

Formal Proofs of Cryptographic Protocols with Squirrel

Part 2: Squirrel

EJCP 2026, Nantes

Adrien Koutsos Inria, Paris
JosephALLEmand CNRS, IRISA, Rennes

October 2026

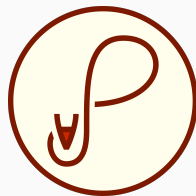


[Slides adapted from David Baelde's EJCP'25 presentation]

Introduction

What is Squirrel?

A **proof assistant** for
verifying cryptographic protocols,
based on the **CCSA approach**.



Bana & Comon. *A Computationally Complete Symbolic Attacker for Equivalence Properties*. CCS 2014.

Developed in collaborations by researchers in **Rennes**, **Paris** and **Nancy**.

This talk

An informal introduction to Squirrel:

- Preparing the ground for hands-on learning.
- How to formally model protocols and reason about their properties.
- Limited to trace properties: no equivalences.

I'm not going to talk about the theory, open problems, related works. . .

Demo: Basic Reasoning in Squirrel



0-logic.sp



Squirrel uses **standard proof assistant UI**, à la Rocq.

We prove formulas by organizing them in **sequents**:

$$\phi_1, \dots, \phi_n \vdash \psi \quad \text{reads as} \quad (\bigwedge_i \phi_i) \Rightarrow \psi$$

The concrete notation is shown on the right, with identifiers for hypotheses.

```
H_1 : phi_1
...
H_n : phi_n
-----
psi
```

Demo: Basic Reasoning in Squirrel



0-logic.sp



Squirrel uses **standard proof assistant UI**, à la Rocq.

We prove formulas by organizing them in **sequents**:

$$\phi_1, \dots, \phi_n \vdash \psi \quad \text{reads as} \quad (\bigwedge_i \phi_i) \Rightarrow \psi$$

The concrete notation is shown on the right, with identifiers for hypotheses.

```
H_1 : phi_1
...
H_n : phi_n
-----
psi
```

However the Squirrel logic is **not as standard as it seems**.

Reasoning about messages

- 1 Introduction
- 2 Reasoning about messages
 - Messages as terms
 - Modeling an interaction with the attacker
 - Cryptographic reasoning
 - Further notes
- 3 Reasoning about protocols
- 4 Conclusion

Modeling Messages

Crypto is all about probabilistic, polynomial-time (PPTIME) computations.

Key idea #1

Use **logical terms** to denote **PPTIME bitstring computations**.

Modeling Messages

 **language** (honest function symbols)

They represent **deterministic computations**.

Used to model primitives, public constants, etc.

Notation: $f(m)$, $g(m, n)$, $ok \dots$

Modeling Messages

 **language** (honest function symbols)

They represent **deterministic computations**.

Used to model primitives, public constants, etc.

Notation: $f(m)$, $g(m, n)$, $ok \dots$

We assume standard builtins: $equals$, if then $else$, etc.

Example

- $if\ u = v\ then\ (if\ v = u\ then\ t_1\ else\ t_2)\ else\ t_3$
 $if\ u = v\ then\ t_1\ else\ t_3$
(both terms always compute the same thing)

Modeling Messages

language (honest function symbols)

They represent **deterministic computations**.

Used to model primitives, public constants, etc.

Notation: $f(m)$, $g(m, n)$, $ok \dots$

We assume standard builtins: $equals$, if then $else$, etc.

Example

- $if\ u = v\ then\ (if\ v = u\ then\ t_1\ else\ t_2)\ else\ t_3$
 $if\ u = v\ then\ t_1\ else\ t_3$
(both terms always compute the same thing)
- g^x might denote a DH public key associated to private key x .
To model x , we need probabilistic symbols.

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k, \dots$

Example

- $\Pr(m = n)$ is negligible when m and n are distinct name symbols:

$$\Pr(m = n) \approx \frac{1}{2^\eta} \in \text{negl}(\eta).$$

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k \dots$

Example

- $\Pr(m = n)$ is negligible when m and n are distinct name symbols:

$$\Pr(m = n) \approx \frac{1}{2^\eta} \in \text{negl}(\eta).$$

- $\Pr(t = n)$ is negligible when t represents a computation that cannot use n .

$\rightsquigarrow$ This is guaranteed if t contains neither n nor variables.
Variables $x, y, z \dots$ represent *arbitrary* probabilistic computations.

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k \dots$

Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.
($f(\eta)$ is overwhelming iff $1 - f(\eta)$ is negligible)

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k, \dots$

Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.
($f(\eta)$ is overwhelming iff $1 - f(\eta)$ is negligible)

Example

- $m \neq n$ is valid.
- $t \neq n$ is valid whenever t contains neither n nor variables.

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k, \dots$

Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.
($f(\eta)$ is overwhelming iff $1 - f(\eta)$ is negligible)

Example

- $m \neq n$ is valid.
- $t \neq n$ is valid whenever t contains neither n nor variables.
- However, $(\forall x. x \neq n)$ is **not** valid.

Modeling Messages: Names

language (names)

Represent **independent uniform random samplings** of length $\approx \eta$.
Names model private keys, DH exponents, nonces, etc.

Notation: $n, r, k, \dots$

Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.
($f(\eta)$ is overwhelming iff $1 - f(\eta)$ is negligible)

Example

- $m \neq n$ is valid.
- $t \neq n$ is valid whenever t contains neither n nor variables.
- However, $(\forall x. x \neq n)$ is **not** valid.

 Demo: 1-names.sp  [with the **fresh** tactic (also for indexed names)]

Modeling Messages: Adversarial Function Symbols

Key idea #3

Unspecified function symbols models attacker computations.

Modeling Messages: Adversarial Function Symbols

 **language(adversarial function symbols)**

PPTIME computations that **cannot access honest randomness** (i.e. names).

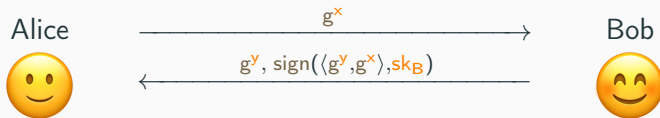
Notation: **att** $(m_1, \dots, m_k)$.

Modeling Messages: Adversarial Function Symbols

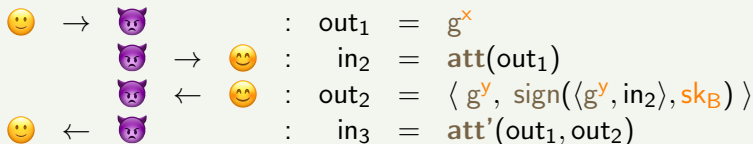
 language(adversarial function symbols)

PPTIME computations that **cannot access honest randomness** (i.e. names).

Notation: $\text{att}(m_1, \dots, m_k)$.



Example (Modelling a trace of signed DH protocol)



Key idea #4

Reformulate cryptographic assumptions as axiom schemes by **viewing terms as attacker computations**.

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$?

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)
- Can we have $\text{att}(g^a, g^b) = g^{a \cdot b}$?

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)
- Can we have $\text{att}(g^a, g^b) = g^{a \cdot b}$? **X**

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{ll} g : \text{message} & g^x : \text{message} \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x \quad \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)
- Can we have $\text{att}(g^a, g^b) = g^{a \cdot b}$? **X**
- The formula $t \neq g^{a \cdot b}$ is valid whenever...

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)
- Can we have $\text{att}(g^a, g^b) = g^{a \cdot b}$? **X**
- The formula $t \neq g^{a \cdot b}$ is valid whenever...
 t has no variable, only contains a as g^a , only contains b as g^b .

Cryptographic Reasoning: Diffie-Hellman

Axiom scheme: CDH

Consider a generator g , exponentiation, and exponent operations, interpreted in a cyclic group for which we assume **CDH**.

$$\begin{array}{lll} g : \text{message} & g^x : \text{message} & \\ (g^x)^y = g^{x \cdot y} & x \cdot y = y \cdot x & \dots \end{array}$$

Example

- Can we have $g^a = g^{a \cdot b}$? **X**
(or $\mathcal{A}(g^a, g^b) := g^a$ would break CDH)
- Can we have $\text{att}(g^a, g^b) = g^{a \cdot b}$? **X**
- The formula $t \neq g^{a \cdot b}$ is valid whenever...
 t has no variable, only contains a as g^a , only contains b as g^b .

 Demo: 2-cdh.sp  [with the **cdh** tactic (also for indexed secrets)]

 Demo: 2.5-cdh-signed-dh.sp 

Cryptographic Reasoning: Signatures

Axiom scheme: EUF-CMA (signatures)

Assume function symbols representing an **EUF-CMA** signature:

$$\begin{aligned} \text{sign}(m, k) &: \text{message} & \text{verify}(m, s, \text{pub}(k)) &: \text{bool} \\ \text{verify}(m, \text{sign}(m, k), \text{pub}(k)) &= \text{true} \end{aligned}$$

Cryptographic Reasoning: Signatures

Axiom scheme: EUF-CMA (signatures)

Assume function symbols representing an **EUF-CMA** signature:

$\text{sign}(m, k) : \text{message}$ $\text{verify}(m, s, \text{pub}(k)) : \text{bool}$
 $\text{verify}(m, \text{sign}(m, k), \text{pub}(k)) = \text{true}$

The following **axiom scheme** is valid:

$$\text{verify}(m, s, \text{pub}(k)) \Rightarrow \bigvee_{m' \in S} m = m'$$

when:

- $S = \{ m' \mid \text{sign}(m', k) \text{ occurs in } m, s \}$
- m, s are ground terms only containing k as $\text{pub}(k)$ and $\text{sign}(_, k)$

Cryptographic Reasoning: Signatures

Axiom scheme: EUF-CMA (signatures)

Assume function symbols representing an **EUF-CMA** signature:

$$\begin{aligned} \text{sign}(m, k) &: \text{message} & \text{verify}(m, s, \text{pub}(k)) &: \text{bool} \\ \text{verify}(m, \text{sign}(m, k), \text{pub}(k)) &= \text{true} \end{aligned}$$

The following **axiom scheme** is valid:

$$\text{verify}(m, s, \text{pub}(k)) \Rightarrow \bigvee_{m' \in S} m = m'$$

when:

- $S = \{ m' \mid \text{sign}(m', k) \text{ occurs in } m, s \}$
- m, s are ground terms only containing k as $\text{pub}(k)$ and $\text{sign}(_, k)$

In practice, the tactic (**euf** H) allows to reason on $H:\text{verify}(m, s, \text{pk})$ to deduce the above axioms and more.

Cryptographic Reasoning: Signatures

Axiom scheme: EUF-CMA (signatures)

Assume function symbols representing an **EUF-CMA** signature:

$$\begin{aligned} \text{sign}(m, k) &: \text{message} & \text{verify}(m, s, \text{pub}(k)) &: \text{bool} \\ \text{verify}(m, \text{sign}(m, k), \text{pub}(k)) &= \text{true} \end{aligned}$$

The following **axiom scheme** is valid:

$$\text{verify}(m, s, \text{pub}(k)) \Rightarrow \bigvee_{m' \in S} m = m'$$

when:

- $S = \{ m' \mid \text{sign}(m', k) \text{ occurs in } m, s \}$
- m, s are ground terms only containing k as $\text{pub}(k)$ and $\text{sign}(_, k)$

In practice, the tactic (**euf** H) allows to reason on $H:\text{verify}(m, s, pk)$ to deduce the above axioms and more.

Cryptographic Reasoning: Hash Functions

Axiom scheme: EUF-CMA (hashes)

Assume h models a keyed hash functions.

If h is **EUF-CMA** secure, the following **axiom scheme** is valid:

$$u = h(v, k) \Rightarrow \bigvee_{s \in S} s = v$$

when:

- $S = \{s \mid h(s, k) \text{ occurs in } u, v\}$
- u, v are variable-free terms where k only occurs as $h(_, k)$.

Cryptographic Reasoning: Hash Functions

Axiom scheme: EUF-CMA (hashes)

Assume h models a keyed hash functions.

If h is **EUF-CMA** secure, the following **axiom scheme** is valid:

$$u = h(v, k) \Rightarrow \bigvee_{s \in S} s = v$$

when:

- $S = \{s \mid h(s, k) \text{ occurs in } u, v\}$
- u, v are variable-free terms where k only occurs as $h(_, k)$.

Axiom scheme: Collision Resistance

If h is **collision-resistant**, the following **axiom scheme** is valid:

$$h(u, k) = h(v, k) \Rightarrow u = v$$

when u, v are ground terms where k only occurs as $h(_, k)$.

Cryptographic Reasoning: Hash Functions

Axiom scheme: EUF-CMA (hashes)

Assume h models a keyed hash functions.

If h is **EUF-CMA** secure, the following **axiom scheme** is valid:

$$u = h(v, k) \Rightarrow \bigvee_{s \in S} s = v$$

when:

- $S = \{s \mid h(s, k) \text{ occurs in } u, v\}$
- u, v are variable-free terms where k only occurs as $h(_, k)$.

Axiom scheme: Collision Resistance

If h is **collision-resistant**, the following **axiom scheme** is valid:

$$h(u, k) = h(v, k) \Rightarrow u = v$$

when u, v are ground terms where k only occurs as $h(_, k)$.

Existential unforgeability implies collision-resistance,
but the **collision** tactic is more convenient than **euf**.

What's in the full local logic?

- Actually, some terms do not represent PPTIME computations.
For example, $(\forall x : \text{message}. \phi)$ asks to evaluate ϕ infinitely often.
 $\rightsquigarrow$ Axioms schemes can be refined to account for that.

What's in the full local logic?

- Actually, some terms do not represent PPTIME computations.
For example, $(\forall x : \text{message}. \phi)$ asks to evaluate ϕ infinitely often.
 $\rightsquigarrow$ Axioms schemes can be refined to account for that.
- Function types, polymorphism, higher-order quantification...
for modularity/abstraction in proofs. Recursive definitions (next).

What's in the full local logic?

- Actually, some terms do not represent PPTIME computations.
For example, $(\forall x : \text{message}. \phi)$ asks to evaluate ϕ infinitely often.
 $\rightsquigarrow$ Axioms schemes can be refined to account for that.
- Function types, polymorphism, higher-order quantification...
for modularity/abstraction in proofs. Recursive definitions (next).

Support for proof automation using SMTs (Stanislas Riou's PhD).

Further Notes

What's in the full local logic?

- Actually, some terms do not represent PPTIME computations. For example, $(\forall x : \text{message}. \phi)$ asks to evaluate ϕ infinitely often.
 $\rightsquigarrow$ Axioms schemes can be refined to account for that.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Support for proof automation using SMTs (Stanislas Riou's PhD).

What's beyond the local logic?

The **global** logic is a classical logic over random vars with predicates:

- $[\phi]$: overwhelming truth of a local formula,
- $u_1, \dots, u_n \sim v_1, \dots, v_n$: computational indistinguishability,
- exact truth of a local formula, (non/bi)-deducibility, etc.

Further Notes

What's in the full local logic?

- Actually, some terms do not represent PPTIME computations. For example, $(\forall x : \text{message}. \phi)$ asks to evaluate ϕ infinitely often.
 $\rightsquigarrow$ Axioms schemes can be refined to account for that.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Support for proof automation using SMTs (Stanislas Riou's PhD).

What's beyond the local logic?

The **global** logic is a classical logic over random vars with predicates:

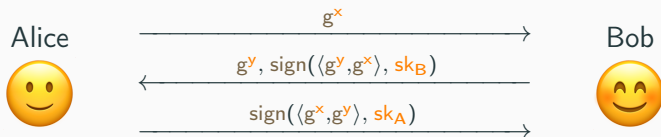
- $[\phi]$: overwhelming truth of a local formula,
- $u_1, \dots, u_n \sim v_1, \dots, v_n$: computational indistinguishability,
- exact truth of a local formula, (non/bi)-deducibility, etc.

Crypto tactics are begin subsumed by bideduction (Justine Sauvage's PhD).

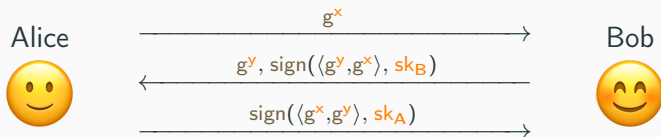
Reasoning about protocols

- 1 Introduction
- 2 Reasoning about messages
- 3 Reasoning about protocols**
 - Systems of actions
 - Protocol semantics along a trace
 - Reasoning with recursive definitions
 - Further notes
- 4 Conclusion

Systems of Actions: Example (Signed DH)

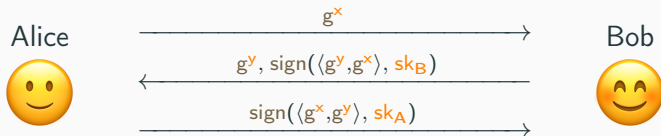


Systems of Actions: Example (Signed DH)



```
(* Alice 😊 *)  
process A (skA, pkB) =  
  new x : exponent;  
  let X = gx in  
  out(X);  
  in(m);  
  let Y' = fst(m) in  
  let s = snd(m) in  
  if verify(⟨Y', X⟩, s, pkB) then  
    out(sign(⟨X, Y'⟩, skA))  
  else  
    out(👎)
```

Systems of Actions: Example (Signed DH)



(* Alice 😊 *)

```
process A (skA, pkB) =  
  new x : exponent;  
  let X = gx in  
  out(X);  
  
  in(m);  
  let Y' = fst(m) in  
  let s = snd(m) in  
  if verify(⟨Y', X⟩, s, pkB) then  
    out(sign(⟨X, Y'⟩, skA))  
  else  
    out(👎)
```

(* Bob 😊 *)

```
process B(skB, pkA) =  
  new y : exponent;  
  let Y = gy in  
  in(x');  
  let X' = x' in  
  out(⟨Y, sign(⟨Y, X'⟩, skB)⟩);  
  
  in(s);  
  if verify(⟨X', Y⟩, s, pkA) then  
    out(👍)  
  else  
    out(👎)
```

Systems of Actions

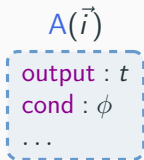
A protocol is modeled by a set of **actions**.

Each action is identified by an indexed action symbol $A(\vec{i})$.

The semantics of action $A(\vec{i})$ is given by:

- a local formula ϕ describing the executability **condition**;
- a **message** term t describing its **output**;
- and possibly some additional values (from **let**-definitions).

Both can use a special **message** term $\text{input}@A(\vec{i})$.



Systems of Actions: Example (Signed DH)

(* Alice 😊 *)

process A (sk_A, pk_B) =

new x : exponent;

let $X = g^x$ **in**

out(X);

in(m);

let $Y' = \text{fst}(m)$ **in**

let $s = \text{snd}(m)$ **in**

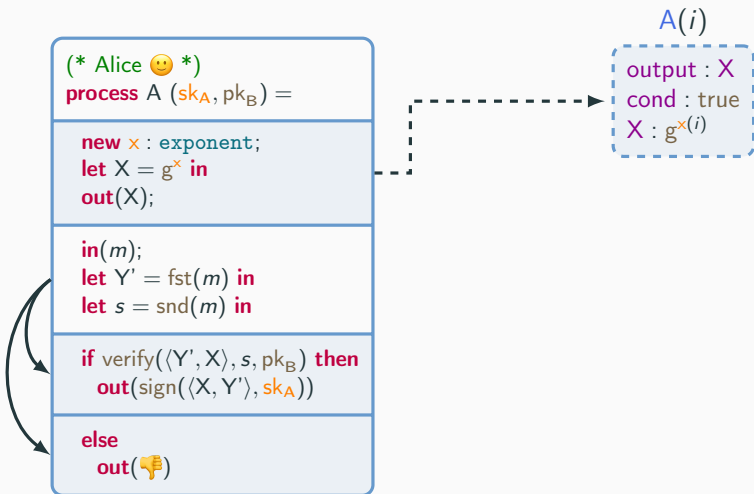
if $\text{verify}(\langle Y', X \rangle, s, pk_B)$ **then**

out($\text{sign}(\langle X, Y' \rangle, sk_A)$)

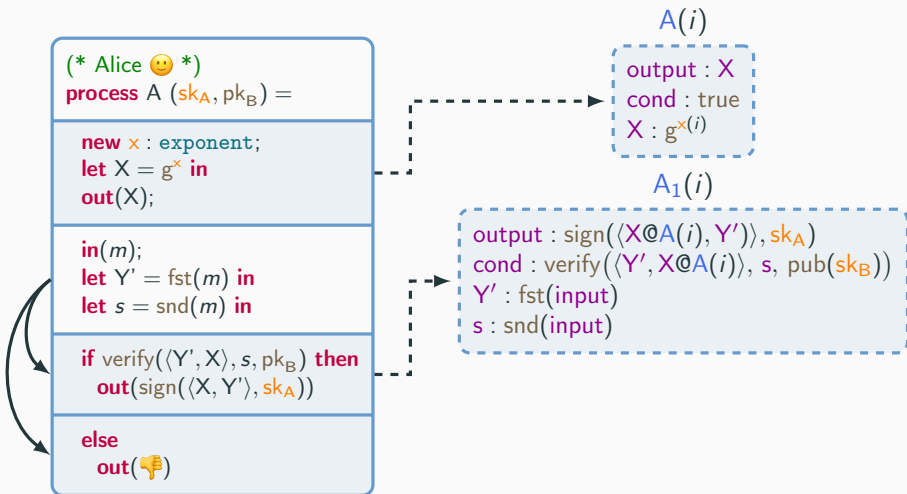
else

out(👎)

Systems of Actions: Example (Signed DH)

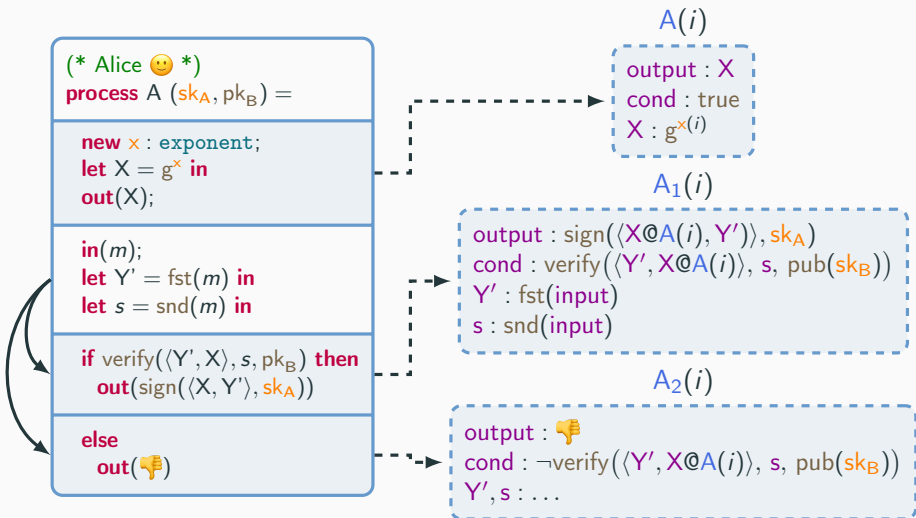


Systems of Actions: Example (Signed DH)



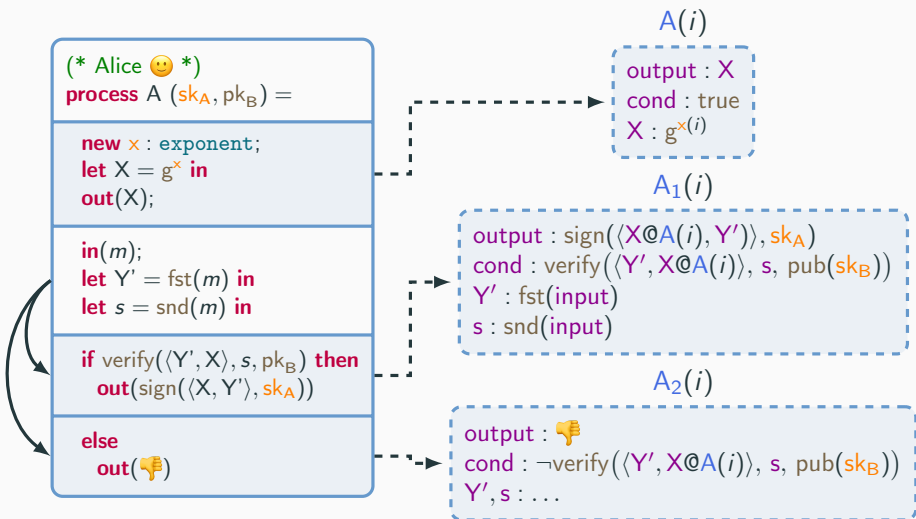
Convention: In an action $C(i)$, a macro X is sugar for $X@C(i)$.

Systems of Actions: Example (Signed DH)



Convention: In an action $C(i)$, a macro X is sugar for $X@C(i)$.

Systems of Actions: Example (Signed DH)



Convention: In an action $C(i)$, a macro X is sugar for $X@C(i)$.

🕒 Demo: 3.5-signed-dh-many.sp 🕒 [actions compiled from π -processes]

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- $A(1).B(7).A(1)$
- $A(1).B(7).A_1(1)$
- $A(1).B(7).A_1(3)$
- $A(1).B(7).A_2(1)$
- $A(1).B(7).A_1(1).A_2(1)$
- $B(7).A(1).A_1(1)$

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | |
|----------------------|---|-----------------------------|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ |
| ■ $A(1).B(7).A_1(3)$ | | ■ $B(7).A(1).A_1(1)$ |

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | |
|----------------------|---|-----------------------------|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ |
| ■ $A(1).B(7).A_1(3)$ | ✗ | ■ $B(7).A(1).A_1(1)$ |

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | | |
|----------------------|---|-----------------------------|---|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ | ✓ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ | |
| ■ $A(1).B(7).A_1(3)$ | ✗ | ■ $B(7).A(1).A_1(1)$ | |

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | | |
|----------------------|---|-----------------------------|---|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ | ✓ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ | ✗ |
| ■ $A(1).B(7).A_1(3)$ | ✗ | ■ $B(7).A(1).A_1(1)$ | |

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | | |
|----------------------|---|-----------------------------|---|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ | ✓ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ | ✗ |
| ■ $A(1).B(7).A_1(3)$ | ✗ | ■ $B(7).A(1).A_1(1)$ | ✓ |

Traces of Actions

A **trace** is a non-repeating sequence of actions $\alpha_1 \dots \alpha_n$ subject to protocol-specific conditions.

Example Signed DH

The actions of the Signed DH protocol are:

$\text{init} \mid A(i) \mid A_1(i) \mid A_2(i) \mid B(i) \mid B_1(i) \mid B_2(i)$



A few examples of Signed DH traces:

- | | | | |
|----------------------|---|-----------------------------|---|
| ■ $A(1).B(7).A(1)$ | ✗ | ■ $A(1).B(7).A_2(1)$ | ✓ |
| ■ $A(1).B(7).A_1(1)$ | ✓ | ■ $A(1).B(7).A_1(1).A_2(1)$ | ✗ |
| ■ $A(1).B(7).A_1(3)$ | ✗ | ■ $B(7).A(1).A_1(1)$ | ✓ |

A trace indicates a **tentative schedule** for actions.

↪ Its **execution probability** depends on the primitives/attackers.

Modeling Traces in the Logic

We use type `timestamp` which is (almost) an algebraic data-type:

$$\text{init} \mid A(\vec{i}) \mid B(\vec{j}) \mid \dots$$

- $\vec{i}, \vec{j}$ are the **indices** (of type `index`);
- A, B are **action constructors** in $\mathcal{A}$.

Modeling Traces in the Logic

We use type `timestamp` which is (almost) an algebraic data-type:

$$\text{init} \mid A(\vec{i}) \mid B(\vec{j}) \mid \dots$$

- $\vec{i}, \vec{j}$ are the **indices** (of type `index`);
- A, B are **action constructors** in $\mathcal{A}$.

We equip `timestamp` with the function symbols:

- `happens`(τ) stating that τ is part of the trace;
- a total order $<$ on timestamps that happen.

$$\begin{array}{ccc} \text{trace} & \approx & \text{trace model} \\ \text{(sequence of timestamps)} & & \left(\begin{array}{l} \text{interpretation for} \\ \text{actions, happens, } < \end{array} \right) \end{array}$$

Modeling Traces in the Logic: Properties

Injectivity (part of `auto` tactic)

For distinct action constructors $A, B \in \mathcal{A}$:

- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(B(\vec{j})) \Rightarrow A(\vec{i}) \neq B(\vec{j})$
- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(A(\vec{j})) \wedge A(\vec{i}) = A(\vec{j}) \Rightarrow \vec{i} = \vec{j}$

Modeling Traces in the Logic: Properties

Injectivity (part of `auto` tactic)

For distinct action constructors $A, B \in \mathcal{A}$:

- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(B(\vec{j})) \Rightarrow A(\vec{i}) \neq B(\vec{j})$
- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(A(\vec{j})) \wedge A(\vec{i}) = A(\vec{j}) \Rightarrow \vec{i} = \vec{j}$

Order (part of `auto` tactic)

- $\text{happens}(\tau) \wedge \text{happens}(\tau') \Leftrightarrow \tau \leq \tau' \vee \tau' \leq \tau$
- $\text{happens}(\text{pred}(\tau)) \Rightarrow \text{pred}(\tau) < \tau$

Modeling Traces in the Logic: Properties

Injectivity (part of `auto` tactic)

For distinct action constructors $A, B \in \mathcal{A}$:

- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(B(\vec{j})) \Rightarrow A(\vec{i}) \neq B(\vec{j})$
- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(A(\vec{j})) \wedge A(\vec{i}) = A(\vec{j}) \Rightarrow \vec{i} = \vec{j}$

Order (part of `auto` tactic)

- $\text{happens}(\tau) \wedge \text{happens}(\tau') \Leftrightarrow \tau \leq \tau' \vee \tau' \leq \tau$
- $\text{happens}(\text{pred}(\tau)) \Rightarrow \text{pred}(\tau) < \tau$

Case analysis and induction (`case` and `induction`)

- $\forall \tau. \text{happens}(\tau) \Rightarrow \tau = \text{init} \vee \bigvee_{A \in \mathcal{A}} \exists \vec{i}. \tau = A(\vec{i})$
- $(\forall \tau. (\forall \tau'. \tau' < \tau \Rightarrow \phi[\tau']) \Rightarrow \phi[\tau]) \Rightarrow \forall \tau. \phi[\tau]$

Modeling Traces in the Logic: Properties

Injectivity (part of `auto` tactic)

For distinct action constructors $A, B \in \mathcal{A}$:

- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(B(\vec{j})) \Rightarrow A(\vec{i}) \neq B(\vec{j}) \quad (\dagger)$
- $\forall \vec{i}, \vec{j}. \text{happens}(A(\vec{i})) \wedge \text{happens}(A(\vec{j})) \wedge A(\vec{i}) = A(\vec{j}) \Rightarrow \vec{i} = \vec{j} \quad (\dagger)$

Order (part of `auto` tactic)

- $\text{happens}(\tau) \wedge \text{happens}(\tau') \Leftrightarrow \tau \leq \tau' \vee \tau' \leq \tau$
- $\text{happens}(\text{pred}(\tau)) \Rightarrow \text{pred}(\tau) < \tau$

Case analysis and induction (`case` and `induction`)

- $\forall \tau. \text{happens}(\tau) \Rightarrow \tau = \text{init} \vee \bigvee_{A \in \mathcal{A}} \exists \vec{i}. \tau = A(\vec{i}) \quad (\dagger)$
- $(\forall \tau. (\forall \tau'. \tau' < \tau \Rightarrow \phi[\tau']) \Rightarrow \phi[\tau]) \Rightarrow \forall \tau. \phi[\tau]$

Remark $(\dagger)$: `timestamp` is almost an algebraic data-type, up-to `happens`.

Modeling Traces in the Logic: Properties

Signed DH specific axioms (used by `smt` but not `auto`)

- $\forall i. \text{happens}(A_1(i)) \Rightarrow A(i) < A_1(i)$ (dependency)
- $\forall i. \neg(\text{happens}(A_1(i)) \wedge \text{happens}(A_2(i)))$ (conflict)
- ...

Given a trace we define recursively several **macros** encoding the attacker's interaction with the protocol along that trace:

- **input**@ τ : input at time τ
- **output**@ τ : output at time τ
- **cond**@ τ : condition at time τ
- **exec**@ τ : accumulated conditions up to time τ
- **frame**@ τ : adversary's knowledge at time τ

Roughly, this is all **output**@ τ_0 for any $\tau_0 \leq \tau$.

Macros

More precisely, **macros** are defined as follows:¹

output@ τ = $\langle \text{output of action } \tau \rangle$ if **init** < τ

cond@ τ = $\langle \text{condition of action } \tau \rangle$ if **init** < τ

¹Missing cases are not important.

Macros

More precisely, **macros** are defined as follows:¹

$\text{output@}\tau$ = $\langle \text{output of action } \tau \rangle$ if $\text{init} < \tau$

$\text{cond@}\tau$ = $\langle \text{condition of action } \tau \rangle$ if $\text{init} < \tau$

exec@init = true

$\text{exec@}\tau$ = $\text{exec@pred}(\tau) \wedge \text{cond@}\tau$ if $\text{init} < \tau$

¹Missing cases are not important.

Macros

More precisely, **macros** are defined as follows:¹

$\text{output@}\tau$ = $\langle \text{output of action } \tau \rangle$ if $\text{init} < \tau$

$\text{cond@}\tau$ = $\langle \text{condition of action } \tau \rangle$ if $\text{init} < \tau$

exec@init = true

$\text{exec@}\tau$ = $\text{exec@pred}(\tau) \wedge \text{cond@}\tau$ if $\text{init} < \tau$

frame@init = empty

$\text{frame@}\tau$ = $\langle \text{frame@pred}(\tau),$
 $\langle \text{exec@}\tau,$
 if $\text{exec@}\tau$ then $\text{output@}\tau$ else empty $\rangle \rangle$ if $\text{init} < \tau$

¹Missing cases are not important.

Macros

More precisely, **macros** are defined as follows:¹

$\text{output@}\tau$ = $\langle \text{output of action } \tau \rangle$ if $\text{init} < \tau$

$\text{cond@}\tau$ = $\langle \text{condition of action } \tau \rangle$ if $\text{init} < \tau$

exec@init = true

$\text{exec@}\tau$ = $\text{exec@pred}(\tau) \wedge \text{cond@}\tau$ if $\text{init} < \tau$

frame@init = empty

$\text{frame@}\tau$ = $\langle \text{frame@pred}(\tau),$
 $\langle \text{exec@}\tau,$
 if $\text{exec@}\tau$ then $\text{output@}\tau$ else empty $\rangle \rangle$ if $\text{init} < \tau$

$\text{input@}\tau$ = $\text{att}(\text{frame@pred}(\tau))$ if $\text{init} < \tau$

¹Missing cases are not important.

Macros

More precisely, **macros** are defined as follows:¹

$\text{output@}\tau$ = $\langle \text{output of action } \tau \rangle$ if $\text{init} < \tau$

$\text{cond@}\tau$ = $\langle \text{condition of action } \tau \rangle$ if $\text{init} < \tau$

exec@init = true

$\text{exec@}\tau$ = $\text{exec@pred}(\tau) \wedge \text{cond@}\tau$ if $\text{init} < \tau$

frame@init = empty

$\text{frame@}\tau$ = $\langle \text{frame@pred}(\tau),$
 $\langle \text{exec@}\tau,$
 if $\text{exec@}\tau$ then $\text{output@}\tau$ else empty $\rangle \rangle$ if $\text{init} < \tau$

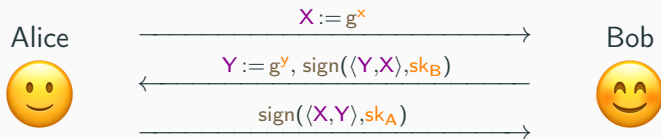
$\text{input@}\tau$ = $\text{att}(\text{frame@pred}(\tau))$ if $\text{init} < \tau$

Additional macros reflect **let**-definitions used in processes.

For example $Y'@A_1(i)$ is the value of Y' in that action.

¹Missing cases are not important.

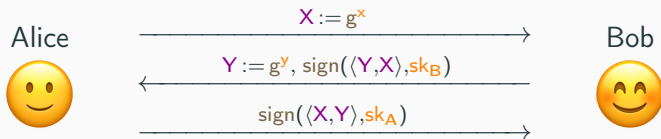
Security Properties for All Traces



Example (Agreement for)

$$\begin{aligned} \forall i. \text{cond}@A_1(i) \Rightarrow \exists j. B(j) < A_1(i) \wedge \\ X@A(i) = X'@B(j) \wedge \\ Y'@A_1(i) = Y@B(j) \end{aligned}$$

Security Properties for All Traces



Example (Agreement for)

$$\begin{aligned} \forall i. \text{cond}@A_1(i) \Rightarrow \exists j. B(j) < A_1(i) \wedge \\ X@A(i) = X'@B(j) \wedge \\ Y'@A_1(i) = Y@B(j) \end{aligned}$$

 Demo: 3.5-signed-dh-many.sp 

Reasoning with Recursive Definitions

Constraining occurrences becomes more complex with macros.

Example (Freshness without macros nor indices)

$t \neq n$

(t ground and does not contain n)

Reasoning with Recursive Definitions

Constraining occurrences becomes more complex with macros.

Example (Freshness without macros nor indices)

$t \neq n$ (t ground and does not contain n)

Example (Freshness without macros)

$t = n(\vec{i}) \Rightarrow \forall_{n(\vec{j}) \in t} \vec{i} = \vec{j}$ (t without message variables)

Reasoning with Recursive Definitions

Constraining occurrences becomes more complex with macros.

Example (Freshness without macros nor indices)

$$t \neq n \quad (t \text{ ground and does not contain } n)$$

Example (Freshness without macros)

$$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j} \quad (t \text{ without message variables})$$

Example (Freshness with macros)

$$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j} \vee \bigvee_{n(\vec{j}) \in A(\vec{k})} \exists \vec{k}. A(\vec{k}) \leq T \wedge \vec{i} = \vec{j}$$

for t without message variables,
 $n(\vec{j}) \in A(\vec{k})$ denotes occurrences in **out**, **cond** or **let**-definitions of A .

Reasoning with Recursive Definitions

Constraining occurrences becomes more complex with macros.

Example (Freshness without macros nor indices)

$$t \neq n \quad (t \text{ ground and does not contain } n)$$

Example (Freshness without macros)

$$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j} \quad (t \text{ without message variables})$$

Example (Freshness with macros)

$$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j} \vee \bigvee_{n(\vec{j}) \in A(\vec{k})} \exists \vec{k}. A(\vec{k}) \leq T \wedge \vec{i} = \vec{j}$$

for t without message variables,
 $n(\vec{j}) \in A(\vec{k})$ denotes occurrences in **out**, **cond** or **let**-definitions of A .

Further refinements are possible and even necessary in practice.

Systems of Actions: Example (Signed DH)

$A(i)$

output : X
cond : true
 $X : g^{x(i)}$

$A_1(i)$

output : $\text{sign}(\langle g^{x(i)}, Y' \rangle, sk_A)$
cond : $\text{verify}(\langle Y', g^{x(i)} \rangle, s, pk_B)$
 $Y' : \text{fst}(\text{input})$
 $s : \text{snd}(\text{input})$

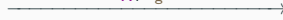
$A_2(i)$

output : 👎
cond : $\neg \text{verify}(\langle Y', g^{x(i)} \rangle, s, pk_B)$
 $Y', s : \dots$

Alice



$X := g^x$



$Y := g^y, \text{sign}(\langle Y, X \rangle, sk_B)$



$\text{sign}(\langle X, Y \rangle, sk_A)$



Bob



Example (Signed DH)

$\text{att}(\text{frame@}\tau) = x(j) \Rightarrow$

Systems of Actions: Example (Signed DH)

$A(i)$

```

output : X
cond   : true
X      : gx(i)
    
```

$A_1(i)$

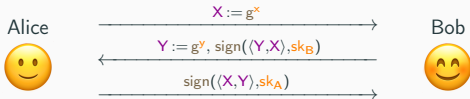
```

output : sign(⟨gx(i), Y'⟩, skA)
cond   : verify(⟨Y', gx(i)⟩, s, pkB)
Y'     : fst(input)
s      : snd(input)
    
```

$A_2(i)$

```

output : 👎
cond   : ¬verify(⟨Y', gx(i)⟩, s, pkB)
Y', s : ...
    
```



Example (Signed DH)

$$\text{att}(\text{frame@}\tau) = x(j) \Rightarrow$$

$$\begin{aligned} & \exists i. A(i) \leq \tau \wedge j = i \\ \vee & \exists i. A_1(i) \leq \tau \wedge j = i \\ \vee & \exists i. A_2(i) \leq \tau \wedge j = i \end{aligned}$$

Systems of Actions: Example (Signed DH)

$A(i)$

output : X
cond : true
 $X : g^{x(i)}$

$A_1(i)$

output : $\text{sign}(\langle g^{x(i)}, Y' \rangle, sk_A)$
cond : $\text{verify}(\langle Y', g^{x(i)} \rangle, s, pk_B)$
 $Y' : \text{fst}(\text{input})$
 $s : \text{snd}(\text{input})$

$A_2(i)$

output : 🙅
cond : $\neg \text{verify}(\langle Y', g^{x(i)} \rangle, s, pk_B)$
 $Y', s : \dots$

Alice



$X := g^x$

$Y := g^y, \text{sign}(\langle Y, X \rangle, sk_B)$

$\text{sign}(\langle X, Y \rangle, sk_A)$

Bob



Example (Signed DH)

$$\begin{aligned} \text{att}(\text{frame@}\tau) = x(j) &\Rightarrow \\ &\exists i. A(i) \leq \tau \wedge j = i \\ &\vee \exists i. A_1(i) \leq \tau \wedge j = i \\ &\vee \exists i. A_2(i) \leq \tau \wedge j = i \end{aligned}$$

or after simplifications:

$$\text{att}(\text{frame@}\tau) = x(j) \Rightarrow A(j) \leq \tau$$

Protocols with mutable state

Stateful protocols are supported (using `cell@ $\tau$` macros).

The translation from processes to systems of actions, and its soundness, is recent work notably involving Clément Herouard.

Protocols with mutable state

Stateful protocols are supported (using `cell@ $\tau$` macros).

The translation from processes to systems of actions, and its soundness, is recent work notably involving Clément Herouard.

Polynomial security

A subtle discrepancy between security notions:

- We prove that, for each trace T , there is no attacker along T .
- We would like to prove that there is no attacker, choosing the trace depending on η and previous messages.

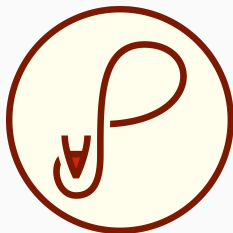
This is the topic of Théo Vignon's PhD.

Conclusion

What's next?

Hands on experience in practical sessions!

<https://gitlab.inria.fr/akoutsos1/ejcp-2026/>



Learn some more on our website (documentation, more tutorials, papers):

<https://squirrel-prover.github.io/>