

# Proving Cryptographic Protocols with Squirrel

## Part 2: Squirrel

David Baelde & Joseph Lallemand  
ENS Rennes, IRISA



# 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 by a group of 7 permanent researchers and 4 PhD students  
in Rennes, Paris and Nancy.

# This talk

An informal introduction to the Squirrel system:

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



# This talk

An informal introduction to the Squirrel system:

- 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

Proving basic logical facts in Squirrel:



0-logic.sp



Squirrel uses **standard proof assistant UI**, and is inspired by Coq.  
We prove formulas by organizing them in *sequents*:

$\phi_1, \dots, \phi_n \vdash \psi$       reads as       $(\wedge_i \phi_i) \Rightarrow \psi$

The concrete notation is as follows, with identifiers for hypotheses:

H\_1 : phi\_1

...

H\_n : phi\_n

-----

psi

## Demo

Proving basic logical facts in Squirrel:



0-logic.sp



Squirrel uses **standard proof assistant UI**, and is inspired by Coq.  
We prove formulas by organizing them in *sequents*:

$\phi_1, \dots, \phi_n \vdash \psi$       reads as       $(\wedge_i \phi_i) \Rightarrow \psi$

The concrete notation is as follows, with identifiers for hypotheses:

H\_1 : phi\_1

...

H\_n : phi\_n

-----

psi

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

# Outline

1 Introduction

2 Reasoning about messages

- Messages as terms
- Modelling an interaction with the attacker
- Cryptographic reasoning
- Further notes

3 Reasoning about protocols

4 Conclusion

## Modelling messages

Crypto is all about probabilistic, polynomial-time (PPTIME) computations.  
Reasoning about these directly is intimidating.

### Key idea #1

**Let's use logical terms to denote PPTIME bitstring computations.**

Honest function symbols are interpreted as deterministic computations,  
used to represent primitives, public constants, etc.

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

We assume builtins with standard semantics: `equals`, `ifthenelse`, etc.

### Example

- $\text{if } u = v \text{ then } (\text{if } v = u \text{ then } t_1 \text{ else } t_2) \text{ else } t_3$  and  
 $\text{if } u = v \text{ then } t_1 \text{ else } t_3$  always compute the same thing.

## Modelling messages

Crypto is all about probabilistic, polynomial-time (PPTIME) computations.  
Reasoning about these directly is intimidating.

### Key idea #1

**Let's use logical terms to denote PPTIME bitstring computations.**

Honest function symbols are interpreted as deterministic computations,  
used to represent primitives, public constants, etc.

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

We assume builtins with standard semantics: `equals`, `ifthenelse`, etc.

### Example

- if  $u = v$  then (if  $v = u$  then  $t_1$  else  $t_2$ ) else  $t_3$  and  
if  $u = v$  then  $t_1$  else  $t_3$  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.

# Modelling messages: names

## Names

Interpreted as independent uniform random samplings of length  $\approx \eta$ .

Notation:  $n, r, k\dots$

Names are used to model private keys, DH exponents, nonces, etc.

## Example

- When  $m$  and  $n$  are distinct name symbols,  
there is a negligible probability that  $m$  and  $n$  yield the same result.

# Modelling messages: names

## Names

Interpreted as independent uniform random samplings of length  $\approx \eta$ .

Notation:  $n, r, k\dots$

Names are used to model private keys, DH exponents, nonces, etc.

## Example

- When  $m$  and  $n$  are distinct name symbols, there is a negligible probability that  $m$  and  $n$  yield the same result.
- There is a negligible probability that  $t = n$  returns **true**, provided that  $t$  represents a computation that cannot use  $n$ .
  - ~~ This is guaranteed if  $t$  contains neither  $n$  nor variables.  
Variables  $x, y, z\dots$  represent *arbitrary* probabilistic computations.

# Modelling messages: names

## Names

Interpreted as independent uniform random samplings of length  $\approx \eta$ .

Notation:  $n, r, k\dots$

Names are used to model private keys, DH exponents, nonces, etc.

## Key idea #2

A formula is **valid** when it is true with overwhelming probability.

## Example

- The formula  $n \neq m$  is valid.
- The formula  $t \neq n$  is valid for any  $t$  containing neither  $n$  nor variables.

# Modelling messages: names

## Names

Interpreted as independent uniform random samplings of length  $\approx \eta$ .

Notation:  $n, r, k\dots$

Names are used to model private keys, DH exponents, nonces, etc.

## Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.

## Example

- The formula  $n \neq m$  is valid.
- The formula  $t \neq n$  is valid for any  $t$  containing neither  $n$  nor variables.
- However, the formula  $(\forall x. x \neq n)$  is **not** valid.

# Modelling messages: names

## Names

Interpreted as independent uniform random samplings of length  $\approx \eta$ .

Notation:  $n, r, k\dots$

Names are used to model private keys, DH exponents, nonces, etc.

## Key idea #2

A formula is **valid** when it is **true with overwhelming probability**.

## Example

- The formula  $n \neq m$  is valid.
- The formula  $t \neq n$  is valid for any  $t$  containing neither  $n$  nor variables.
- However, the formula  $(\forall x. x \neq n)$  is **not** valid.

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

# Modelling messages: adversarial function symbols

Key idea #3

Use unspecified function symbols to model attacker computations.

Adversarial function symbols represent PPTIME computations that cannot access honest randomness (names).

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

Example (Modelling a trace of signed DH protocol)

	$\rightarrow$		:	$\text{out}_1 = g^x$
	$\rightarrow$		:	$\text{in}_2 = \text{att}(\text{out}_1)$
	$\leftarrow$		:	$\text{out}_2 = \langle g^y, \text{sign}(\langle g^y, \text{in}_2 \rangle, \text{sk} \smiley) \rangle$
	$\leftarrow$		:	$\text{in}_3 = \text{att}'(\text{out}_1, \text{out}_2)$

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

## Example

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

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

## Example

- Can we have  $g^a = g^{a \times b}$ ? No.

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

### Example

- Can we have  $g^a = g^{a \times b}$ ? No.
- Can we have  $g^{a \times b} = \text{att}(g^a, g^b)$ ?

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

### Example

- Can we have  $g^a = g^{a \times b}$ ? No.
- Can we have  $g^{a \times b} = \text{att}(g^a, g^b)$ ? No.

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

### Example

- Can we have  $g^a = g^{a \times b}$ ? No.
- Can we have  $g^{a \times b} = \text{att}(g^a, g^b)$ ? No.
- The formula  $g^{a \times b} \neq t$  is valid whenever...

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

### Example

- Can we have  $g^a = g^{a \times b}$ ? No.
- Can we have  $g^{a \times b} = \text{att}(g^a, g^b)$ ? No.
- The formula  $g^{a \times b} \neq t$  is valid whenever...  
 $t$  contains no variable, only contains  $a$  as  $g^a$ , only contains  $b$  as  $g^b$ .

# Cryptographic reasoning: Diffie-Hellman

## Key idea #4

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

Assume function symbols for a generator  $g$  and exponentiation,  
interpreted in a cyclic group for which we assume CDH.

### Example

- Can we have  $g^a = g^{a \times b}$ ? No.
- Can we have  $g^{a \times b} = \text{att}(g^a, g^b)$ ? No.
- The formula  $g^{a \times b} \neq t$  is valid whenever...  
 $t$  contains 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-cdh-sdh.sp`

## Cryptographic reasoning: signatures

Assume function symbols representing an EUF-CMA signature:

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

$$\text{verify}(m, \text{sign}(m, k), \text{pub}(k)) = \text{true}$$

## Cryptographic reasoning: signatures

Assume function symbols representing an EUF-CMA signature:

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

$$\text{verify}(m, \text{sign}(m, k), \text{pub}(k)) = \text{true}$$

The following axiom scheme is valid:

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

where  $S = \{ m' \mid \text{sign}(m', k) \text{ occurs in } m, s \}$

and  $m, s$  are closed terms only containing  $k$  as  $\text{pub}(k)$  and  $\text{sign}(\_, k)$ .

## Cryptographic reasoning: signatures

Assume function symbols representing an EUF-CMA signature:

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

$$\text{verify}(m, \text{sign}(m, k), \text{pub}(k)) = \text{true}$$

The following axiom scheme is valid:

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

where  $S = \{ m' \mid \text{sign}(m', k) \text{ occurs in } m, s \}$

and  $m, s$  are closed 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.

Demo: `2-cdh-sdh.sp`

## Cryptographic reasoning: hash functions

Assume  $h$  models a keyed hash functions.

## Cryptographic reasoning: hash functions

Assume  $\mathbf{h}$  models a keyed hash functions.

If  $\mathbf{h}$  is EUF-CMA secure, we have

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

where  $S = \{ s \mid \mathbf{h}(s, \mathbf{k}) \text{ occurs in } u, v \}$

and  $u, v$  are variable-free terms where  $\mathbf{k}$  only occurs as  $\mathbf{h}(\_, \mathbf{k})$ .

## Cryptographic reasoning: hash functions

Assume  $\mathbf{h}$  models a keyed hash functions.

If  $\mathbf{h}$  is EUF-CMA secure, we have

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

where  $S = \{ s \mid \mathbf{h}(s, \mathbf{k}) \text{ occurs in } u, v \}$

and  $u, v$  are variable-free terms where  $\mathbf{k}$  only occurs as  $\mathbf{h}(\_, \mathbf{k})$ .

If  $\mathbf{h}$  is collision-resistant, we have

$$\mathbf{h}(u, \mathbf{k}) = \mathbf{h}(v, \mathbf{k}) \Rightarrow u = v$$

when  $u, v$  are variable-free terms where  $\mathbf{k}$  only occurs as  $\mathbf{h}(\_, \mathbf{k})$ .

## Cryptographic reasoning: hash functions

Assume  $\mathbf{h}$  models a keyed hash functions.

If  $\mathbf{h}$  is EUF-CMA secure, we have

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

where  $S = \{ s \mid \mathbf{h}(s, \mathbf{k}) \text{ occurs in } u, v \}$

and  $u, v$  are variable-free terms where  $\mathbf{k}$  only occurs as  $\mathbf{h}(\_, \mathbf{k})$ .

If  $\mathbf{h}$  is collision-resistant, we have

$$\mathbf{h}(u, \mathbf{k}) = \mathbf{h}(v, \mathbf{k}) \Rightarrow u = v$$

when  $u, v$  are variable-free terms where  $\mathbf{k}$  only occurs as  $\mathbf{h}(\_, \mathbf{k})$ .

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

## Further notes

### What's in the full local logic?

- Equalities, quantification over indices, boolean connectives, etc.  
Can be seen as PPTIME computation because **index** is finite.

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Stanislas Riou (PhD student) works on automating the local logic using SMT solvers.

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Stanislas Riou (PhD student) works on automating the local logic using SMT solvers.

### What's beyond the local logic?

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

- overwhelming truth of a local formula,
- indistinguishability between sequences of messages,

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Stanislas Riou (PhD student) works on automating the local logic using SMT solvers.

### What's beyond the local logic?

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

- overwhelming truth of a local formula,
- indistinguishability between sequences of messages,
- exact truth of a local formula, (non/bi)-deducibility, etc.

## Further notes

### What's in the full local logic?

- With quantifications on **message**, formulas are not PPTIME anymore, but we can refine our axioms and tactics to carry over.
- Function types, polymorphism, higher-order quantification... for modularity/abstraction in proofs. Recursive definitions (next).

Stanislas Riou (PhD student) works on automating the local logic using SMT solvers.

### What's beyond the local logic?

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

- overwhelming truth of a local formula,
- indistinguishability between sequences of messages,
- exact truth of a local formula, (non/bi)-deducibility, etc.

Cryptographic tactics are slowly being subsumed by bi-deduction and cryptographic games thanks to the PhD work of Justine Sauvage.

# Outline

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

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

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

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

- a local formula describing the executability **condition**;
- a **message** term describing its **output**.

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

### Example (Signed DH with several sessions)

$\mathbf{A}(i)$  = first action of Alice for session  $i$ :

- Executes if **true**.
- Outputs  $\mathbf{g}^{\times(i)}$ .

# Systems of actions

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

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

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

- a local formula describing the executability **condition**;
- a **message** term describing its **output**.

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

## Example (Signed DH with several sessions)

$\text{B}(j)$  = first action of Bob for session  $j$ :

- Executes if **true**.
- Outputs  $\langle \text{g}^{y(j)}, \text{sign}(\langle \text{g}^{y(j)}, \text{input}@\text{B}(j) \rangle, \text{sk}\smiley) \rangle$ .

# Systems of actions

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

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

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

- a local formula describing the executability **condition**;
- a **message** term describing its **output**.

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

## Example (Signed DH with several sessions)

$\text{A}_1(i)$  = second action of Alice for session  $i$ , upon success:

- Executes if  
 $\text{verify}\left(\langle \text{fst}(\text{input}@\text{A}_1(i)), g^{x(i)} \rangle, \text{snd}(\text{input}@A_1(i)), \text{pub}(\text{sk}\smiley) \right)$ .
- Outputs  $\text{sign}(\langle g^{x(i)}, \text{fst}(\text{input}@A_1(i)) \rangle, \text{sk}\smiley)$ .

# Systems of actions

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

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

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

- a local formula describing the executability **condition**;
- a **message** term describing its **output**.

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

## Example (Signed DH with several sessions)

$\mathbf{A}_2(i)$  = second action of Alice for session  $i$ , upon failure:

- Executes if  
 $\neg\text{verify}(\langle \text{fst}(\text{input}@A_2(i)), g^{x(i)} \rangle, \text{snd}(\text{input}@A_2(i)), \text{pub}(\text{sk}\heartsuit))$ .
- Outputs .

# Systems of actions

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

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

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

- a local formula describing the executability **condition**;
- a **message** term describing its **output**.

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

## Example (Signed DH with several sessions)

$\mathbf{A}_2(i)$  = second action of Alice for session  $i$ , upon failure:

- Executes if  
 $\neg\text{verify}(\langle \text{fst}(\text{input}@A_2(i)), g^{x(i)} \rangle, \text{snd}(\text{input}@A_2(i)), \text{pub}(\text{sk}\heartsuit))$ .
- Outputs .

Demo: `signed-dh-many.sp` (actions compiled from  $\pi$ -calculus process)

## Traces of actions

A trace is a non-repeating sequence of actions subject to protocol-specific conditions:

- A(1).B(7).A(1) is **not** a trace.
- A(1).B(7).A<sub>1</sub>(1) is a trace.

## Traces of actions

A trace is a non-repeating sequence of actions subject to protocol-specific conditions:

- A(1).B(7).A(1) is **not** a trace.
- A(1).B(7).A<sub>1</sub>(1) is a trace.
- A(1).B(7).A<sub>1</sub>(3) is **not** a trace.

## Traces of actions

A trace is a non-repeating sequence of actions subject to protocol-specific conditions:

- $A(1).B(7).A(1)$  is **not** a trace.
- $A(1).B(7).A_1(1)$  is a trace.
- $A(1).B(7).A_1(3)$  is **not** a trace.
- $A(1).B(7).A_2(1)$  is a trace.

## Traces of actions

A trace is a non-repeating sequence of actions subject to protocol-specific conditions:

- $A(1).B(7).A(1)$  is **not** a trace.
- $A(1).B(7).A_1(1)$  is a trace.
- $A(1).B(7).A_1(3)$  is **not** a trace.
- $A(1).B(7).A_2(1)$  is a trace.
- $A(1).B(7).A_1(1).A_2(1)$  is **not** a trace.

## Traces of actions

A trace is a non-repeating sequence of actions subject to protocol-specific conditions:

- $A(1).B(7).A(1)$  is **not** a trace.
- $A(1).B(7).A_1(1)$  is a trace.
- $A(1).B(7).A_1(3)$  is **not** a trace.
- $A(1).B(7).A_2(1)$  is a trace.
- $A(1).B(7).A_1(1).A_2(1)$  is **not** a trace.
- $B(7).A(1).A_1(1)$  is a trace.

A trace just indicates a **tentative schedule** for actions.  
Depending on the interpretation of primitives and attackers,  
it will execute with a certain probability.

## Modelling traces in the logic

We use terms of sort **timestamp**:

- **happens**( $\tau$ ) means that  $\tau$  is part of the trace
- **init** is the first timestamp that happens
- $<$  is a total order on timestamps that happen

Each trace yields a **trace model**, i.e.,  
an interpretation for the action symbols, **happens** and  $<$ .

# Modelling traces in the logic

We use terms of sort **timestamp**:

- **happens**( $\tau$ ) means that  $\tau$  is part of the trace
- **init** is the first timestamp that happens
- $<$  is a total order on timestamps that happen

Each trace yields a **trace model**, i.e.,  
an interpretation for the action symbols, **happens** and  $<$ .

## Injectivity (part of **auto** tactic)

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

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

# Modelling traces in the logic

We use terms of sort **timestamp**:

- `happens`( $\tau$ ) means that  $\tau$  is part of the trace
- `init` is the first timestamp that happens
- $<$  is a total order on timestamps that happen

Each trace yields a **trace model**, i.e.,  
an interpretation for the action symbols, `happens` and  $<$ .

## Order (part of `auto` tactic)

- `happens`( $\tau$ )  $\wedge$  `happens`( $\tau'$ )  $\Leftrightarrow \tau \leq \tau' \vee \tau' \leq \tau$
- `happens`(`pred`( $\tau$ ))  $\Rightarrow$  `pred`( $\tau$ )  $< \tau$

# Modelling traces in the logic

We use terms of sort **timestamp**:

- **happens**( $\tau$ ) means that  $\tau$  is part of the trace
- **init** is the first timestamp that happens
- $<$  is a total order on timestamps that happen

Each trace yields a **trace model**, i.e.,  
an interpretation for the action symbols, **happens** and  $<$ .

## Case analysis and induction (**case** and **induction**)

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

# Modelling traces in the logic

We use terms of sort **timestamp**:

- `happens`( $\tau$ ) means that  $\tau$  is part of the trace
- `init` is the first timestamp that happens
- $<$  is a total order on timestamps that happen

Each trace yields a **trace model**, i.e.,  
an interpretation for the action symbols, `happens` and  $<$ .

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

- $\forall i. \text{happens}(\text{A}_1(i)) \Rightarrow \text{A}(i) < \text{A}_1(i)$  (dependency)
- $\forall i. \neg(\text{happens}(\text{A}_1(i)) \wedge \text{happens}(\text{A}_2(i)))$  (conflict)

## Macros

Given a trace we define<sup>1</sup> recursively several **macros** encoding the attacker's interaction with the protocol along that trace:

$$\begin{aligned}\text{output}@\tau &= \langle \text{output of action } \tau \rangle && \text{if } \text{init} < \tau \\ \text{cond}@\tau &= \langle \text{condition of action } \tau \rangle && \text{if } \text{init} < \tau\end{aligned}$$

---

<sup>1</sup>Missing cases are not important.

## Macros

Given a trace we define<sup>1</sup> recursively several **macros** encoding the attacker's interaction with the protocol along that trace:

$\text{output}@\tau$	$= \langle \text{output of action } \tau \rangle$	if $\text{init} < \tau$
$\text{cond}@\tau$	$= \langle \text{condition of action } \tau \rangle$	if $\text{init} < \tau$
$\text{exec}@{\text{init}}$	$= \text{true}$	
$\text{exec}@\tau$	$= \text{exec}@{\text{pred}}(\tau) \wedge \text{cond}@\tau$	if $\text{init} < \tau$

---

<sup>1</sup>Missing cases are not important.

## Macros

Given a trace we define<sup>1</sup> recursively several **macros** encoding the attacker's interaction with the protocol along that trace:

$\text{output}@\tau$	$= \langle \text{output of action } \tau \rangle$	if $\text{init} < \tau$
$\text{cond}@\tau$	$= \langle \text{condition of action } \tau \rangle$	if $\text{init} < \tau$
$\text{exec}@{\text{init}}$	$= \text{true}$	
$\text{exec}@\tau$	$= \text{exec}@{\text{pred}}(\tau) \wedge \text{cond}@\tau$	if $\text{init} < \tau$
$\text{frame}@\text{init}$	$= \text{empty}$	
$\text{exec}@\tau$	$= \langle \text{frame}@{\text{pred}}(\tau),$ $\quad \langle \text{exec}@\tau,$ $\quad \quad \text{if } \text{exec}@\tau \text{ then } \text{output}@\tau \text{ else } \text{empty} \rangle \rangle$	if $\text{init} < \tau$

---

<sup>1</sup>Missing cases are not important.

## Macros

Given a trace we define<sup>1</sup> recursively several **macros** encoding the attacker's interaction with the protocol along that trace:

$\text{output}@\tau$	$= \langle \text{output of action } \tau \rangle$	if $\text{init} < \tau$
$\text{cond}@\tau$	$= \langle \text{condition of action } \tau \rangle$	if $\text{init} < \tau$
$\text{exec}@{\text{init}}$	$= \text{true}$	
$\text{exec}@\tau$	$= \text{exec}@{\text{pred}}(\tau) \wedge \text{cond}@\tau$	if $\text{init} < \tau$
$\text{frame}@\text{init}$	$= \text{empty}$	
$\text{exec}@\tau$	$= \langle \text{frame}@{\text{pred}}(\tau),$ $\quad \langle \text{exec}@\tau,$ $\quad \quad \text{if } \text{exec}@\tau \text{ then } \text{output}@\tau \text{ else } \text{empty} \rangle \rangle$	if $\text{init} < \tau$
$\text{input}@\tau$	$= \text{att}(\text{frame}@\tau)$	if $\text{init} < \tau$

---

<sup>1</sup>Missing cases are not important.

## Security properties for all traces

Additional macros reflect let-definitions used in processes.  
For example  $\text{Y}'@\text{A}_1(i)$  is the value of  $\text{Y}'$  in that action.

### 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: `signed-dh-many.sp`

## Reasoning with recursive definitions

Constraining occurrences becomes more complex with macros.

Example (Freshness without macros nor indices)

$t \neq n$  is valid for any variable-free term  $t$  that does not contain  $n$

## Reasoning with recursive definitions

Constraining occurrences becomes more complex with macros.

### Example (Freshness without macros nor indices)

$t \neq n$  is valid for any variable-free term  $t$  that does not contain  $n$

### Example (Freshness without macros)

$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j}$  valid for any term  $t$  without message variables

## Reasoning with recursive definitions

Constraining occurrences becomes more complex with macros.

### Example (Freshness without macros nor indices)

$t \neq n$  is valid for any variable-free term  $t$  that does not contain  $n$

### Example (Freshness without macros)

$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j}$  valid for any term  $t$  without message variables

### Example (Freshness with macros)

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

valid for any term  $t$  without message variables,

where  $n(j) \in A(\vec{k})$  denotes occurrences in output or condition of  $A$

## Reasoning with recursive definitions

Constraining occurrences becomes more complex with macros.

### Example (Freshness without macros nor indices)

$t \neq n$  is valid for any variable-free term  $t$  that does not contain  $n$

### Example (Freshness without macros)

$t = n(\vec{i}) \Rightarrow \bigvee_{n(\vec{j}) \in t} \vec{i} = \vec{j}$  valid for any term  $t$  without message variables

### Example (Freshness with macros)

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

valid for any term  $t$  without message variables,

where  $n(j) \in A(\vec{k})$  denotes occurrences in output or condition of  $A$

Further refinements are possible and even necessary in practice.

## Further notes

### Protocols with mutable state

Protocols with mutable memory cells are supported (using `cell@τ` macros). The translation from processes to systems of actions, and its soundness, is recent work notably involving Clément Herouard.

## Further notes

### Protocols with mutable state

Protocols with mutable memory cells 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  $\eta$  and previous messages.

This is the topic of Théo Vigneron's ongoing PhD thesis.

# What's next?

Hands on experience in practical sessions!



Learn some more on our website, with more tutorials and papers:

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