Introduction to Symbolic and Computational Proofs

Bruno Blanchet

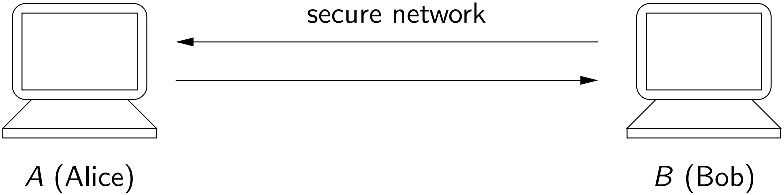
Inria, Paris, France
Bruno.Blanchet@inria.fr

December 2021

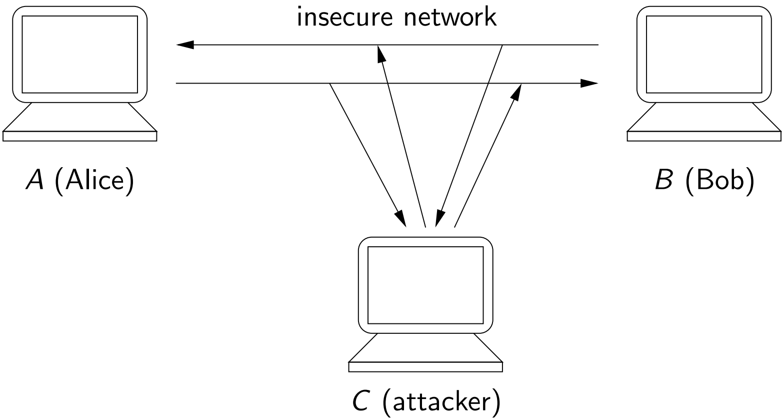
Outline

1. Introduction to security protocols
2. Verifying protocols in the symbolic model
3. Verifying protocols in the computational model
4. Conclusion and future challenges

Communications over a secure network



Communications over an insecure network



A talks to B on an insecure network

$\Rightarrow$ need for cryptography in order to make communications secure for instance, encrypt messages to preserve secrets.

Cryptographic primitives

Definition (Cryptographic primitives)

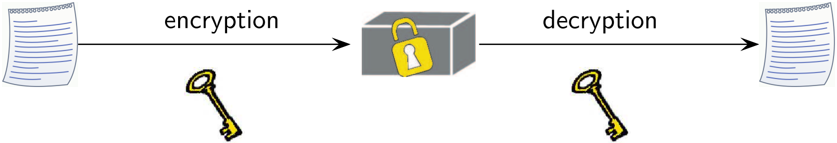
Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Cryptographic primitives

Definition (Cryptographic primitives)

Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Shared-key encryption

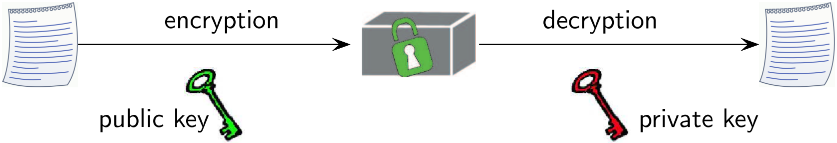


Cryptographic primitives

Definition (Cryptographic primitives)

Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Public-key encryption

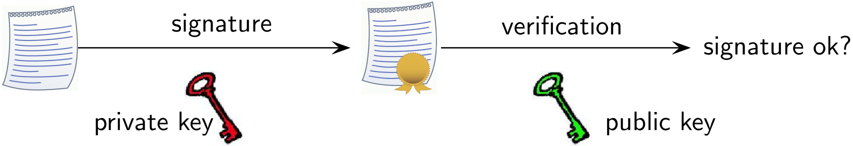


Cryptographic primitives

Definition (Cryptographic primitives)

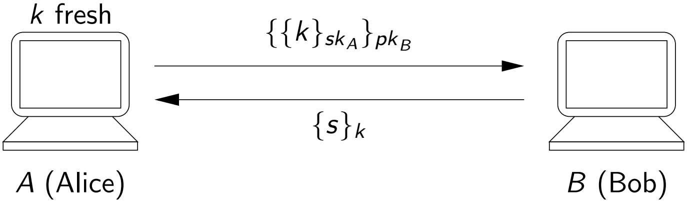
Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Signatures



Example

Denning-Sacco key distribution protocol [Denning, Sacco, 1981] (simplified)



k fresh

$\{\{k\}_{sk_A}\}_{pk_B}$

$\{s\}_k$

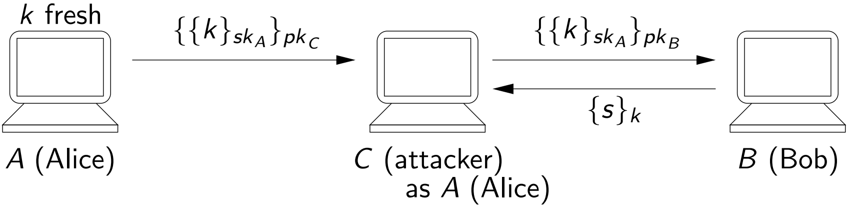
A (Alice)

B (Bob)

The goal of the protocol is that the key k should be a secret key, shared between A and B. So s should remain secret.

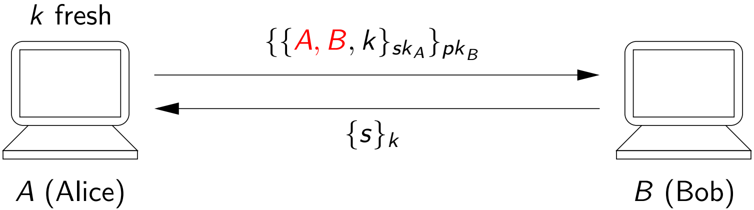
The attack

The (well-known) attack against this protocol.


k fresh
$\{\{k\}_{sk_A}\}_{pk_C}$
$\{\{k\}_{sk_A}\}_{pk_B}$
$\{s\}_k$
A (Alice)
C (attacker) as A (Alice)
B (Bob)

The attacker C impersonates A and obtains the secret s.

The corrected protocol



k fresh

$\{\{A, B, k\}_{sk_A}\}_{pk_B}$

$\{s\}_k$

A (Alice)

B (Bob)

Now C cannot impersonate A because in the previous attack, the first message is $\{\{A, C, k\}_{sk_A}\}_{pk_B}$, which is not accepted by B.

Examples

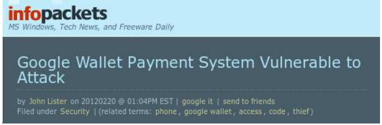
Many protocols exist, for various goals:

- secure channels: SSH (Secure SHell);
  SSL (Secure Socket Layer), renamed TLS (Transport Layer Security);
  IPsec
- e-voting
- contract signing
- certified email
- wifi (WEP/WPA/WPA2)
- banking
- mobile phones
- . . .

Why verify security protocols ?

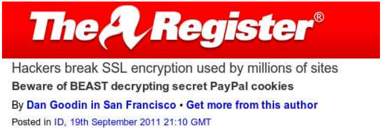
The verification of security protocols has been and is still a very active research area.

- Their design is error prone.
- Security errors not detected by testing: appear only in the presence of an attacker.
- Errors can have serious consequences.


infopackets
MS Windows, Tech News, and Freeware Daily

Google Wallet Payment System Vulnerable to Attack

by John Lister on 20120220 @ 01:04PM EST | google it | send to friends
Filed under Security | (related terms: phone , google wallet , access , code , thief)


The Register®

Hackers break SSL encryption used by millions of sites

Beware of BEAST decrypting secret PayPal cookies

By Dan Goodin in San Francisco • Get more from this author

Posted in ID, 19th September 2011 21:10 GMT

Models of protocols

Active attacker:

- The attacker can intercept all messages sent on the network
- He can compute messages
- He can send messages on the network

Models of protocols: the symbolic model

The symbolic model or "Dolev-Yao model" is due to Needham and Schroeder (1978) and Dolev and Yao (1983).

- Cryptographic primitives are blackboxes. sencrypt
- Messages are terms on these primitives. $\mathsf{sencrypt}(\mathit{Hello}, k)$
- The attacker is restricted to compute only using these primitives.
  ⇒ perfect cryptography assumption
  - So the definitions of primitives specify what the attacker can do.
    One can add equations between primitives.
    Hypothesis: the only equalities are those given by these equations.

This model makes automatic proofs relatively easy.

Models of protocols: the computational model

The computational model has been developped at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

- Messages are bitstrings. 01100100
- Cryptographic primitives are functions on bitstrings.

  $\mathrm{sencrypt}(011, 100100) = 111$
- The attacker is any probabilistic polynomial-time Turing machine.
  - The security assumptions on primitives specify what the attacker cannot do.

This model is much more realistic than the symbolic model, but proofs are mostly manual.

Models of protocols: side channels

The computational model is still just a model, which does not exactly match reality.

In particular, it ignores side channels:

- timing
- power consumption
- noise
- physical attacks against smart cards

which can give additional information.

Security properties: trace and equivalence properties

- Trace properties: properties that can be defined on a trace.
  - Symbolic model: they hold when they are true for all traces.
  - Computational model: they hold when they are true except for a set of traces of negligible probability.
- Equivalence (or indistinguishability) properties: the attacker cannot distinguish two protocols (with overwhelming probability)
  - Give compositional proofs.
  - Hard to prove in the symbolic model.

Security properties: secrecy

The attacker cannot obtain information on the secrets.

- Symbolic model:
  - (syntactic) secrecy: the attacker cannot obtain the secret (trace property)
  - strong secrecy: the attacker cannot distinguish when the value of the secrecy changes (equivalence property)
- Computational model: the attacker can distinguish the secret from a random number only with negligible probability (equivalence property)

Security properties: authentication

If A thinks she is talking to B, then B thinks he is talking to A, with the same protocol parameters.

- Symbolic model: formalized using correspondence assertions of the form "if some event has been executed, then some other events have been executed" (trace property).
- Computational model: matching conversations or session identifiers, which essentially require that the messages exchanged by A and B are the same up to negligible probability (trace property).

Verifying protocols in the symbolic model

Main idea (for most verifiers):

- Compute the knowledge of the attacker.

Difficulty: security protocols are infinite state.

- The attacker can create messages of unbounded size.
- Unbounded number of sessions of the protocol.

Verifying protocols in the symbolic model

Solutions:

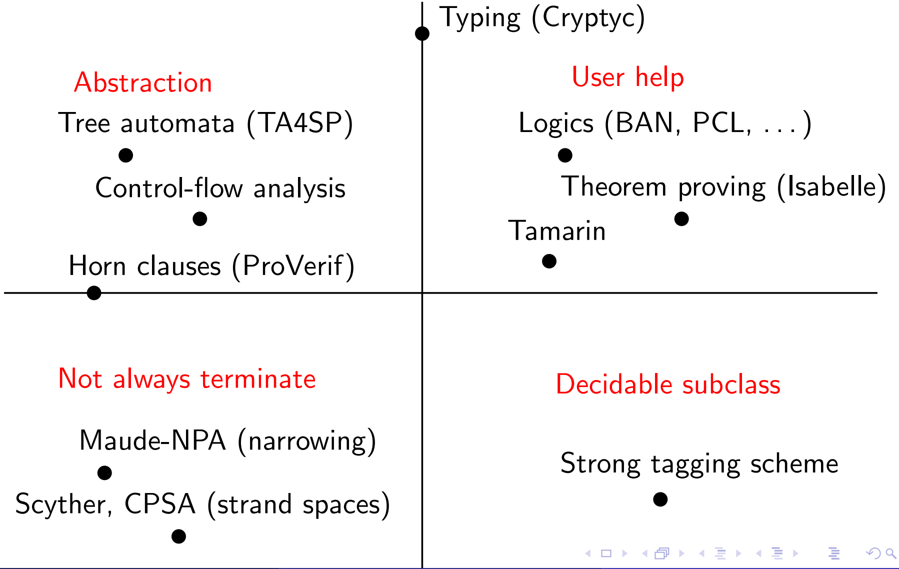
- Bound the state space arbitrarily:
  Trace properties: exhaustive exploration (model-checking: FDR, SATMC, . . .);
  find attacks but not prove security.
- Bound the number of sessions:
  - Trace properties: insecurity is NP-complete (with reasonable assumptions).
    OFMC, Cl-AtSe
  - Equivalence properties: AKISS, APTE, DeepSec, SAT-Equiv, SPEC
- Unbounded case:
  the problem is undecidable.

Solutions to undecidability

To solve an undecidable problem, we can

- Use approximations, abstraction.
- Not always terminate.
- Rely on user interaction or annotations.
- Consider a decidable subclass.

Solutions to undecidability



Typing (Cryptyc)

Abstraction

Tree automata (TA4SP)

Control-flow analysis

Horn clauses (ProVerif)

User help

Logics (BAN, PCL, . . .)

Theorem proving (Isabelle)

Tamarin

Not always terminate

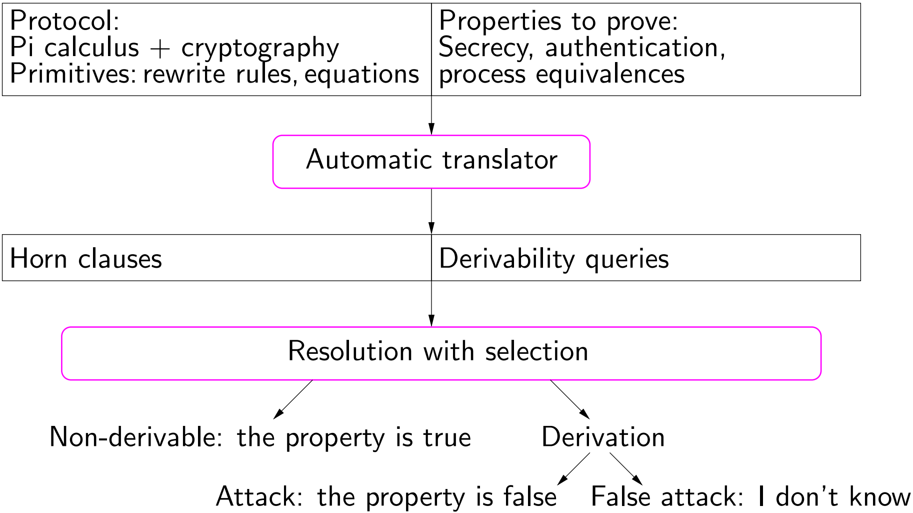
Maude-NPA (narrowing)

Scyther, CPSA (strand spaces)

Decidable subclass

Strong tagging scheme

ProVerif, https://proverif.inria.fr/



Protocol:
Pi calculus + cryptography
Primitives: rewrite rules, equations

Properties to prove:
Secrecy, authentication,
process equivalences

Automatic translator

Horn clauses

Derivability queries

Resolution with selection

Non-derivable: the property is true

Derivation

Attack: the property is false

False attack: I don't know

Features of ProVerif

- Fully automatic.
- Works for unbounded number of sessions and message space.
- Handles a wide range of cryptographic primitives, defined by rewrite rules or equations.
- Handles various security properties: secrecy, authentication, some equivalences.
- Does not always terminate and is not complete. In practice:
    - Efficient: small examples verified in less than 0.1 s;
      complex ones in a few minutes.
    - Very precise: no false attack in our tests on examples of the literature for secrecy and authentication.

Syntax of the process calculus

Pi calculus + cryptographic primitives

$M, N ::=$	terms
$x, y, z, \ldots$	variable
$a, b, c, s, \ldots$	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\textbf{out}(M, N); P$	output
$\textbf{in}(M, x : T); P$	input
0	nil process
$P \mid Q$	parallel composition
$!P$	replication
$\textbf{new}\ a : T; P$	restriction
$\textbf{let}\ x = g(M_1, \ldots, M_n)\ \textbf{in}\ P\ \textbf{else}\ Q$	destructor application
$\textbf{if}\ M = N\ \textbf{then}\ P\ \textbf{else}\ Q$	conditional

Constructors and destructors

Two kinds of operations:

- Constructors f are used to build terms $f(M_1, \dots, M_n)$

Example: Shared-key encryption $\mathrm{sencrypt}(M, N)$

fun sencrypt(bitstring, key) : bitstring.

- Destructors g manipulate terms
  let $x = g(M_1, \dots, M_n)$ **in** P **else** Q
  Destructors are defined by rewrite rules $g(M_1, \dots, M_n) \rightarrow M$.

Example: Decryption $\mathrm{sdecrypt}(\mathrm{sencrypt}(m, k), k) \rightarrow m$

fun sdecrypt(bitstring, key) : bitstring
reduc forall m : bitstring, k : key; $\mathrm{sdecrypt}(\mathrm{sencrypt}(m, k), k) = m$.

We represent in the same way public-key encryption, signatures, hash functions, ...

Example: The Denning-Sacco protocol (simplified)

Message 1. $A \to B : \quad \{\{k\}_{sk_A}\}_{pk_B} \quad k$ fresh

Message 2. $B \to A : \quad \{s\}_k$

$$
\begin{array}{ll}
 & \textbf{new } sk_A : \mathrm{sskey}; \textbf{new } sk_B : \mathrm{eskey}; \textbf{let } pk_A = \mathrm{spk}(sk_A) \textbf{ in} \\
 & \textbf{let } pk_B = \mathrm{pk}(sk_B) \textbf{ in out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & \quad ! \, \textbf{in}(c, x\_pk_B : \mathrm{epkey}); \textbf{new } k : \mathrm{key}; \\
 & \quad\ \ \textbf{out}(c, \mathrm{pencrypt}(\mathrm{sign}(k, sk_A), x\_pk_B)); \\
 & \quad\ \ \textbf{in}(c, x : \mathrm{bitstring}); \textbf{let } s = \mathrm{sdecrypt}(x, k) \textbf{ in } 0 \\
(B) & | \quad ! \, \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let } y' = \mathrm{pdecrypt}(y, sk_B) \textbf{ in} \\
 & \quad\ \ \textbf{let } k = \mathrm{checksign}(y', pk_A) \textbf{ in out}(c, \mathrm{sencrypt}(\mathrm{s}, k))
\end{array}
$$

The Horn clause representation

The first encoding of protocols in Horn clauses was given by Weidenbach (1999).

The main predicate used by the Horn clause representation of protocols is attacker:

$\text{attacker}(M)$ *means* *"the attacker may have M".*

We can model actions of the attacker and of the protocol participants thanks to this predicate.

Processes are automatically translated into Horn clauses (joint work with Martín Abadi).

Coding of primitives

- Constructors $f(M_1, \dots, M_n)$
  $\mathrm{attacker}(x_1) \wedge \dots \wedge \mathrm{attacker}(x_n) \to \mathrm{attacker}(f(x_1, \dots, x_n))$

Example: Shared-key encryption $\mathrm{sencrypt}(m, k)$

$\mathrm{attacker}(m) \wedge \mathrm{attacker}(k) \to \mathrm{attacker}(\mathrm{sencrypt}(m, k))$

- Destructors $g(M_1, \dots, M_n) \to M$
  $\mathrm{attacker}(M_1) \wedge \dots \wedge \mathrm{attacker}(M_n) \to \mathrm{attacker}(M)$

Example: Shared-key decryption $\mathrm{sdecrypt}(\mathrm{sencrypt}(m, k), k) \to m$

$\mathrm{attacker}(\mathrm{sencrypt}(m, k)) \wedge \mathrm{attacker}(k) \to \mathrm{attacker}(m)$

Coding of a protocol

If a principal A has received the messages $M_1, \dots, M_n$ and sends the message M,

$$\mathrm{attacker}(M_1) \wedge \dots \wedge \mathrm{attacker}(M_n) \rightarrow \mathrm{attacker}(M).$$

Example

Upon receipt of a message of the form $\mathrm{pencrypt}(\mathrm{sign}(y, sk_A), pk_B)$, B replies with $\mathrm{sencrypt}(\mathrm{s}, y)$:

$$\mathrm{attacker}(\mathrm{pencrypt}(\mathrm{sign}(y, sk_A), pk_B)) \rightarrow \mathrm{attacker}(\mathrm{sencrypt}(\mathrm{s}, y))$$

The attacker sends $\mathrm{pencrypt}(\mathrm{sign}(y, sk_A), pk_B)$ to B, and intercepts his reply $\mathrm{sencrypt}(\mathrm{s}, y)$.

Proof of secrecy

Theorem (Secrecy)

If attacker(M) *cannot be derived from the clauses, then M is secret.*

The term M cannot be built by an attacker.

The resolution algorithm will determine whether a given fact can be derived from the clauses.

Example

query attacker(s).

Resolution with free selection

$$\frac{R = H \to F \qquad R' = F'_1 \wedge H' \to F'}{\sigma H \wedge \sigma H' \to \sigma F'}$$

where σ is the most general unifier of F and F'_1,
F and F'_1 are selected.

The selection function selects:

- a hypothesis not of the form attacker(x) if possible,
- the conclusion otherwise.

Key idea: avoid resolving on facts attacker(x).

Resolve until a fixpoint is reached.
Keep clauses whose conclusion is selected.

Theorem

The obtained clauses derive the same facts as the initial clauses.

Other security properties (1)

Correspondence assertions:

If an event has been executed, then some other events must have been executed.

$$\textbf{new } sk_A : \mathrm{sskey}; \textbf{new } sk_B : \mathrm{eskey}; \textbf{let } pk_A = \mathrm{spk}(sk_A) \textbf{ in}$$
$$\textbf{let } pk_B = \mathrm{pk}(sk_B) \textbf{ in } \textbf{out}(c, pk_A); \textbf{out}(c, pk_B);$$

(A) $! \textbf{in}(c, x\_pk_B : \mathrm{epkey}); \textbf{new } k : \mathrm{key}; \textbf{event } eA(pk_A, x\_pk_B, k);$
$\textbf{out}(c, \mathrm{pencrypt}(\mathrm{sign}(k, sk_A), x\_pk_B));$
$\textbf{in}(c, x : \mathrm{bitstring}); \textbf{let } s = \mathrm{sdecrypt}(x, k) \textbf{ in } 0$

(B) $\mid$ $! \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let } y' = \mathrm{pdecrypt}(y, sk_B) \textbf{ in}$
$\textbf{let } k = \mathrm{checksign}(y', pk_A) \textbf{ in } \textbf{event } eB(pk_A, pk_B, k);$
$\textbf{out}(c, \mathrm{sencrypt}(\mathrm{s}, k))$

$$\textbf{query } x : \mathrm{spkey}, y : \mathrm{epkey}, z : \mathrm{key}; \textbf{event}(eB(x, y, z)) \Longrightarrow \textbf{event}(eA(x, y, z))$$

Other security properties (2)

Process equivalences

- Strong secrecy
- Equivalences between processes that differ only by terms they contain (joint work with Martín Abadi and Cédric Fournet)

  In particular, proof of protocols relying on weak secrets.

Sound approximations

- Main approximation = repetitions of actions are ignored: the clauses can be applied any number of times.
- In **out**$(M, N).P$, the Horn clause model considers that P can always be executed.

These approximations can cause (rare) false attacks.

We have built an algorithm that reconstructs attacks from derivations from Horn clauses, when the derivation corresponds to an attack (with Xavier Allamigeon).

Results (1)

- Tested on many protocols of the literature.
- More ambitious case studies:
  - Certified email (with Martín Abadi)
  - JFK (with Martín Abadi and Cédric Fournet)
  - Plutus (with Avik Chaudhuri)
  - Signal (with Karthikeyan Bhargavan and Nadim Kobeissi)
  - TLS 1.3 (with Karthikeyan Bhargavan and Nadim Kobeissi)
  - ARINC823 avionic protocols
- Case studies by others:
  - E-voting protocols (Delaune, Kremer, and Ryan; Backes et al)
  - Zero-knowledge protocols, DAA (Backes et al)
  - Shared authorisation data in TCG TPM (Chen and Ryan)
  - Electronic cash (Luo et al)
  - Google 2-step and FIDO U2F (Jacomme and Kremer)
  - Noise (Kobeissi, Nicolas, and Bhargavan)
  - ...

Results (2)

- Extensions and tools:
  - Extension to XOR and (improved) Diffie-Hellman (Küsters and Truderung)
  - Web service verifier TulaFale (Microsoft Research).
  - Support for state (StatVerif, GSVerif)
  - Support for sets (AIF-Omega, Set-pi)
  - Unlinkability and anonymity (Ukano)
  - Verification of implementations (FS2PV, Spi2Java).
  - . . .

Verifying protocols in the computational model

1. Linking the symbolic and the computational models
2. Adapting techniques from the symbolic model
3. Direct computational proofs

Linking the symbolic and the computational models

- Computational soundness theorems:

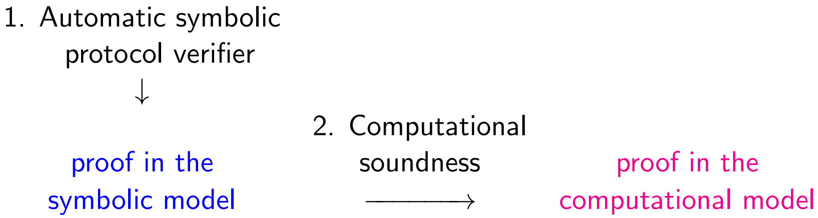
  Secure in the symbolic model ⇒ secure in the computational model

  modulo additional assumptions.

  Approach pioneered by Abadi & Rogaway [2000]; many works since then.

Linking the symbolic and the computational models: application

- Indirect approach to automating computational proofs:



1. Automatic symbolic protocol verifier

proof in the symbolic model

2. Computational soundness

proof in the computational model

Various approaches

- Trace mapping [Micciancio & Warinschi 2004], followed by others

  Computational trace $\mapsto$ symbolic trace

  up to negligible probability.

  - computational soundness for trace properties (authentication), for public-key encryption, signatures, hash functions, . . .
  - computational soundness for observational equivalence [Comon-Lundh & Cortier 2008]
  - modular computational soundness proofs.
- Backes-Pfitzmann-Waidner library
- UC-based approach [Canetti & Herzog 2006]

Advantages and limitations

- + symbolic proofs easier to automate
- + reuse of existing symbolic verifiers
- − additional hypotheses:
  - − strong cryptographic primitives
  - − length-hiding encryption or modify the symbolic model
  - − honest keys
  - − no key cycles
- Going through the symbolic model is a detour

Adapting techniques from the symbolic model

Some symbolic techniques can also be adapted to the computational model:

- Logics: computational PCL, CIL, Bana-Comon
- Type systems: computationally sound type system
  - Well-typed ⇒ secure in the computational model

The Bana-Comon logic

Usually:

- in symbolic models, we specify what the attacker can do, e.g. apply encryption, decryption, signatures, ...
- in computational models, we specify what the attacker cannot do, e.g. cannot distinguish two ciphertexts, cannot forge signatures, ...

$\Rightarrow$ difficult to get computational soundness.

Main idea of the Bana-Comon logic: design a new symbolic model, in which we specify what the attacker cannot do (through axioms).

We get a computational proof using symbolic methods.

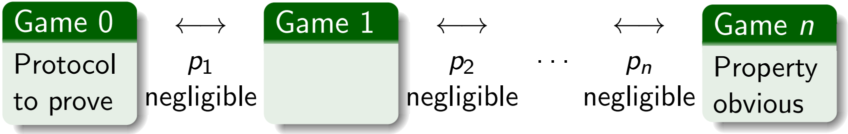
Limitation: number of sessions bounded independently of the security parameter.

Tool Squirrel `https://squirrel-prover.github.io/`

Direct computational proofs

Proofs in the computational model are typically proofs by sequences of games [Shoup, Bellare & Rogaway]:

- The first game is the real protocol.
- One goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property is obvious from the form of the game. (The advantage of the adversary is 0 for this game.)



Mechanizing proofs by sequences of games (1)

CryptoVerif, `https://cryptoverif.inria.fr`

- generates proofs by sequences of games.
- proves secrecy, correspondence, and indistinguishability properties.
- provides a generic method for specifying properties of many cryptographic primitives.
- works for N sessions (polynomial in the security parameter), with an active attacker.
- gives a bound on the probability of an attack (exact security).
- automatic and user-guided modes.

VeriCrypt'20 tutorial at `https://cryptoverif.inria.fr/tutorial`

Similar tool by Tšahhirov and Laud [2007], using a different game representation (dependency graph).

CryptoVerif: Case studies

- Many small protocols of the literature.
- Full domain hash signature (with David Pointcheval)
  Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay)
- OEKE (variant of Encrypted Key Exchange)
- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA)
- SSH Transport Layer Protocol (with David Cadé)
- ARINC823 avionic protocols
- Signal (with Nadim Kobeissi and Karthikeyan Bhargavan)
- TLS 1.3 draft 18 (with Karthikeyan Bhargavan and Nadim Kobeissi)
- WireGuard (with Benjamin Lipp and Karthikeyan Bhargavan)

Mechanizing proofs by sequences of games (2)

EasyCrypt, `https://github.com/EasyCrypt/easycrypt`

- The user gives the games and a proof of their indistinguishability. The tool verifies this proof.
- Can do more subtle reasoning than CryptoVerif, but is less automated.
- Better suited for proving primitives than CryptoVerif. The automation of CryptoVerif helps for protocols.
- Successor of CertiCrypt

EasyCrypt: Case studies

- Cramer-Shoup, hashed ElGamal
- MEE-CBC
- ChaChaPoly
- ZAEP
- RSA–PSS Provably Secure against Non-random Faults
- One-round authenticated key exchange protocols (Naxos, Nets, . . .)
- Pairing-based cryptography
- SHA-3
- Amazon Web Services Key Management Service
- . . .

Other computational tools

- FCF (Foundation Cryptography Framework): library over Coq
- CryptHOL: framework over Isabelle
- Specialized tools:
  - AutoG&P: pairing-based schemes
  - ZooCrypt: padding-based public-key encryption schemes

Conclusion

- Very active research area
- Many different tools:
  - symbolic tools
    - + mature
    - + easier to use than computational tools
    - + find attacks and security proofs
    - − may miss computational attacks
  - computational tools
    - − more delicate to use
      − require more guidance from the user
    - + provide stronger security proofs
    - − do not find attacks
- Collaborations between tools