	✓	✓	✓
ReplyBusy	?	✓	✓	✓	✓	✓	✓	✓

Information flow analysis

Back to confidentiality



Classify data as either

- private/secret/confidential
- public

A basic security policy:

"Confidential data should not become public"

Breaking confidentiality

```
int secret s;    //  $s \in \{0,1\}$   
int public p;
```

```
p := s;
```

Direct flow

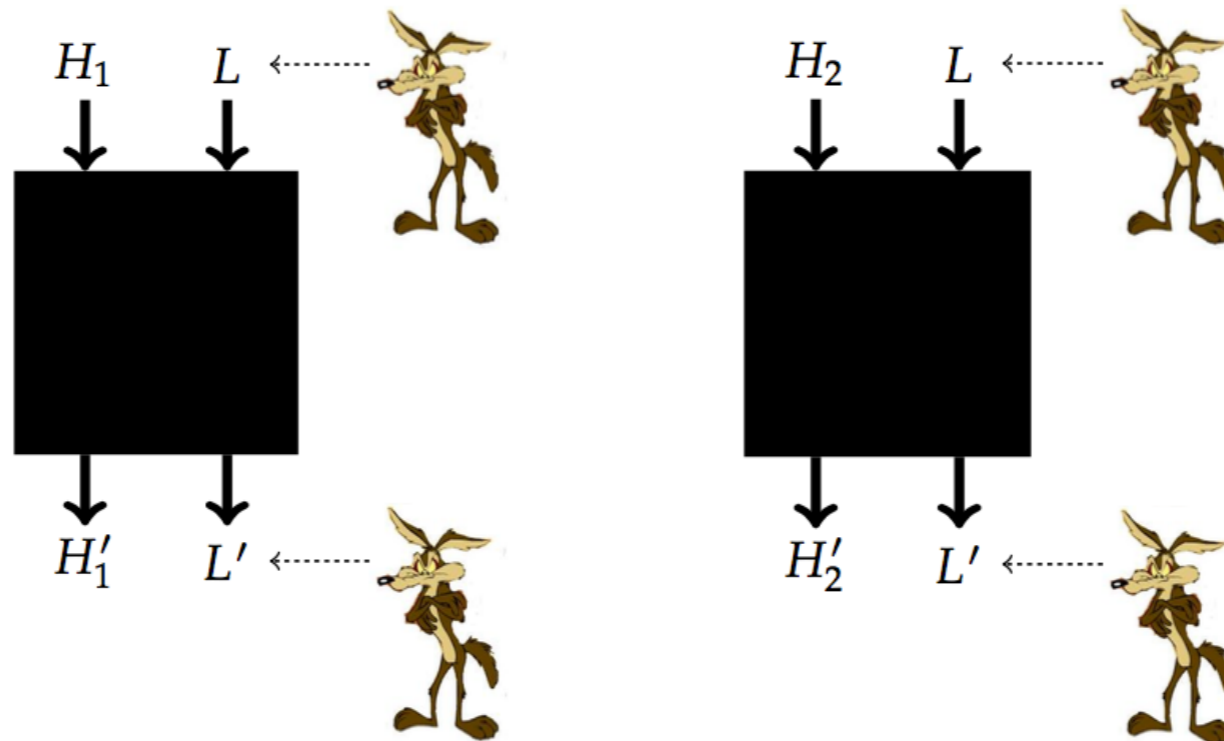
```
if s == 1 then  
  p := 1  
else  
  p := 0
```

Indirect flow

Non-interference

Confidentiality can be formalised as **non-interference**:

Changes in secret values should not be publicly observable



$$\forall s_1, s_2, s'_1, s'_2, \quad s_1 \sim s_2 \wedge (P, s_1) \Downarrow s'_1 \wedge (P, s_2) \Downarrow s'_2 \implies s'_1 \sim s'_2$$

Dynamic enforcement

Add a security level ("taint") to all data and variables

Security levels evolve due to assignments

```
p := s;           // direct flow
```

and when we assign under secret control:

```
if s == 1 then  
  p := 1
```

Secure?

Not enough to enforce confidentiality!

```
int secret s; // s ∈ {0,1}
int public p,q;
```

<pre>p := 0; q := 1;</pre>	<pre>s=0</pre>	<pre>s=1</pre>
<pre>if s == 0 then</pre>	<pre>p=0, q=1</pre>	<pre>p=0, q=1</pre>
<pre> q := 0;</pre>	<pre>p=0, q=0</pre>	<pre>skip</pre>
<pre>if q == 1 then</pre>	<pre>skip</pre>	<pre>p=1, q=1</pre>
<pre> p := 1;</pre>	<pre>p=0</pre>	<pre>p=1</pre>

Need the "no-sensitive-upgrade" principle

Static information flow control

Information flow types:

$$T, T_{\mathbf{x}}, T_{\text{pc}} \in \{\mathbf{public} \sqsubseteq \mathbf{secret}\}$$

Typing rules:

$$\frac{\vdash \mathbf{e} : T \quad T \sqsubseteq T_{\mathbf{x}} \quad T_{\text{pc}} \sqsubseteq T_{\mathbf{x}}}{T_{\text{pc}} \vdash \mathbf{x} := \mathbf{e}} \quad \textit{assign}$$

$$\frac{\vdash \mathbf{e} : T \quad T_{\text{pc}} \sqcup T \vdash \mathbf{S}_i \quad \mathbf{i} = 1, 2}{T_{\text{pc}} \vdash \mathbf{if} \ \mathbf{e} \ \mathbf{then} \ \mathbf{S}_1 \ \mathbf{else} \ \mathbf{S}_2} \quad \textit{if}$$

Well-typed programs are non-interferent

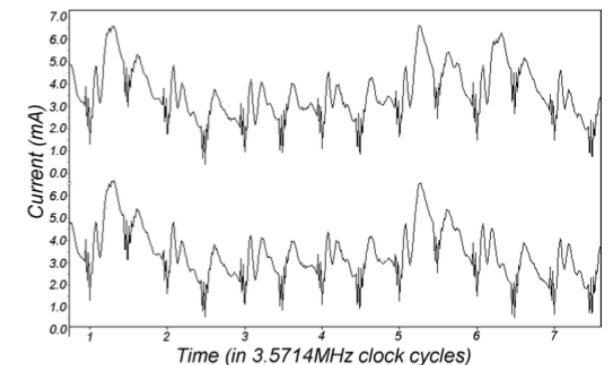
Declassification and side channels

How to declassify confidential data:

- what and when to declassify?
- how much to declassify (passwd, statistics) ?

Information leaks due to other channels

- timing
- energy consumption



Challenge: analysis tools to check constant-time properties of (well-crafted) cryptographic computations.

Coda

Many more topics

Malware detection

- analysis of (obfuscated) binaries.

Access control

- formal models and enforcement.

Attack trees.

Web security

- secure web programming with JavaScript *et al.*

Privacy

- differential privacy (theory vs. practice),
- software in coherence with legislation (EU GDPR).

Thank you

Formal methods for software security

- Formal methods can improve the security of software.
- Come with solid foundations and mature tools.
- More and more industrial applications.
- Technology is becoming main-stream.

Thank you