Security Protocol Verification: Symbolic and Computational Models

Bruno Blanchet

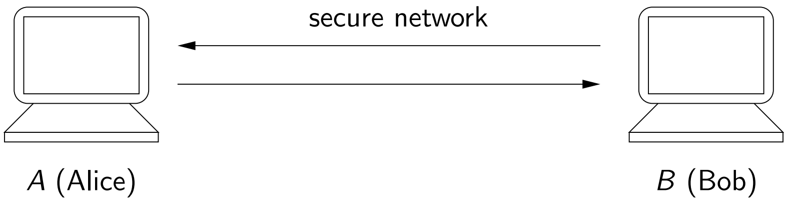
INRIA, École Normale Supérieure, CNRS
Bruno.Blanchet@ens.fr

March 2012

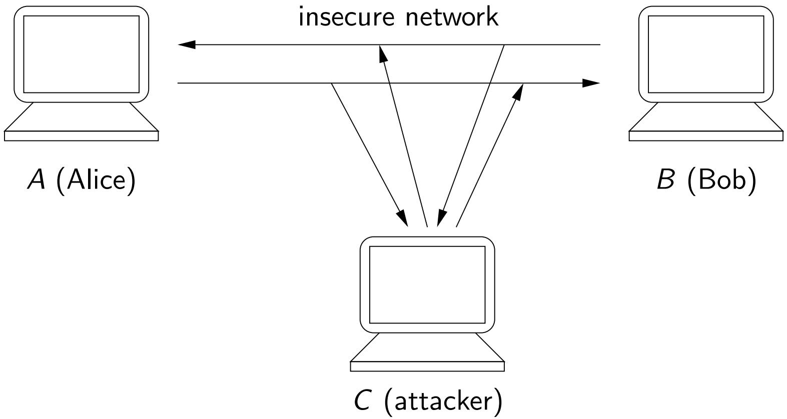
Outline

1. Introduction to security protocols
2. Verifying protocols in the symbolic model
3. Verifying protocols in the computational model
4. Verifying protocol implementations
5. 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)

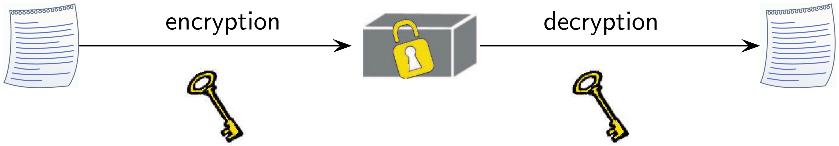
Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Cryptographic primitives

Definition (Cryptographic primitives)

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

Shared-key encryption

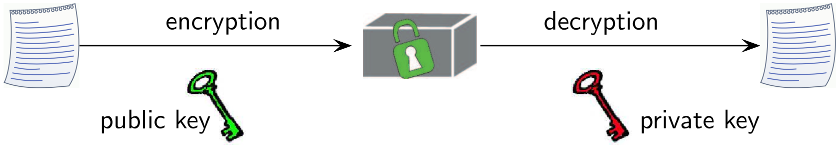


Cryptographic primitives

Definition (Cryptographic primitives)

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

Public-key encryption

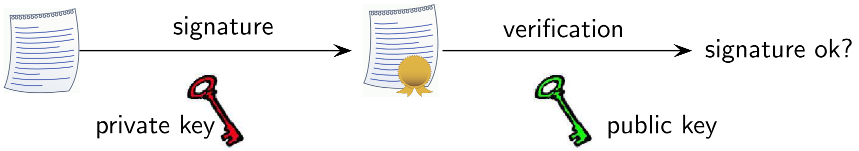


Cryptographic primitives

Definition (Cryptographic primitives)

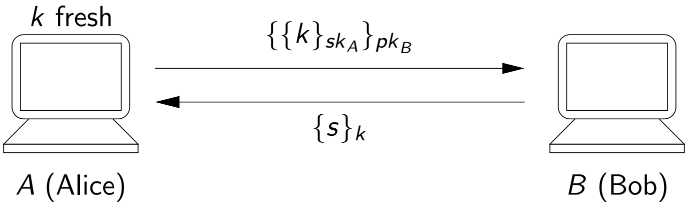
Basic cryptographic algorithms, used as building blocks for protocols, e.g. encryption and signatures.

Signatures



Example

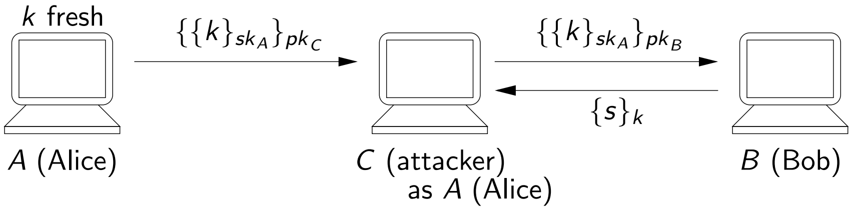
Denning-Sacco key distribution protocol [Denning, Sacco, 1981] (simplified)



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.

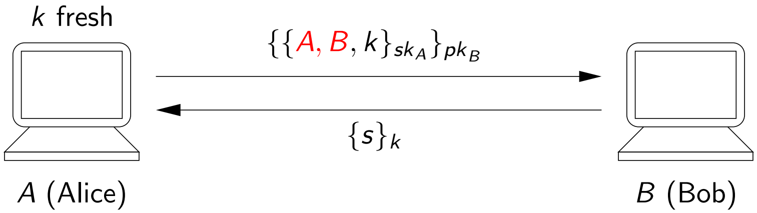
The attack

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



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

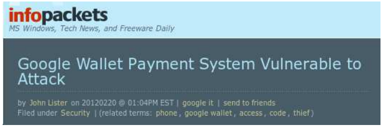
Many protocols exist, for various goals:

- 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 ?

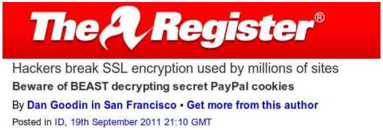
The verification of security protocols has been and is still a very active research area.

- 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. sencrypt($Hello, k$)
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ 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 until recently proofs were only 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:

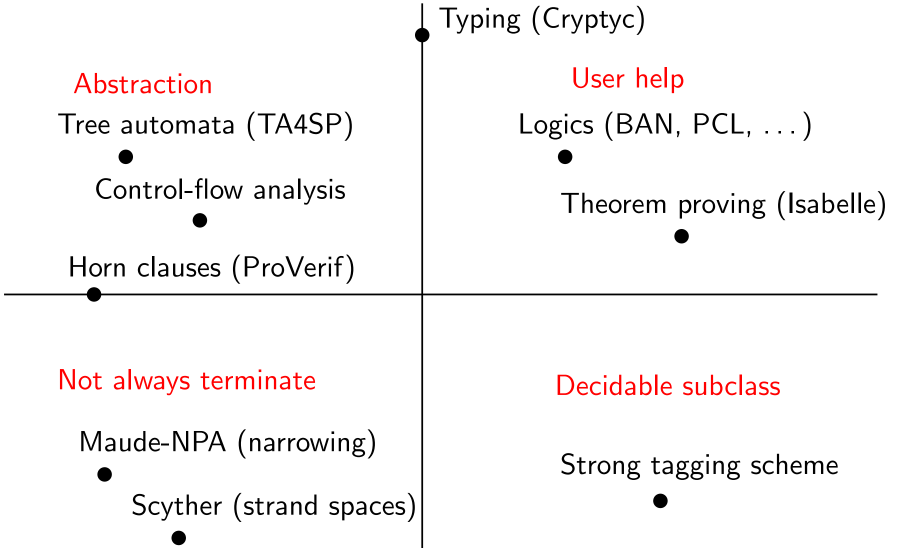
- Bound the state space arbitrarily:
  Trace properties: exhaustive exploration (model-checking: FDR, SATMC, . . .);
  find attacks but not prove security.
- Bound the number of sessions:
  - Trace properties: insecurity is NP-complete (with reasonable assumptions).
    OFMC, Cl-AtSe
  - Equivalence properties: a few recent decision procedures and tools, e.g. [Cheval et al, CCS 2011], [Chadha et al, ESOP 2012]
- Unbounded case:
  the problem is undecidable.

Solutions to undecidability

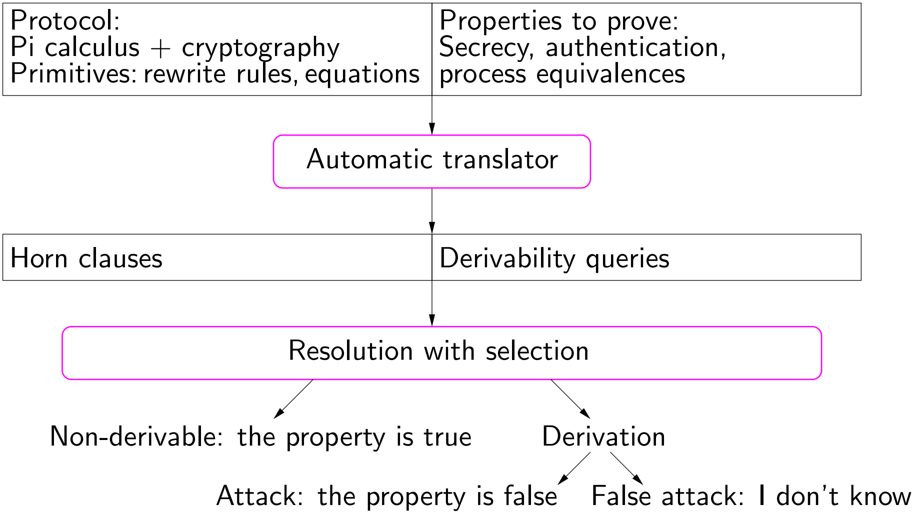
To solve an undecidable problem, we can

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

Solutions to undecidability



ProVerif


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 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
$\overline{M}\langle N \rangle.P$	output
$M(x).P$	input
0	nil process
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	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 $\mathrm{sencrypt}(M, N)$.

- 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 $\mathrm{sdecrypt}(M', N)$: $\mathrm{sdecrypt}(\mathrm{sencrypt}(m, k), k) \rightarrow m$.

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

Example: The Denning-Sacco protocol (simplified)

$$\begin{array}{llll} \text{Message 1.} & A \to B: & \{\{k\}_{sk_A}\}_{pk_B} & k \text{ fresh} \\ \text{Message 2.} & B \to A: & \{s\}_k & \end{array}$$

$(\nu sk_A)(\nu sk_B)\textbf{let } pk_A = \text{pk}(sk_A) \textbf{ in let } pk_B = \text{pk}(sk_B) \textbf{ in}$
$\overline{c}\langle pk_A\rangle.\overline{c}\langle pk_B\rangle.$

(A) $! \; c(x\_pk_B).(\nu k)\overline{c}\langle \text{pencrypt}(\text{sign}(k, sk_A), x\_pk_B)\rangle.$
$c(x).\textbf{let } s = \text{sdecrypt}(x, k) \textbf{ in } 0$

(B) $| \quad ! \; c(y).\textbf{let } y' = \text{pdecrypt}(y, sk_B) \textbf{ in}$
$\textbf{let } k = \text{checksign}(y', pk_A) \textbf{ in } \overline{c}\langle \text{sencrypt}(s, k)\rangle$

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:

$\mathsf{attacker}(M)$ *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 $f(M_1, \dots, M_n)$
  $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \rightarrow \text{attacker}(f(x_1, \dots, x_n))$

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

$\text{attacker}(m) \wedge \text{attacker}(k) \rightarrow \text{attacker}(\text{sencrypt}(m, k))$

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

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

$\text{attacker}(\text{sencrypt}(m, k)) \wedge \text{attacker}(k) \rightarrow \text{attacker}(m)$

Coding of a protocol

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

$$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \rightarrow \text{attacker}(M).$$

Example

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

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

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

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.

Remark: Soundness and completeness are swapped.
The resolution prover is complete
(If attacker(M) is derivable, it finds a derivation.)
$\Rightarrow$ The protocol verifier is sound
(If it proves secrecy, then secrecy is true.)

Resolution with free selection

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

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

- Correspondence assertions:
  If an event has been executed, then some other events must have been executed.
- 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 $\overline{M}\langle N\rangle.P$, 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

- 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)
- 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)
  - ...
- Extensions and tools:
  - Extension to XOR and Diffie-Hellman (Küsters and Truderung)
  - Web service verifier TulaFale (Microsoft Research).
  - Translation from HLPSL, input language of AVISPA (Gotsman, Massacci, Pistore)
  - 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 $\Rightarrow$ 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:

$$\begin{tikzcd}
\text{1. Automatic symbolic protocol verifier} \arrow[d] & \\
\text{proof in the symbolic model} \arrow[r, "\text{2. Computational soundness}"] & \text{proof in the computational model}
\end{tikzcd}$$

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 [but see Comon-Lundh et al, POST 2012]
  - − no key cycles
- Going through the symbolic model is a detour

- An attempt to solve these problems:
  symbolic model in which we specify what the attacker cannot do
  [Bana & Comon-Lundh, POST 2012]

Adapting techniques from the symbolic model

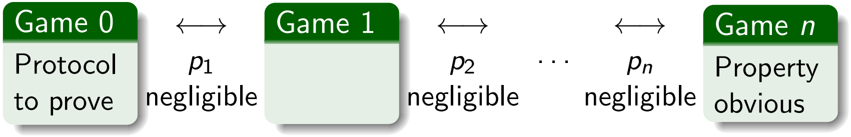
Some symbolic techniques can also be adapted to the computational model:

- Logics: computational PCL, CIL
- Type systems: computationally sound type system
  - Well-typed $\Rightarrow$ secure in the computational model

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.)


Game 0
Protocol to prove
$\longleftrightarrow$ p_1 negligible
Game 1
$\longleftrightarrow$ p_2 negligible
$\cdots$
$\longleftrightarrow$ p_n negligible
Game n
Property obvious

Mechanizing proofs by sequences of games (1)

CryptoVerif, `www.cryptoverif.ens.fr`

- generates proofs by sequences of games.
- proves secrecy and correspondence 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.

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

Mechanizing proofs by sequences of games (2)

CertiCrypt, `http://software.imdea.org/~szanella/`

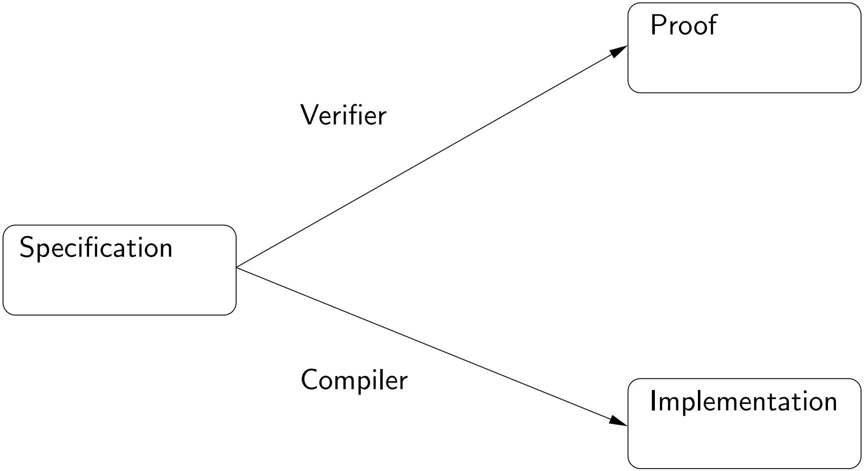
- Machine-checked cryptographic proofs in Coq
- Interesting case studies, *e.g.* OAEP
- Good for proving primitives: can prove complex mathematical theorems
- Requires much human effort
- Improved by EasyCrypt: generates CertiCrypt proofs from proof sketches (sequence of games and hints)

Idea also followed by Nowak et al.

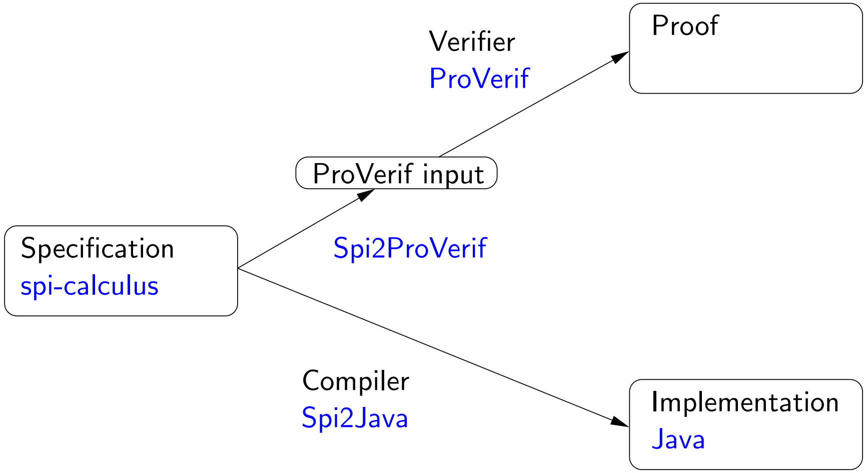
Verifying protocol implementations

- Errors may appear in the protocol implementation, even if the specification is secure.
- $\Rightarrow$ one needs to prove the implementation itself, not only the specification.
- Proving implementations is more difficult.
    - Handle a full programming language.

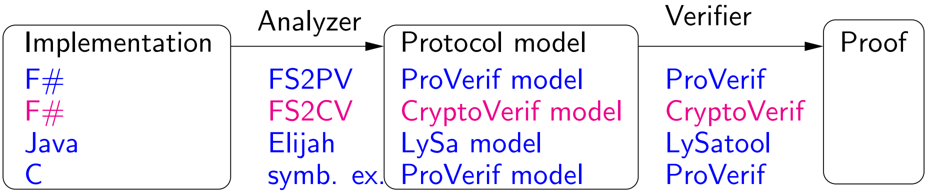
Generating implementations from specifications



Generating implementations from specifications: Spi2Java



Extracting specifications from implementations



Adapted and new methods

- CSur:

$$\begin{tikzcd}\boxed{\text{C implementation}} \arrow[r, "\text{CSur analyzer}"] & \boxed{\text{Horn clauses}} \arrow[r, "\mathcal{H}_1"] & \boxed{\text{Proof}}\end{tikzcd}$$

- F7/F*: typing F# implementations
- Computational F7: use typing to test whether game transformations are applicable.
- ASPIER: verify C implementations by model-checking
- Dupressoir et al [CSF'11] use the general purpose C verifier VCC to prove memory safety and security.

Conclusion and future challenges

- Very active research area
- Progress in all directions:
    - symbolic model
    - computational model
    - implementations

Conclusion and future challenges

- Very active research area
- Progress in all directions:
  - symbolic model: fairly mature
  - computational model: much work to do
  - implementations: much work to do
- Physical attacks: only the beginning.