Equivalence Properties by Typing in Cryptographic Protocols

Véronique Cortier, Niklas Grimm, Joseph Lallemand, Matteo Maffei

March 18, 2019

Introduction: cryptographic protocols

Cryptographic protocols

- Programs aiming to secure communications
  $\Longrightarrow$ payment, web, phones, voting...
- Use cryptographic primitives
  $\Longrightarrow$ encryption, signature, hashing...
- Provide security properties
  $\Longrightarrow$ secrecy of private data, authentication, anonymity, vote secrecy, ...

Example: SSL/TLS, AKA, ...

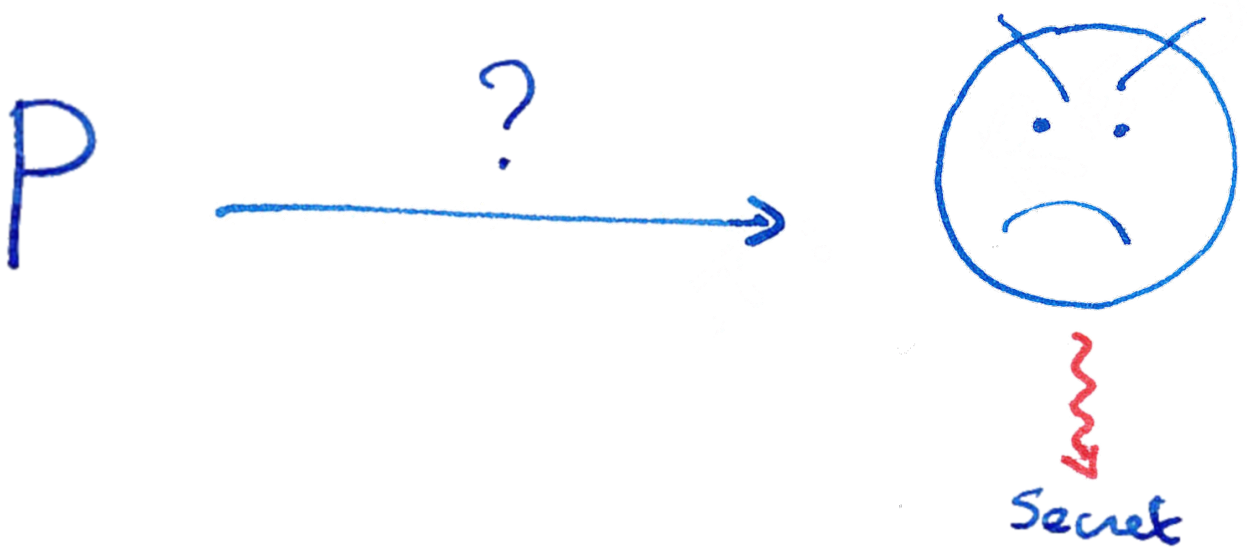
$\Longrightarrow$ Importance of formal analysis of security protocols

Trace properties

Trace properties = satisfied by all traces of a protocol

Example: reachability properties:

Can the attacker learn a given message ?

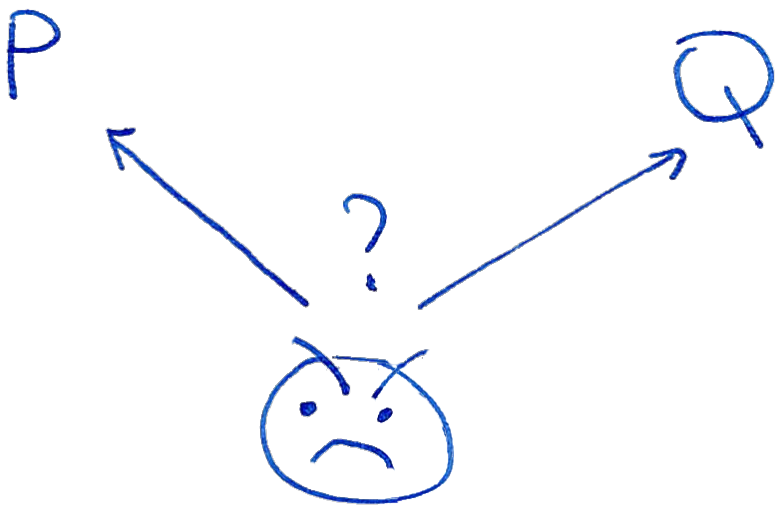


$\Longrightarrow$ secrecy, authentication, ...

Equivalence

Some properties require the notion of equivalence:

Are two protocols indistinguishable for an attacker?

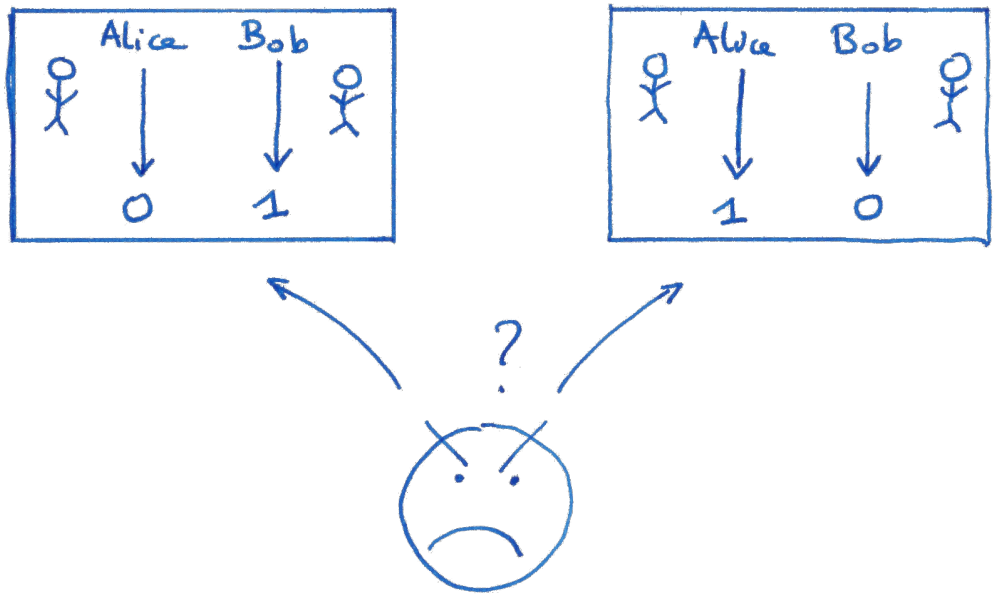


Example:

vote privacy, strong flavours of secrecy, anonymity, unlinkability, . . .

Example: vote privacy

Example: Privacy of the vote in voting protocols



Alice and Bob vote for either 0 or 1.

The values of the votes = 0 and 1 are not secret

The votes are secret if:

$$Alice(0) \mid Bob(1) \approx Alice(1) \mid Bob(0)$$

Proving equivalence

$\longrightarrow$ Procedures and tools exist to prove equivalence
such as ProVerif, Tamarin, Akiss, Apte, Spec, SAT-Equiv, Deepsec

- ProVerif, Tamarin handle unbounded numbers of sessions
- Tamarin is interactive, ProVerif is fully automated but more restrictive
- Deepsec, Akiss, Apte, Spec, SAT-Equiv prove equivalence for a bounded number of sessions

$\longrightarrow$ mostly work by analysing all possible traces

Type systems

Idea: design a type system that ensures protocols satisfy security properties

- Type systems: already applied to trace properties

$$M : Secret \vdash P \implies M \text{ is not deducible in } P$$

- Now: for equivalence

$$\vdash P \sim Q \quad \implies \quad P \approx Q$$

- Efficient (though incomplete) procedures
- Modularity

Type systems

Idea: design a type system that ensures protocols satisfy security properties

- Type systems: already applied to trace properties

$$M : Secret \vdash P \implies M \text{ is not deducible in } P$$

- Now: for equivalence

$$\vdash P \sim Q \quad \implies \quad P \approx Q$$

- Efficient (though incomplete) procedures
- Modularity

Problem:

- Usually: typing $\rightarrow$ overapproximate the set of traces.
- Sound for trace properties, but not equivalence
  $\rightarrow$ might miss that some traces are only possible for P and not Q

Main idea

- Step 1: $\vdash P \sim Q : C$
  typing to ensure no leaks in behaviours
  collect all symbolic messages sent on the network into a *constraint*

- Step 2: $check(C)$
  ensure there are no leaks in the messages sent
  $\longrightarrow$ checking for repetitions

  Example:

$$C = \{\mathtt{enc}(x, k) \sim \mathtt{enc}(a, k),\ \mathtt{enc}(y, k) \sim \mathtt{enc}(b, k)\}$$

  If in some execution we can have $x = y$, equivalence is broken.

Main result: Soundness

Theorem (Soundness)

If $\Gamma \vdash P \sim Q : C$ *and* $\forall\theta.\ C\theta$ *does not leak information, then*

$$P \approx Q$$

Theorem (Procedure to check constraints)

$$\mathit{check}(C) \ \Rightarrow\ \forall\theta.\ C\theta \text{ does not leak information.}$$

Hypotheses:

- atomic keys only
- fixed cryptographic primitives: symmetric and asymmetric encryption, signature, hash, concatenation
- no replication (bounded number of sessions only)

Main result: Soundness

Theorem (Soundness)

If $\Gamma \vdash P \sim Q : C$ *and* $\forall\theta.\ C\theta$ *does not leak information, then*

$$P \approx Q$$

Theorem (Procedure to check constraints)

$$check(C) \Rightarrow \forall\theta.\ C\theta \text{ does not leak information.}$$

Hypotheses:

- atomic keys only
- fixed cryptographic primitives: symmetric and asymmetric encryption, signature, hash, concatenation
- no replication (bounded number of sessions only)

From two to unbounded number of sessions

If one session typechecks, then any number of sessions typecheck:

Theorem (informal)

$$\Gamma \vdash P \sim Q : C \implies \Gamma \vdash !P \sim !Q : !C$$

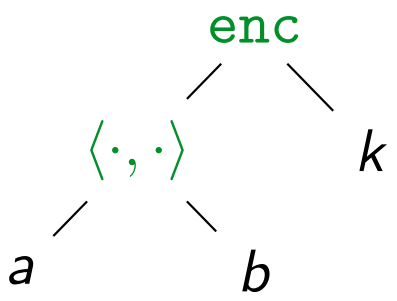
How to check that $!C$ does not leak information?
$\longrightarrow$ It is sufficient to check two copies of C:

Theorem (informal)

$$check(C \cup C') \implies check(!C)$$

Symbolic model

- Messages are terms
  constructed using abstract cryptographic primitives,



- Symbolic attacker
  with abilities defined by deduction rules

$$\frac{\mathtt{enc}(x, y) \qquad y}{x} \qquad\qquad \frac{x \qquad y}{\langle x, y \rangle}$$

Symbolic model

Process algebra similar to the applied pi-calculus

$$
\begin{array}{rcl}
P, Q & ::= & \\
& & 0 \\
& | & \texttt{new}\ n.P \\
& | & \texttt{out}(M).P \\
& | & \texttt{in}(x).P \\
& | & P \mid Q \\
& | & \texttt{let}\ x = d(y)\ \texttt{in}\ P\ \texttt{else}\ Q \\
& | & \texttt{if}\ M = N\ \texttt{then}\ P\ \texttt{else}\ Q \\
& | & !P
\end{array}
$$

Static equivalence

Frames are sequences of messages modelling the attacker's knowledge

$$\phi = \{x_1 \mapsto k,\ x_2 \mapsto a,\ x_3 \mapsto \mathtt{enc}(b, k)\}$$

Static equivalence = indistinguishability of frames

$$\phi \approx \phi' \quad \Longleftrightarrow \quad \forall R, S.\ R\phi = S\phi \Leftrightarrow R\phi' = S\phi'$$

Example:

$$\{\mathtt{enc}(a, k)\} \quad \approx \quad \{\mathtt{enc}(b, k)\}$$

but

$$\{\mathtt{enc}(a, k), \mathtt{enc}(a, k)\} \quad \not\approx \quad \{\mathtt{enc}(a, k), \mathtt{enc}(b, k)\}$$

and

$$\{k, \mathtt{enc}(a, k)\} \quad \not\approx \quad \{k, \mathtt{enc}(b, k)\}$$

Trace equivalence

A trace (tr, ϕ) is a sequence of observable actions
+ a frame of messages sent on the network

Definition (Trace equivalence)

P and Q are trace equivalent if any trace of P can be mimicked by a trace of Q (and conversely)

i.e.

$$\forall (tr, \phi) \in \mathsf{trace}(P).\ \exists (tr, \phi') \in \mathsf{trace}(Q).\ \phi \approx \phi'$$

and

$$\forall (tr, \phi) \in \mathsf{trace}(Q).\ \exists (tr, \phi') \in \mathsf{trace}(P).\ \phi \approx \phi'$$

Typing messages

Types for messages :

$$
\begin{array}{lcl}
l & ::= & \mathtt{LL} \mid \mathtt{HL} \mid \mathtt{HH} \\
T & ::= & l \\
 & \mid & \mathsf{key}^{l}(T) \\
 & \mid & T * T \\
 & \mid & T \vee T \\
 & \mid & \ldots
\end{array}
$$

- labels = levels of confidentiality and integrity
  - LL for public messages
  - HH for secret values
- key types $\mathsf{key}^{l}(T)$
  Example:

$$\mathsf{key}^{\mathtt{HH}}(\mathtt{LL} * \mathtt{HH})$$

Typing messages

$$\frac{\Gamma \vdash M \sim N : T \qquad \Gamma(k) = key^{\mathtt{HH}}(T)}{\Gamma \vdash \mathtt{enc}(M, k) \sim \mathtt{enc}(N, k) : \mathtt{LL}}$$

Ensure the messages sent are safe to output:
$\longrightarrow$ similar structure

$$\langle a, b \rangle \not\sim a$$

$$\mathtt{enc}(\langle a, b \rangle, k) \sim \mathtt{enc}(a, k) \quad \text{only if } k \text{ is secret}$$

Typing messages

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma(k) = key^{\mathtt{HH}}(T)}{\Gamma \vdash \mathtt{enc}(M,k) \sim \mathtt{enc}(N,k) : \mathtt{LL} \to c \cup \{\mathtt{enc}(M,k) \sim \mathtt{enc}(N,k)\}}$$

- Establish invariants regarding the types of keys
  If k is secret, the type of M, N must match the type of k

- Collect constraints
  Here we add the couple $\mathtt{enc}(M,k) \sim \mathtt{enc}(N,k)$ to the constraint

Typing processes

- All output messages must be of type LL
- Their constraints are collected

$$\frac{\Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash P \sim Q : C}{\Gamma \vdash \mathtt{out}(M).P \sim \mathtt{out}(N).Q : C \cup c}$$

- All input messages are considered to be of type LL:

$$\frac{\Gamma, x : \mathtt{LL} \vdash P \sim Q : C}{\Gamma \vdash \mathtt{in}(x).P \sim \mathtt{in}(x).Q : C}$$

Typing processes (2)

$\longrightarrow$ Processes have to progress the same way:

accept inputs/outputs at the same time,
follow (typably) equivalent branches

Example: applying destructors

$$\frac{\Gamma(x) = \mathtt{LL} \qquad \Gamma(k) = \mathsf{key}^{\mathtt{HH}}(T) \qquad \Gamma, y : T \vdash P \sim Q : C \qquad \Gamma \vdash P' \sim Q' : C'}{\Gamma \vdash \mathtt{let}\ y = \mathtt{dec}(x, k)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ y = \mathtt{dec}(x, k)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' : C \cup C'}$$

Constraints

Why do we need constraints?

Constraints

Why do we need constraints?

$\longrightarrow$ Local checks on the messages are not sufficient for equivalence

Constraints

Why do we need constraints?

$\longrightarrow$ Local checks on the messages are not sufficient for equivalence

Example: If k is a secret key

$$\begin{array}{lcll} \mathtt{out}(\mathtt{enc}(a,k)) & \sim & \mathtt{out}(\mathtt{enc}(b,k)) & \text{is fine} \\ \mathtt{out}(\mathtt{enc}(a,k)) & \sim & \mathtt{out}(\mathtt{enc}(a,k)) & \text{is fine} \end{array}$$

but not both together

$$\mathtt{out}(\mathtt{enc}(a,k)) \mid \mathtt{out}(\mathtt{enc}(a,k)) \not\sim \mathtt{out}(\mathtt{enc}(b,k)) \mid \mathtt{out}(\mathtt{enc}(a,k))$$

Constraints

Why do we need constraints?

$\longrightarrow$ Local checks on the messages are not sufficient for equivalence

Example: If k is a secret key

$$\begin{array}{rcll} \mathtt{out}(\mathtt{enc}(a,k)) & \sim & \mathtt{out}(\mathtt{enc}(b,k)) & \text{is fine} \\ \mathtt{out}(\mathtt{enc}(a,k)) & \sim & \mathtt{out}(\mathtt{enc}(a,k)) & \text{is fine} \end{array}$$

but not both together

$$\mathtt{out}(\mathtt{enc}(a,k)) \mid \mathtt{out}(\mathtt{enc}(a,k)) \not\sim \mathtt{out}(\mathtt{enc}(b,k)) \mid \mathtt{out}(\mathtt{enc}(a,k))$$

$$\mathcal{C} = \{\mathtt{enc}(a,k) \sim \mathtt{enc}(b,k), \mathtt{enc}(a,k) \sim \mathtt{enc}(a,k)\}$$

Constraints

Collect symbolic messages in a constraint C while typing and check that it is consistent

i.e. for any possible instantiation, C instantiated does not leak anything:

$$C = \{u_1 \sim v_1, \ldots, u_n \sim v_n\}$$

must satisfy

$$\forall \theta, \theta'. \quad \{u_1\theta, \ldots, u_n\theta\} \approx \{v_1\theta', \ldots, v_n\theta'\}$$

$\longrightarrow$ Actually only consider well-typed θ, θ'

i.e.

$$\forall x. \vdash \theta(x) \sim \theta'(x) : \Gamma(x)$$

Experimental results

- Prototype implementation for our type system
- We implement a typechecker,
  together with the procedure for constraints
- Very efficient
- But requires some type annotations

Protocol	Akiss	Apte	Apte-POR	Spec	Sat-Eq	Deepsec	TypeEq
Denning-Sacco	10	6	12	7	>20	>20	>20
Wide Mouth Frog	14	7	12	7	>20	>20	>20
Needham-Schroeder Symmetric Key	10	6	10	6	>20	>20	>20
Yahalom-Lowe	10	6	10	7	>20	>20	>20
Otway-Rees	6	3	6	6	-	>20	>20

Number of sessions treated when proving secrecy
(bounded case)

Experimental results

Closer look for the Needham-Schroeder symmetric key protocol:

# sessions	Akiss	Apte	Apte-POR	Spec	Sat-Eq	Deepsec (on 40 cpu cores)	TypeEq
3	0.1s	0.4s	0.02s	52s	0.1s	0.16s	0.003s
6	20s	TO	4s	MO	0.2s	0.17s	0.003s
7	2m		8m		0.5s	0.17s	0.003s
10	SO		TO		0.9s	0.18s	0.005s
12					1.7s	0.2s	0.005s
14					2s	0.3s	0.007s
30					17s	8s	0.01s

Experimental results (unbounded)

We also compare to ProVerif for unbounded numbers of sessions:

Protocols	ProVerif	TypeEq
Helios	x	0.005s
Needham-Schroeder (sym)	0.23s	0.016s
Needham-Schroeder-Lowe	0.08s	0.008s
Yahalom-Lowe	0.48s	0.020s
Private Authentication	0.034s	0.008s
BAC	0.038s	0.005s

- Performances comparable to ProVerif for unbounded numbers of sessions
- First automated proof for Helios with unbounded number of sessions without private channels

Conclusion and future work

- a new approach to automatic proofs of equivalence properties for cryptographic protocols
- based on type systems + constraints
- handle bounded and unbounded numbers of sessions (CCS'17), dynamic keys, non uniform branching (POST'18)
- efficient implementation

Future work:

- type inference
- computational soundness
- composition