

MPRI

SECURE: Proofs of Security Protocols

1. A Formal Framework for Cryptographic Protocol Verification

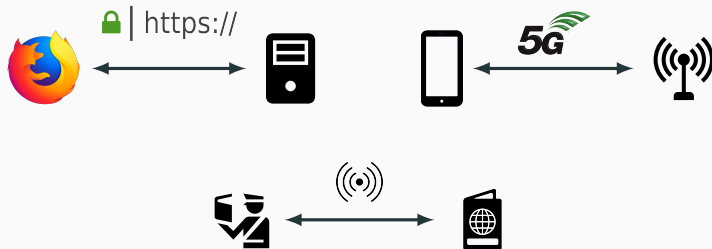
Adrien Koutsos, Inria Paris

2026/2027

Introduction

Security Protocols

- **Distributed programs** which aim at providing some **security properties**.
- Uses **cryptographic primitives**: e.g. encryptions, signatures, hashes.



Such protocols can target a large variety of **security properties**:

- **confidentiality, authentication, privacy, ...**

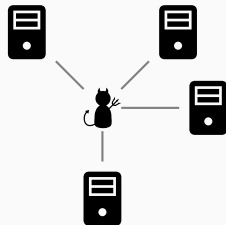
Context: Attacker Model

Against whom should these properties hold?

- in the **real world**: malicious individuals, corporations, state agencies, ...
- more **abstractly**: one (or many) computers sitting on the network.

Abstract attacker model

- **Network capabilities**: worst-case scenario *eavesdrop, block and forge* messages.
- **Computational capabilities**: the adversary's *computational power*.
- **Side-channels capabilities**: observing the agents (e.g. time, power-consumption).
⇒ not in this lecture.



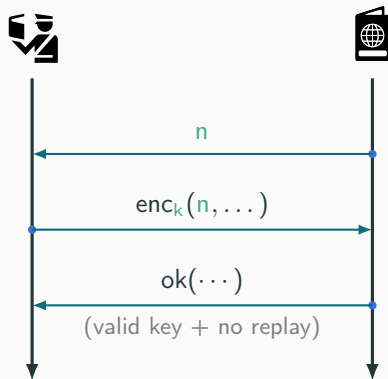
BAC Protocol (simplified)

The Basic Access Control protocol deployed in **e-passports**:

- uses an RFID tag;
- guard access to information stored;
- should guarantee **data confidentiality** and **user privacy**.

Some security mechanisms:

- **integrity**: obtaining **key k** requires **physical access**.
- **no replay**: random nonce n , old messages cannot be re-used.



BAC Protocol (simplified)

Privacy (Unlinkability)

No adversary can know whether it interacted with a particular user, **in any context**.

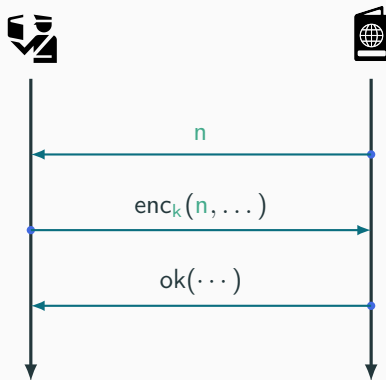
Example. For two user sessions:

$$\text{att} \left(\left[\text{book icon} \right], \left[\text{book icon} \right] \right) = \left\{ \begin{array}{ll} \left[\text{purple book icon} \right], \left[\text{purple book icon} \right] ? \\ \left[\text{purple book icon} \right], \left[\text{red book icon} \right] ? \end{array} \right.$$

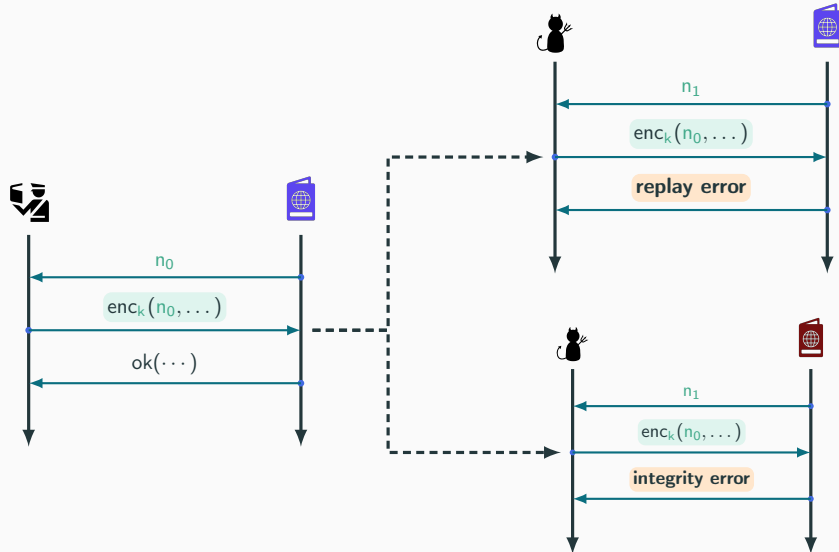
French version of BAC:

≠ error messages for replay and integrity checks.

⇒ **unlinkability attack**.



BAC Protocol: Privacy Attack



BAC Protocol: Lessons

Take-away lessons:

- This is a **protocol-level attack**: no issue with cryptography.
⇒ cryptographic primitives are but an **ingredient**.
- **Innocuous-looking changes** can **break** security.
⇒ designing security protocols is **hard**.

Protocol-level attacks are found even on **critical** cryptographic protocols under a lot of **public scrutiny**. Some examples:

- **TLS**, e.g. **LogJam**₁₄ [1], **TripleHandshake**₁₅ [10], **Drown**₁₆ [2], ...
- **Wifi encryption** with WPA2, **nonce reuse attacks**_{17,18} [12, 13].
- **Credit card** payment system: **PIN code bypass**₂₁ [8, 9].

How to get a **strong confidence** in a protocol's **security guarantees**?

High-Confidence Security Guarantees

Verification

Formal mathematical **proof** of security protocols:



- Must be **sound**: $\text{proof} \Rightarrow \text{property}$ always holds.
- Usually **undecidable**: approaches either **incomplete** or **interactive**.
- **Machine-checked proofs** yield a high degree of confidence.
 - ▶ **General-purpose** tools (e.g. **Rocq** and **Lean**).
 - ▶ In security protocol analysis, often in **dedicated** tools.
E.g. **CRYPTOVERIF** [11], **EASYCRYPT** [7, 6], **SQUIRREL** [4, 3].

Computer-aided Verification of Cryptographic Protocols

Goal of these lectures

Design **formal frameworks** allowing for the **mechanized verification** of **cryptographic protocols**.

- At the intersection of **cryptography** and **verification**.
- Particular verification challenges:
 - ▶ small or medium-sized programs
 - ▶ complex properties
 - ▶ concurrent and probabilistic programs + arbitrary adversary

What is **not** in these lectures:

- **Post-quantum cryptography.**
- Verification of **implementations** and **side-channels**.
(See Aymeric Fromherz's lectures.)
- Using **tools** (we focus on foundations).
(See Bruno Blanchet's lectures.)

The CCSA Approach to Cryptographic Protocol Verification

Lecture 1 to 3

The Computationally Complete Symbolic Attacker (**CCSA**) [5] is a framework in the **computational model** for the **verification** of cryptographic protocols.

Key ingredients

- Protocol executions modeled as **pure symbolic terms**.
- A **probabilistic logic**.
⇒ interpret terms as PTIME-computable bitstring distributions.
- **Reasoning rules** capturing **cryptographic arguments**.
- **Abstract approach**: no probabilities, no security parameter.

Lecture 4

The **probabilistic Relational Hoare Logic** (pRHL) approach to cryptographic verification.

Outline

Introduction

Processes

- Process Syntax

- Terms Syntax and Semantics

- Process Semantics

A Motivating Example

Symbolic Protocol Execution

- Terms

- Symbolic Rules

Semantics of Terms

Processes

Processes

Process Syntax

Process: Syntax

Elementary processes

$$E ::= \mathbf{in}(c, x) \mid \mathbf{out}(c, t) \mid \nu n \mid \text{if } b \text{ then } E \mid E.E \mid \mathbf{null} \quad (c \in \mathcal{C}, \ x, n \in \mathcal{X})$$

where $\mathcal{C}$ is a set of channel symbols and $\mathcal{X}$ a set of variables.

Processes

$$P ::= \nu n_1, \dots, n_l. (P_1 \mid \dots \mid P_k) \quad (n_1, \dots, n_l \in \mathcal{X})$$

We let $\text{chans}(P)$ be the channels of a process P .

Restrictions: elementary process must use a single channel, and distinct elementary processes must use distinct channels. I.e. if $P = \nu \vec{n}. E_1 \mid \dots \mid E_n$

$$\text{then } \forall i. |\text{chans}(E_i)| \leq 1 \quad \forall i \neq j, \text{chans}(E_i) \cap \text{chans}(E_j) = \emptyset.$$

Example of a Protocol

As an example, we consider a simple authentication protocol:

The Private Authentication (PA) Protocol, v1

$$\begin{aligned} I &= \nu n_I. \quad \text{out}(I, \{\langle pk_I, n_I \rangle\}_{pk_S}) \\ S &= \nu n_S. \text{in}(S, x). \text{out}(S, \{\langle \pi_2(\text{dec}(x, sk_S)), n_S \rangle\}_{pk_I}) \end{aligned}$$

where $pk_I \equiv pk(k_I)$ and $pk_S \equiv pk(k_S)$.

The full protocol is $\nu k_I. \nu k_S. (I \mid S)$.

Notation: $\equiv$ denotes **syntactic** equality of terms.

Processes

Terms Syntax and Semantics

Terms

We model *protocol messages* using **terms**, built upon a set of **symbols** $\mathcal{S}$ which includes:

- **Variables** $\mathcal{X}$, for example x in $\text{in}(A, x)$ or n in νn .
- **Function symbols** $\mathcal{F}$, e.g.:

$$\langle \cdot, \cdot \rangle, \pi_1(\cdot), \pi_2(\cdot) \quad \{\cdot\}!, \text{pk}(\cdot), \text{sk}(\cdot), \\ \text{if } \cdot \text{ then } \cdot \text{ else } \cdot \quad \cdot = \cdot \quad \cdot \wedge \cdot, \cdot \vee \cdot, \cdot \rightarrow \cdot$$

We note $\mathcal{T}(\mathcal{S})$ the set of well-typed terms over $\mathcal{S}$ (see next slide).

Terms are usually written t , and boolean terms b .

Examples

$$\text{pk}(k_A)$$

$$\{\langle \text{pk}_A, n_A \rangle\}_{\text{pk}_B}$$

$$\pi_1(n_A)$$

Types

Each symbol $s \in \mathcal{S}$ comes with a type $\text{type}(s)$ of the form:

$$(\tau_b^1 \star \cdots \star \tau_b^n) \rightarrow \tau_b \quad \text{or} \quad \tau_b$$

where $\tau_b^1, \dots, \tau_b^n, \tau_b$ are all **base types** in $\mathbb{B}$.

- We ask that $\mathbb{B}$ contains at least the `message` and `bool` types.
- We restrict *variables* to base types, i.e.:

$$\forall x \in \mathcal{X}, \text{type}(x) \in \mathbb{B}.$$

- We require that terms are well-typed and of a base type:

$$\vdash t : \tau_b \quad \text{where } \tau_b \in \mathbb{B}.$$

The **interpretation** $\langle t \rangle_{\mathbb{L}}^{\eta, \sigma}$ of a term t as a bitstring is parameterized by:

- the **security parameter** η ;
- a **library** $\mathbb{L}$ which provides the semantics $\langle \cdot \rangle_{\mathbb{L}}$ of **symbols** in $\mathcal{F}$ (details on next slides);
- the **valuation** $\sigma : \mathcal{X} \hookrightarrow \{0, 1\}^*$ mapping variables to their values.¹

We may omit σ , $\mathbb{L}$ and η when they are clear from the context.

¹ $f : \mathbb{A} \hookrightarrow \mathbb{B}$ denotes a *partial* f function from $\mathbb{A}$ to $\mathbb{B}$.

Function symbols.

For a function symbols $f \in \mathcal{F}$, we simply apply $\langle f \rangle_{\mathbb{L}}$:

$$\langle f(t_1, \dots, t_n) \rangle_{\mathbb{L}}^{\eta, \sigma} \stackrel{\text{def}}{=} \langle f \rangle_{\mathbb{L}}(1^\eta, \langle t_1 \rangle_{\mathbb{L}}^{\eta, \sigma}, \dots, \langle t_n \rangle_{\mathbb{L}}^{\eta, \sigma})$$

Restriction: $\langle f \rangle_{\mathbb{L}}$ must be **poly-time** computable and **deterministic**.

Variables.

The interpretation $\langle x \rangle_{\mathbb{L}}^{\eta, \sigma}$ of a **variable** $x \in \mathcal{X}$ is given by the valuation σ :

$$\langle x \rangle_{\mathbb{L}}^{\eta, \sigma} \stackrel{\text{def}}{=} \sigma(x)$$

Terms: Builtins

We **force** the interpretation of some **function symbols**.

- `if · then · else ·` is interpreted as **branching**:

$$\llbracket \text{if } \cdot \text{ then } \cdot \text{ else } \cdot \rrbracket_{\mathbb{M}}(1^\eta, b, v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} v_1 & \text{if } b = 1 \\ v_2 & \text{otherwise} \end{cases}$$

- `· = ·` is interpreted as an **equality** test:

$$\llbracket \cdot = \cdot \rrbracket_{\mathbb{M}}(1^\eta, v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } v_1 = v_2 \\ 0 & \text{otherwise} \end{cases}$$

Similarly, we force the interpretations of $\wedge, \vee, \rightarrow, \text{true}, \text{false}, \dots$

Processes

Process Semantics

Process: Concrete Configuration

A **concrete configuration** is a tuple (ϕ, σ, P) representing a **partially executed process** where:

- the **concrete frame** ϕ is the sequence bitstrings $w_1, \dots, w_n \in \{0, 1\}^*$ outputted since the protocol execution started.
- the **valuation** σ stores the inputs and random samplings.
- the **process** P remains to be executed.

A concrete configuration records the **current state** of an execution.

Initial configuration: $(\epsilon, [\text{st} \mapsto \epsilon], P)$

- **st** is a special variable used to store the **adversarial state**.

Finite Distributions

Processes are **probabilistic**:

⇒ The semantics of a process is a **distribution** of **configurations**.

Discrete distributions.

A **discrete distribution** over a set $\mathbb{S}$ can be written as a **formal sum** $\sum_{i \in I} a_i \cdot s_i$ where:

$$I \text{ is countable} \quad \sum_{i \in I} a_i = 1 \quad a_i \geq 0 \text{ for any } i \in I$$

Examples

- $\frac{1}{3} \cdot \text{"foo"} + \frac{2}{3} \cdot \text{"bar"}$
- $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1$ (unbiased coin flip).

Process: Concrete Semantics

A **trace** tr is a sequence of **observable actions** $\alpha_1, \dots, \alpha_n$.

The trace tr represents a protocol/adversary **interaction scenario**.

$$\alpha ::= \mathbf{in}(c) \mid \mathbf{out}(c) \qquad \text{tr} ::= \epsilon \mid \alpha \mid \text{tr}, \text{tr} \qquad (\text{where } c \in \mathcal{C})$$

The **concrete semantics** is given by the relation (see next slides):

$$(\phi, \sigma, P) \xrightarrow[\mathbb{L}, \mathcal{A}, \eta]{\text{tr}} \sum_{i \in I} a_i \cdot (\phi_i, \sigma_i, P_i)$$

Adversary. $\mathcal{A}$ is a **stateful**, **probabilistic** and **poly-time** program.

Notations. We omit $\mathcal{A}, \mathbb{L}, \eta$ when they are fixed or clear from context.

Further, we write $\Rightarrow$ instead of $\xRightarrow{\epsilon}$.

Structural rules

$$\overline{(\phi, \sigma, \mathbf{null} \mid P) \Rightarrow (\phi, \sigma, P)}$$

$$\overline{(\phi, \sigma, P) \Rightarrow (\phi, \sigma, P') \text{ when } P \approx_{\text{AC}} P'}$$

where $\approx_{\text{AC}}$ is the small congruence relation over processes which contains:

- **commutativity** $P_0 \mid P_1 \approx_{\text{AC}} P_1 \mid P_0$;
- **associativity** $(P_0 \mid P_1) \mid P_2 \approx_{\text{AC}} P_0 \mid (P_1 \mid P_2)$;
- **α -renaming** $\nu \mathbf{n}_0. P \approx_{\text{AC}} \nu \mathbf{n}_1. P[\mathbf{n}_0 \mapsto \mathbf{n}_1]$
(same for inputs $\mathbf{in}(c, x). P$).

Branching rules

$$\frac{\langle b \rangle_{\mathbb{L}}^{\eta, \sigma} = 1}{(\phi, \sigma, \text{if } b \text{ then } P \mid P') \Rightarrow (\phi, \sigma, P \mid P')}$$

$$\frac{\langle b \rangle_{\mathbb{L}}^{\eta, \sigma} = 0}{(\phi, \sigma, \text{if } b \text{ then } P \mid P') \Rightarrow (\phi, \sigma, P')}$$

Output rules

$$\frac{\phi' = (\phi, \langle t \rangle_{\mathbb{L}}^{\eta, \sigma})}{(\phi, \sigma, (\text{out}(c, t); P) \mid P') \xRightarrow{\text{out}(c)} (\phi', \sigma, P \mid P')}$$

$$\frac{\begin{array}{l} \phi' = (\phi, \text{error}) \\ c \notin \text{chans}(P) \text{ or } P = (\text{in}(c, x). P_0) \end{array}}{(\phi, \sigma, P) \xRightarrow{\text{out}(c)} (\phi', \sigma, P)}$$

New rule

$$\frac{n \notin \text{dom}(\sigma)}{(\phi, \sigma, (\nu n; P) \mid P') \Rightarrow \sum_{|w|=\eta} \frac{1}{2^\eta} \cdot (\phi, \sigma[n \mapsto w], P \mid P')}$$

Input rule

$$\frac{x \notin \text{dom}(\sigma) \quad \mathcal{A}(1^\eta, \phi, \sigma[\text{st}]) = \sum_i a_i \cdot (w_i, s_i)}{(\phi, \sigma, (\text{in}(c, x); P) \mid P') \xrightarrow{\text{in}(c)} \sum_i a_i \cdot (\phi', \sigma[x \mapsto w_i, \text{st} \mapsto s_i], P \mid P')}$$
$$\frac{c \notin \text{chans}(P) \text{ or } P = (\text{out}(c, t). P_0) \quad \mathcal{A}(1^\eta, \phi, \sigma[\text{st}]) = \sum_i a_i \cdot (w_i, s_i)}{(\phi, \sigma, P) \xrightarrow{\text{in}(c)} \sum_i a_i \cdot (\phi', \sigma[\text{st} \mapsto s_i], P)}$$

Remark: $\text{dom}(f) \stackrel{\text{def}}{=} \{x \mid f(x) \text{ defined}\}$ is the domain of the partial function f .

Transitivity rule

$$\frac{(\phi, \sigma, P) \xRightarrow{\text{tr}_0} \sum_i a_i \cdot (\phi_i, \sigma_i, P_i) \quad \forall i. (\phi_i, \sigma_i, P_i) \xRightarrow{\text{tr}_1} \sum_j b_{i,j} \cdot (\phi_{i,j}, \sigma_{i,j}, P_{i,j})}{(\phi, \sigma, P) \xRightarrow{\text{tr}_0, \text{tr}_1} \sum_{i,j} a_i \cdot b_{i,j} \cdot (\phi_{i,j}, \sigma_{i,j}, P_{i,j})}$$

Process: Concrete Semantics

The relation $\Rightarrow$ is **non-deterministic**.

Still, thanks to the **restrictions** on processes, different non-deterministic choices yield **identical distributions** on frames:

$$\begin{aligned} & (\phi, \sigma, P) \xRightarrow{\text{tr}} \sum_i a_i \cdot (\phi_0^i, \sigma_0^i, P_0^i) \\ \wedge \quad & (\phi, \sigma, P) \xRightarrow{\text{tr}} \sum_j b_j \cdot (\phi_1^j, \sigma_1^j, P_1^j) \\ \Rightarrow \quad & \sum_i a_i \cdot \phi_0^i = \sum_j b_j \cdot \phi_1^j \end{aligned}$$

Thus, given a process P and a trace tr , if:

$$(\epsilon, [\text{st} \mapsto \epsilon], P) \xRightarrow[\mathbb{L}, \mathcal{A}, \eta]{\text{tr}} \sum_i a_i \cdot (\phi_i, \sigma_i, P_i)$$

then we pose $\text{c-frame}_{\mathbb{L}, \mathcal{A}}^\eta(P, \text{tr}) = \sum_i a_i \cdot \phi_i$ the **concrete execution** of P over tr .

A Motivating Example

Toy Protocol

We consider a toy protocol as a motivating example:

$A : \nu n. \quad \text{out}(A, \{n\}_k). \text{out}(A, n)$

$B : \text{in}(B, x). \text{if } \text{dec}(x, k) \neq \perp \text{ then}$
 $\quad \text{in}(B, y). \text{out}(B, \text{dec}(x, k) = y)$

Security property

B outputs false if A does not send its second message.

Assumptions (informally)

We assume that the *symmetric encryption* $\{\cdot\}$ is a secure AE:

- **Integrity:** if $\text{dec}(m, k) \neq \perp$ then m is an honest encryption.
- **Confidentiality:** $\{m_0\}_k$ and $\{m_1\}_k$ are indistinguishable.
(when m_0 and m_1 are of the same length.)

Security of our Toy Protocol

$A : \nu n. \quad \text{out}(A, \{n\}_k). \text{out}(A, n)$
 $B : \text{in}(B, x). \text{if } \text{dec}(x, k) \neq \perp \text{ then}$
 $\quad \text{in}(B, y). \text{out}(B, \text{dec}(x, k) = y)$

When executing the trace

$\text{out}(A), \text{in}(B), \text{in}(B), \text{out}(B),$

the adversary sees the messages:

$\{n\}_k, \text{if } (\text{dec}(\text{att}_1(\Phi), k) \neq \perp) \text{ then } (\text{dec}(\text{att}_1(\Phi), k) = \text{att}_2(\Phi))$

where $\Phi = \{n\}_k$, and $\text{att}_1, \text{att}_2$ represents, resp., the first and second inputs to B chosen by the adversary.

Security of our Toy Protocol

Our toy protocol is **secure** if the following **indistinguishability** holds:

$$\begin{aligned} & \{n\}_k, \text{ if } (\text{dec}(\text{att}_1(\Phi), k) \neq \perp) \text{ then } (\text{dec}(\text{att}_1(\Phi), k) = \text{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\text{dec}(\text{att}_1(\Phi), k) \neq \perp) \text{ then false} \end{aligned}$$

Remark

This is a **pure encoding** (no runtime samplings, no state)
of an **adversary/protocol interaction**.

Informal Security Proof

Using the **integrity** of the encryption, we have:

$$(\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \Leftrightarrow (\mathbf{att}_1(\Phi) = \{n\}_k).$$

Thus:

$$\begin{aligned} & \{n\}_k, \text{ if } (\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (n = \mathbf{att}_2(\Phi)) \end{aligned}$$

Informal Security Proof

Using the **integrity** of the encryption, we have:

$$(\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \Leftrightarrow (\mathbf{att}_1(\Phi) = \{n\}_k).$$

Thus:

$$\begin{aligned} & \{n\}_k, \text{ if } (\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (n = \mathbf{att}_2(\Phi)) \end{aligned}$$

Using the **confidentiality** of the encryption, we can replace $\{n\}_k$ by $\{0^\eta\}_k$:

$$\approx \{0^\eta\}_k, \text{ if } (\mathbf{att}_1(\{0^\eta\}_k) = \{0^\eta\}_k) \text{ then } (n = \mathbf{att}_2(\{0^\eta\}_k))$$

Informal Security Proof

Using the **integrity** of the encryption, we have:

$$(\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \Leftrightarrow (\mathbf{att}_1(\Phi) = \{n\}_k).$$

Thus:

$$\begin{aligned} & \{n\}_k, \text{ if } (\text{dec}(\mathbf{att}_1(\Phi), k) \neq \perp) \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (\text{dec}(\mathbf{att}_1(\Phi), k) = \mathbf{att}_2(\Phi)) \\ \approx & \{n\}_k, \text{ if } (\mathbf{att}_1(\Phi) = \{n\}_k) \quad \text{ then } (n = \mathbf{att}_2(\Phi)) \end{aligned}$$

Using the **confidentiality** of the encryption, we can replace $\{n\}_k$ by $\{0^n\}_k$:

$$\approx \{0^n\}_k, \text{ if } (\mathbf{att}_1(\{0^n\}_k) = \{0^n\}_k) \text{ then } (n = \mathbf{att}_2(\{0^n\}_k))$$

By **probabilistic independence**: $n \neq \mathbf{att}_2(\{0^n\}_k)$ (overwhelmingly):

$$\approx \{0^n\}_k, \text{ if } (\mathbf{att}_1(\{0^n\}_k) = \{0^n\}_k) \text{ then false}$$

We conclude by a similar proof in the other direction.

Verification of Cryptographic Protocols

$$\forall \mathcal{A} \in \mathcal{C}. (\mathcal{A} \parallel P) \models \Phi$$

To build a verification framework from what we just did, we need to:

- Model the **adversary/protocol interaction** as **symbolic terms**.
- Express the **security property** Φ using a **logical formula**.
- Capture the **cryptographic arguments** $\models$ as **reasoning rules**.

Symbolic Protocol Execution

Goal: obtain **symbolic representations** of protocol executions that

- **faithfully** model the protocol semantics;
- are amenable to **formal reasoning**.

How? Use the same **techniques** as in our motivating example.

1. **Explicit probabilistic dependencies**
randomness is sampled **eagerly** and not **lazily**
2. **Pure encoding of the adversary**
no adversarial state

Symbolic Execution: Explicit Probabilistic Dependencies

1. Explicit probabilistic dependencies.

Key idea.

Sample randomness beforehand (eagerly), and retrieve it as needed.

Concretely, we rewrite a process P by **moving randomness early**:

$$\begin{aligned} P = E_1 \mid \cdots \mid E_l &\Leftrightarrow (\nu \vec{n}_1. E'_1) \mid \cdots \mid (\nu \vec{n}_l. E'_l) \\ &\Leftrightarrow \nu \vec{n}_1, \dots, \vec{n}_l. (E'_1 \mid \cdots \mid E'_l) \end{aligned}$$

where $(\nu \vec{n}_i. E'_i)$ is E_i with all random samplings moved at the beginning, and where all names are distinct:

$$\forall i \neq j. \vec{n}_i \cap \vec{n}_j = \emptyset.$$

Symbolic Execution: Explicit Probabilistic Dependencies

More precisely, $E_i \rightsquigarrow^! (\nu \vec{n}_i. E'_i)$ where $\rightsquigarrow$ is defined by the rules:

$$(E_0. \nu \mathbf{n}. E_1) \rightsquigarrow \nu \mathbf{n}. (E_0. E_1) \qquad (E_0 \mid \nu \mathbf{n}. E_1) \rightsquigarrow \nu \mathbf{n}. (E_0 \mid E_1)$$

$$\text{if } b \text{ then } \nu \mathbf{n}. E \rightsquigarrow \nu \mathbf{n}. \text{if } b \text{ then } E$$

where $\mathbf{n} \notin \text{fv}(E_0)$ and $\mathbf{n} \notin \text{fv}(b)$.

💡 *Recall that process are taken modulo α -renaming.*

Notations. $E_0 \rightsquigarrow^! E_1$ iff $E_0 \rightsquigarrow^* E_1$ and $E_1 \not\rightsquigarrow E'$ for all E' .

$\text{fv}(E)$ and $\text{fv}(t)$ are the **free variables** of, resp., E and t .

2. Pure encoding of the adversary.

We need a **deterministic** and **stateless** representation $\mathcal{A}_p$ of the adversary $\mathcal{A}$.

- **deterministic**: sample the adversary's randomness ρ_a **eagerly** and pass it as an **explicit argument**.
- **stateless**: $\mathcal{A}_p$ recomputes $\mathcal{A}$'s state each time it is called.
 - 💡 *This exploits determinism, and requires us to **provide the full history** to $\mathcal{A}_p$.*

Symbolic Executions: Pure Encoding of the Adversary

Informally, we want that:

$$\begin{array}{c} \mathcal{A}(w_1) \quad , \quad \dots \quad , \quad \mathcal{A}(w_n) \\ \text{=distr.} \quad \mathcal{A}_p(w_1, \rho_a) \quad , \quad \dots \quad , \quad \mathcal{A}_p(w_1, \dots, w_n, \rho_a) \end{array}$$

where ρ_a is a *long enough* sequence of bits sampled **independently uniformly at random**.

We call $\mathcal{A}_p$ a **pure representation** of $\mathcal{A}$.

We do not detail it, but $\mathcal{A}_p$ can be systematically built from $\mathcal{A}$.

Symbolic Execution Technique

We describe a **systematic method** to compute, given a **process** P and a **trace** tr of **observable actions**, the **terms** representing the **outputted messages** during the execution of P over tr .

This is the **symbolic execution** of P over tr .

We deal with the protocol **randomness** and **adversarial inputs** using the two techniques we just saw.

Symbolic Protocol Execution

Terms

To define our **symbolic execution rules**, we need to **extend** our set of **term symbols** $\mathcal{S} = \mathcal{N} \uplus \mathcal{X} \uplus \mathcal{F} \uplus \mathcal{G}$:

- **Variables** $\mathcal{X}$.
- **Function symbols** $\mathcal{F}$.
- **Names** $\mathcal{N}$.
- **Adversarial function symbols** $\mathcal{G}$, of any arity.

Changes.

- Names are no longer represented by variables in $\mathcal{X}$, but by special symbols in $\mathcal{N}$ with a tailored semantics (presented later).
(For the sake of simplicity, we ask that all names are of type **message**.)
- Adversarial function symbols $\mathcal{G}$ are used to represent calls to $\mathcal{A}_p$. (Details next slide.)

Adversarial function symbols

More precisely, if:

- there has already been n **outputs** $t_1, \dots, t_n$;
- and we are doing the j -th **input** since the protocol started;

then the **input bitstring** is **modeled** by:

$$\mathbf{att}_j(t_1, \dots, t_n)$$

where $\mathbf{att}_j \in \mathcal{G}$ is of arity n .

💡 j allows to have different values for consecutive inputs.

Symbolic Protocol Execution

Symbolic Rules

Symbolic configuration

A **symbolic configuration** is a tuple $(\Phi; \lambda; j; \Pi_1, \dots, \Pi_l)$ where:

- Φ is a sequence of terms (in $\mathcal{T}(\mathcal{S})$).
- $\lambda : \mathcal{X} \hookrightarrow \mathcal{T}(\mathcal{S})$ is a symbolic valuation.
- $j \in \mathbb{N}$.
- for every i , $\Pi_i = (P_i, b_i)$ where P_i is a protocol and b_i is a boolean term.

Symbolic Configuration: Intuition

In a **symbolic configuration** $(\Phi; \lambda; j; \Pi_1, \dots, \Pi_l)$:

- Φ is the **symbolic frame**, i.e. the terms outputted since the execution started.
- λ **records inputs** by mapping input variable to their corresponding term.
- j **counts the number of inputs** since the execution started.
- $\Pi = (P, b)$ **represents** P if b is true (and is **null** otherwise).
Using this interpretation, $\Pi_1, \dots, \Pi_l$ is the **current process**.

Initial configuration: $(\epsilon; \emptyset; 0; (P, \top))$

Symbolic Execution: Branching and Input Rules

Rule for protocol branching

$$\begin{aligned} & (\Phi; \lambda; j; (\text{if } b_0 \text{ then } P, b_1), \Pi_1, \dots, \Pi_l) \\ \hookrightarrow & \quad (\Phi; \lambda; j; (P, b_0 \wedge b_1), \Pi_1, \dots, \Pi_l) \end{aligned}$$

Rules for inputs

$$\begin{aligned} & (\Phi; \lambda; j; (\mathbf{in}(c, x).P, b), \Pi_1, \dots, \Pi_l) \\ \xrightarrow{\mathbf{in}(c)} & (\Phi; \lambda[x \mapsto \mathbf{att}_j(\Phi)]; j+1; (P, b), \Pi_1, \dots, \Pi_l) \end{aligned} \quad (x \notin \text{dom}(\lambda))$$

$$(\Phi; \lambda; j; \Pi_1, \dots, \Pi_l) \xrightarrow{\mathbf{in}(c)} (\Phi; \lambda; j+1; \Pi_1, \dots, \Pi_l) \quad (\dagger)$$

where $(\dagger)$: $c \notin \text{chans}(\Pi_1, \dots, \Pi_l)$ or $\exists i$ s.t. Π_i starts with $\mathbf{out}(c, \cdot)$.

Rules for outputs

$$\begin{array}{c} (\Phi; \lambda; j; (\mathbf{out}(c, t).P, b), \Pi_1, \dots, \Pi_l) \\ \xrightarrow{\mathbf{out}(c)} (\Phi, (\text{if } b \text{ then } t)\lambda; \lambda; j; (P, b), \Pi_1, \dots, \Pi_l) \end{array}$$

$$(\Phi; \lambda; j; \Pi_1, \dots, \Pi_l) \xrightarrow{\mathbf{out}(c)} (\Phi, \mathbf{error}; \lambda; j; \Pi_1, \dots, \Pi_l) \quad (\ddagger)$$

where $(\ddagger)$: $c \notin \text{chans}(\Pi_1, \dots, \Pi_l)$ or $\exists i$ s.t. Π_i starts with $\mathbf{out}(c, \cdot)$.

💡 *The input and output rules make sense because we restrict ourselves to elementary processes with distinct channels.*

Given a process P (without ν) and a trace tr , if:

$$(\epsilon; \emptyset; 0; (P, \top)) \xrightarrow{\text{tr}} (\Phi; _; _; _)$$

then Φ is the **symbolic execution** of P over tr , denoted $\text{frame}(P, \text{tr})$.

Handling the ν construct.

If P contains ν , we compute P_0 s.t. $P \rightsquigarrow^! \nu \vec{n}. P_0$ with $\vec{n} \in \mathcal{N}$, and then symbolically execute P_0 .

Symbolic Execution: Soundness

Claim (informal)

The symbolic execution is **sound** w.r.t. the concrete semantics.

More precisely, for every library $\mathbb{L}$, adversary $\mathcal{A}$, and security parameter η :

$$\text{c-frame}_{\mathbb{L}, \mathcal{A}}^{\eta}(\mathbf{P}, \text{tr}) =_{\text{distr.}} \llbracket \text{frame}(\mathbf{P}, \text{tr}) \rrbracket_{\mathbb{L}[\mathbf{att} \mapsto \mathcal{A}_p]}^{\eta, \rho}$$

where:

- ρ is sampled uniformly at random among bitstrings of sufficient length.
- $\mathcal{A}_p$ is a pure representation of $\mathcal{A}$.

Remark: we define the semantics of the symbolic execution terms $\llbracket t \rrbracket_{\mathbb{M}}^{\eta, \rho}$ next.

Exercise

What are all the **possible symbolic executions** of the following protocols?

in(c, x). **out**(c, t)

out(A, t₁) | **in**(B, x). **out**(B, t₂)

if *b* then **out**(c, t₁) else **out**(c, t₂)

if *b* then **out**(A, t₁) else **out**(B, t₂)

Exercise

Extend the **symbolic** algorithm with a rule allowing to handle processes with let bindings.

Could the same thing be done for mutable, inter-process, state?

Semantics of Terms

Semantics of Terms

We saw how to represent a **protocol execution**, on some fixed trace of observables tr , as a **sequence of terms**.

Intuitively, the terms corresponds to **PTIME-computable bitstring distributions**.

Example

If $\langle _ , _ \rangle$ is the concatenation, and samplings are done uniformly at random among bitstrings of length $\eta \in \mathbb{N}$, then:

$$\nu n_0, \nu n_1, \text{out}(c, \langle n_0, \langle 00, n_1 \rangle \rangle) \quad \text{yields} \quad \langle n_0, \langle 00, n_1 \rangle \rangle$$

which represent a distribution over bitstrings of length $2 \cdot \eta + 2$, where all bits are sampled uniformly and independently, except for the bits at positions η and $\eta + 1$, which are always 0.

We interpret $t \in \mathcal{T}(\mathcal{S})$ as a **Probabilistic Polynomial-time Turing machine** (PPTM), with:

- a **working tape** (also used as input tape);
- two **read-only tapes** $\rho = (\rho_a, \rho_h)$ for adversary and honest randomness.

We let $\mathcal{D}$ be the set of such machines.

💡 *The machine must be polynomial in the size of its input on the working tape only.*

Terms Interpretation

The **interpretation** $\llbracket t \rrbracket_{\mathbb{M}} \in \mathcal{D}$ of a term t is parameterized by a **model** $\mathbb{M}$ which provides:

- the set of random tapes $\mathbb{T}_{\mathbb{M},\eta} = \mathbb{T}_{\mathbb{M},\eta}^a \times \mathbb{T}_{\mathbb{M},\eta}^h$, where $\mathbb{T}_{\mathbb{M},\eta}^a$ and $\mathbb{T}_{\mathbb{M},\eta}^h$ are **finite** same-length set of bit-strings.

We equip it with the **uniform** probability measure.

($\mathbb{T}_{\mathbb{M},\eta}^a$ for the adversary, $\mathbb{T}_{\mathbb{M},\eta}^h$ for honest functions)

- the semantics $\llbracket \cdot \rrbracket_{\mathbb{M}}$ of **symbols** in $\mathcal{S}$ (details on next slides).
(This extends the interpretation $\llbracket \cdot \rrbracket_{\mathbb{L}}$ of symbols by a library $\mathbb{L}$.)

We may omit $\mathbb{M}$ when it is clear from context.

We define the machine $\llbracket t \rrbracket_{\mathbb{M}} \in \mathcal{D}$, by defining its behavior $\llbracket t \rrbracket_{\mathbb{M}}^{\eta,\rho}$ for every $\eta \in \mathbb{N}$ and pairs of random tapes $\rho = (\rho_a, \rho_h) \in \mathbb{T}_{\mathbb{M},\eta}$.

Terms Interpretation: Function Symbols

Function symbols interpretations is just **composition**.

For **function symbols** in $f \in \mathcal{F}$, we simply apply $\langle f \rangle_{\mathbb{M}}$:

$$\llbracket f(t_1, \dots, t_n) \rrbracket_{\mathbb{M}}^{\eta, \rho} \stackrel{\text{def}}{=} \langle f \rangle_{\mathbb{M}}(1^\eta, \llbracket t_1 \rrbracket_{\mathbb{M}}^{\eta, \rho}, \dots, \llbracket t_n \rrbracket_{\mathbb{M}}^{\eta, \rho})$$

Adversarial function symbols $g \in \mathcal{G}$ also have access to ρ_a :

$$\llbracket g(t_1, \dots, t_n) \rrbracket_{\mathbb{M}}^{\eta, \rho} \stackrel{\text{def}}{=} \langle g \rangle_{\mathbb{M}}(1^\eta, \llbracket t_1 \rrbracket_{\mathbb{M}}^{\eta, \rho}, \dots, \llbracket t_n \rrbracket_{\mathbb{M}}^{\eta, \rho}, \rho_a)$$

Restrictions. $\langle f \rangle_{\mathbb{M}}$ and $\langle g \rangle_{\mathbb{M}}$ are:

- PTIME-computable;
- **deterministic** (all randomness must come explicitly from ρ).

Terms Interpretation: Variables and Names

The interpretation $\langle x \rangle_{\mathbb{M}}$ of a **variable** $x \in \mathcal{X}$ is an **arbitrary machine** in $\mathcal{D}$. Then:

$$\llbracket x \rrbracket_{\mathbb{M}}^{\eta, \rho} \stackrel{\text{def}}{=} \langle x \rangle_{\mathbb{M}}(1^\eta, \rho).$$

Names $n \in \mathcal{G}$ are interpreted as **uniform random samplings** among bitstrings of length η , extracted from ρ_h :

$$\llbracket n \rrbracket_{\mathbb{M}}^{\eta, \rho} \stackrel{\text{def}}{=} \langle n \rangle_{\mathbb{M}}(1^\eta, \rho_h)$$

For every pair of different names n_0, n_1 , we require that $\langle n_0 \rangle_{\mathbb{M}}$ and $\langle n_1 \rangle_{\mathbb{M}}$ extracts disjoint parts of ρ_h .

💡 Hence different names are **independent random samplings**.

Terms Interpretation: Names

Examples

- If $(n, n_0 : \text{message})$ then:

$$\llbracket n \rrbracket^\eta =_{\text{distr.}} \text{sample } w \text{ in } \{0, 1\}^\eta$$

$$\begin{aligned} \llbracket (n, n_0) \rrbracket^\eta &=_{\text{distr.}} \text{sample } w \text{ in } \{0, 1\}^\eta \\ &\quad \text{sample } w' \text{ in } \{0, 1\}^\eta \text{ independently} \\ &\quad \text{build } (w, w') \end{aligned}$$

$$\begin{aligned} \llbracket (n, n) \rrbracket^\eta &=_{\text{distr.}} \text{sample } w \text{ in } \{0, 1\}^\eta \\ &\quad \text{build } (w, w) \end{aligned}$$

Indeed:

$$\llbracket (n, n) \rrbracket^{\eta, \rho} = (\llbracket n \rrbracket^{\eta, \rho}, \llbracket n \rrbracket^{\eta, \rho}) = (w, w) \quad (\text{where } w = \llbracket n \rrbracket(1^\eta, \rho_h))$$

Terms Interpretation: Modeling and Randomness

$\neq$ in how randomness is sampled:

- In the “real-world”, the adversary $\mathcal{A}$ samples randomness on-the-fly, as needed.
 $\Rightarrow$ possibly $P(\eta)$ random bits, where P is the (polynomial) running-time of $\mathcal{A}$.
- In the logic, we restrict $\mathbb{T}_{\mathbb{M},\eta} = \mathbb{T}_{\mathbb{M},\eta}^a \times \mathbb{T}_{\mathbb{M},\eta}^h$ to be finite and fixed by $\mathbb{M}$.
 $\Rightarrow$ all randomness sampled eagerly according to $\mathbb{M}$, independently of $\mathcal{A}$.

This $\neq$ of behaviors is not an issue, i.e. the logic can **soundly model** real-world adversaries:

- Indeed, for any adversary $\mathcal{A}$, there exists a model $\mathbb{M}$ with enough randomness.

Verification of Cryptographic Protocols

$$\forall \mathcal{A} \in \mathcal{C}. (\mathcal{A} \parallel P) \models \Phi$$

- ✓ • Model the **adversary/protocol interaction** as **symbolic terms**.
- Express the **security property** Φ using a **logical formula**.
- Capture the **cryptographic arguments** $\models$ as **reasoning rules**.

References i

- [1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann.
Imperfect forward secrecy: how diffie-hellman fails in practice.
Commun. ACM, 62(1):106–114, 2019.
- [2] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohnen, S. Engels, C. Paar, and Y. Shavitt.
DROWN: breaking TLS using sslv2.
In *USENIX Security Symposium*, pages 689–706. USENIX Association, 2016.
- [3] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and J. Lallemand.
The squirrel prover and its logic.
ACM SIGLOG News, 11(2):62–83, 2024.
- [4] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau.
An interactive prover for protocol verification in the computational model.
In *SP*, pages 537–554. IEEE, 2021.
- [5] G. Bana and H. Comon-Lundh.
A computationally complete symbolic attacker for equivalence properties.
In *CCS*, pages 609–620. ACM, 2014.
- [6] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub.
Easycrypt: A tutorial.
In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

- [7] G. Barthe, B. Grégoire, and S. Zanella-Béguelin.
Probabilistic relational hoare logics for computer-aided security proofs.
In *MPC*, volume 7342 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2012.
- [8] D. A. Basin, R. Sasse, and J. Toro-Pozo.
Card brand mixup attack: Bypassing the PIN in non-visa cards by using them for visa transactions.
In *USENIX Security Symposium*, pages 179–194. USENIX Association, 2021.
- [9] D. A. Basin, R. Sasse, and J. Toro-Pozo.
The EMV standard: Break, fix, verify.
In *SP*, pages 1766–1781. IEEE, 2021.
- [10] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub.
Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS.
In *IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society, 2014.
- [11] B. Blanchet.
A computationally sound mechanized prover for security protocols.
IEEE Trans. Dependable Secur. Comput., 5(4):193–207, 2008.
- [12] M. Vanhoef and F. Piessens.
Key reinstallation attacks: Forcing nonce reuse in WPA2.
In *CCS*, pages 1313–1328. ACM, 2017.

- [13] M. Vanhoef and F. Piessens.
Release the kraken: New cracks in the 802.11 standard.
In *CCS*, pages 299–314. ACM, 2018.