

# Automatic Verification of Security Protocols: ProVerif and CryptoVerif

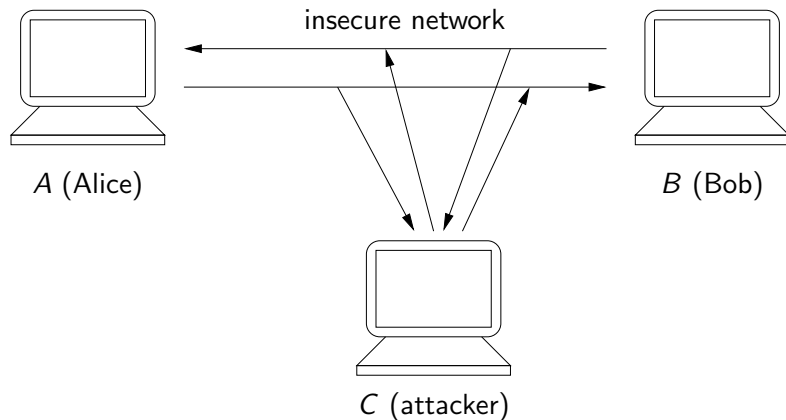
Bruno Blanchet

Google

On leave from Inria, Paris  
`bruno.blanchet@inria.fr`

May 2016

# Communications over an **insecure** network



*A* talks to *B* on an insecure network

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

# 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 are **not detected** by testing:  
they appear only in the presence of an adversary.
- Errors can have **serious consequences**.

# Attacks against TLS


MI TLS mTLS 3SHAKE SMACK VHC

## Triple Handshakes Consider Breaking and Fixing Authn

March 4, 2014

Introduction	TLS Weaknesses	Triple Handshakes
Countermeasures	Disclosure	Other Attacks

### SMACK: State Machine Attacks



Implementations of the Transport Layer Security (TLS) protocol handle a variety of protocol versions, modes and key exchange methods, and prescribe a different message sequence for each. We address the problem of designing a robust, compact state machine that can correctly multiplex these modes.

Slides from the TLS WG session at IETF89 and our proposed Internet-Draft, [Our research paper with more details on the attacks \(see Sections 4-7\)](#), May, 2014.

### The Logjam Attack

THE WALL STREET JOURNAL

Home World U.S. Asia Europe Business Tech Health Opinion Arts Real Estate

Fix for Logjam bug could make more than 20,000 websites unreachable

### New Computer Bug Exposes Broad Security Flaws



## Tracking the FREAK Attack

Good News! Your browser appears to be safe from the FREAK attack.

On Tuesday, March 3, 2015, researchers announced a new SSL/TLS vulnerability called the FREAK attack. It allows an attacker to intercept HTTPS connections between vulnerable clients and servers and force them to use weakened encryption. Tracking the impact of the FREAK attack is dedicated to

The FREAK attack disclosure was coordinated by Michigan, including a team that can be contacted for additional details at this Washington

ars technica

MAIN MENU MY STORIES: 24 FORUMS SUBSCRIBE JOBS ARS CONSENT

Arx Technica has arrived in Europe. Check it out!

### RISK ASSESSMENT / SECURITY & HACKTIVISM

### "FREAK" flaw in Android and Apple devices cripples HTTPS crypto protection

Bug forces millions of sites to use easily breakable key once thought to be dead.

by Dan Goodin - Mar 3, 2015 10:07pm CET



Warning! Your web browser is vulnerable to Logjam and can be tricked into using weak encryption. Update your browser.

Diffie-Hellman key exchange is a popular cryptographic algorithm that allows Internet protocols to share key and negotiate a secure connection. It is fundamental to many protocols including HTTP, SMTPS, and protocols that rely on TLS.

We have uncovered several weaknesses in how Diffie-Hellman key exchange has been deployed

### The BEAST Wins Again: Web Security

#### Documents

- [PDF of slides](#)
- [summary of briefing for non-experts](#)
- [Paper: Virtual Host Conf](#)
- [Paper: Triple Handshakes](#)

#### Exploit videos

Disclaimer: The goal of these videos is to have a strong impact. The attack is not a new discovery. We are happy to acknowledge it and Mozilla. We are also thank

ars technica

MAIN MENU MY STORIES: 24 FORUMS SUBSCRIBE JOBS ARS CONSENT

Arx Technica has arrived in Europe. Check it out!

### RISK ASSESSMENT / SECURITY & HACKTIVISM

### HTTPS-cripping attack threatens tens of thousands of Web and mail servers

Diffie-Hellman downgrade weakness allows attackers to intercept encrypted data.

by Dan Goodin - Mar 2, 2015 7:58am CET



### FREAK Attack Threatens SSL Clients

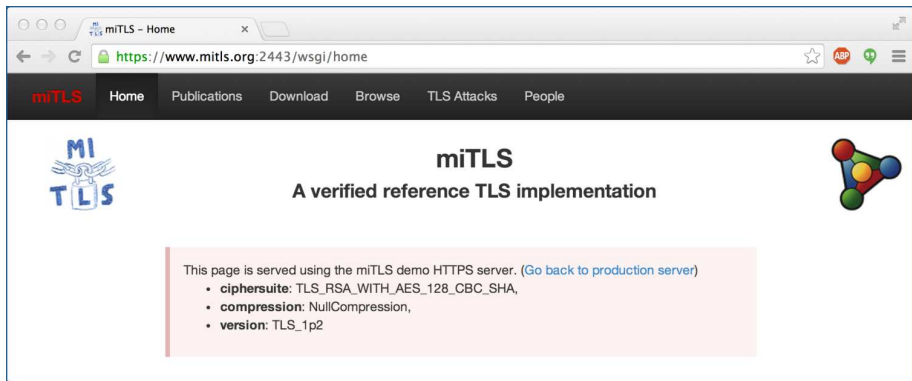
Posted by [Soulskill](#) on Tuesday March 03, 2015 @04:29PM from the another-day-another-vuln dept.

[msm1267](#) writes:

For the nth time in the last couple of years, security experts are warning about [a new Internet-scale vulnerability, this time in some popular SSL clients](#). The flaw [allows an attacker to force clients to downgrade to weakened ciphers](#) and break their supposedly encrypted communications through a man-in-the-middle attack. Researchers recently discovered that some SSL clients, including OpenSSL, will accept weak RSA keys—known as export-grade keys—without asking for those keys. Export-grade refers to 512-bit RSA keys, the key strength that was approved by the United States government for export overseas. This was an artifact from decades ago

# miTLS, <http://www.mitls.org/>

- Formally verified reference implementation of TLS 1.2 in F7/F\* (working towards TLS 1.3)
- Written from scratch focusing on verification



The screenshot shows a web browser window with the address bar displaying `https://www.mitls.org:2443/wsgi/home`. The page features a navigation menu with links for Home, Publications, Download, Browse, TLS Attacks, and People. The main content area includes the miTLS logo (stylized 'MI' above 'TLS' with a key icon), the title 'miTLS', and the subtitle 'A verified reference TLS implementation'. A pink box contains the following text:

This page is served using the miTLS demo HTTPS server. ([Go back to production server](#))

- **ciphersuite:** TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA,
- **compression:** NullCompression,
- **version:** TLS\_1p2

# 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.  
 ⇒ **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 developed at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

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

$$\text{sencrypt}(011, 100100) = 111$$
- 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.

# 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:  
exhaustive exploration (model-checking: FDR, SATMC, ...);  
find attacks but not prove security.
- Bound the number of sessions: insecurity is **NP-complete** (with reasonable assumptions).  
OFMC, CI-AtSe
- 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 (strand spaces)

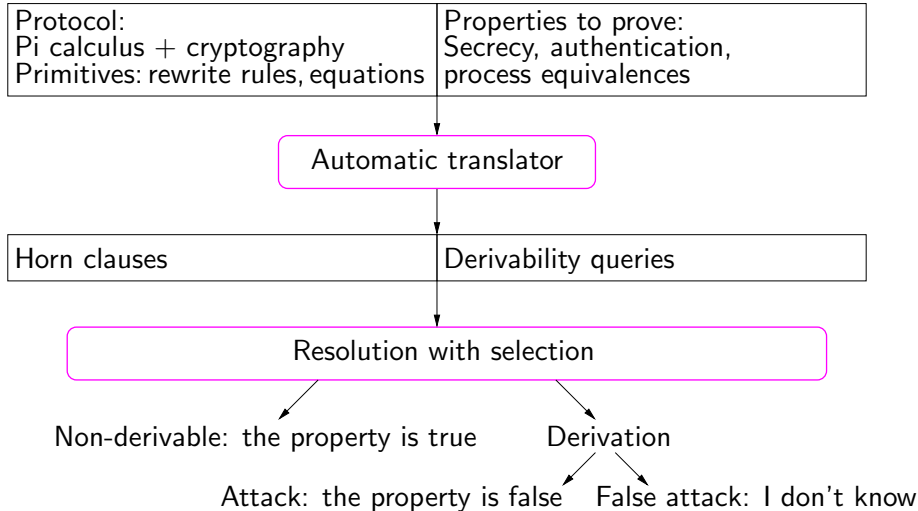
Decidable subclass

Strong tagging scheme

# ProVerif, <http://proverif.inria.fr>

- **Symbolic** security protocol verifier.
- **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 from a few minutes to hours.
  - **Very precise**: no false attack in 19 protocols of the literature tested for secrecy and authentication.

# ProVerif





# Syntax of the process calculus

## Pi calculus + cryptographic primitives

$M, N ::=$

$x, y, z, \dots$

$a, b, c, s, \dots$

$f(M_1, \dots, M_n)$

$P, Q ::=$

**out**( $M, N$ );  $P$

**in**( $M, x$ );  $P$

**let**  $x = g(M_1, \dots, M_n)$  **in**  $P$  **else**  $Q$

**if**  $M = N$  **then**  $P$  **else**  $Q$

$0$

$P \mid Q$

$!P$

**new**  $a$ ;  $P$

terms

variable

name

constructor application

processes

output

input

destructor application

conditional

nil process

parallel composition

replication

restriction

# Constructors and destructors

Two kinds of operations:

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

## Example

Shared-key encryption  $\text{sencrypt}(M, N)$ .

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

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

# Example: The Denning-Sacco protocol (simplified)

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

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

**new**  $sk_A$ ; **new**  $sk_B$ ; **let**  $pk_A = pk(sk_A)$  **in** **let**  $pk_B = pk(sk_B)$  **in**  
**out**( $c, pk_A$ ); **out**( $c, pk_B$ );

(A)     ! **in**( $c, x_{-pk_B}$ ); **new**  $k$ ; **out**( $c, \text{pencrypt}(\text{sign}(k, sk_A), x_{-pk_B})$ ).  
           **in**( $c, x$ ); **let**  $s = \text{sdecrypt}(x, k)$  **in** 0

(B)     | ! **in**( $c, y$ ); **let**  $y' = \text{pdecrypt}(y, sk_B)$  **in**  
           **let**  $k = \text{checksign}(y', pk_A)$  **in** **out**( $c, \text{sencrypt}(s, k)$ )

# 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`:

`attacker( $M$ )` means “the attacker may have  $M$ ”.

We can model actions of the adversary 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  $M_1, \dots, M_n$  and sends the message  $M$ ,

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

## Example

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

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

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

# Proof of secrecy

## Theorem (Secrecy)

If  $\text{attacker}(M)$  *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.

# Other security properties

- **Correspondence assertions:** (authentication)  
If an event has been executed, then some other events must have been executed.
- **Process equivalences:** the adversary cannot distinguish between two processes.
  - **Strong secrecy:** the adversary cannot see when the value of the secret changes.
  - 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.



# Demo

## Demo

### Denning-Sacco example

# Applications

## 1 Case studies:

- 19 protocols of the literature
- Certified email (with Martín Abadi)
- JFK (with Martín Abadi and Cédric Fournet)
- Plutus (with Avik Chaudhuri)
- Avionic protocols (ARINC 823)

### 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)
- ...

## 2 Extensions

## 3 ProVerif as back-end

# Applications

- ① Case studies
- ② Extensions:
  - Extensions to **XOR** and **Diffie-Hellman** (Küsters and Truderung), to **bilinear pairings** (Pankova and Laud)
  - StatVerif: extension to **mutable state** (Arapinis et al)
  - Set-Pi: extension to **sets with revocation** (Bruni et al)
- ③ ProVerif as back-end

# Applications

- 1 Case studies
- 2 Extensions
- 3 ProVerif as back-end:
  - TulaFale: **Web service** verifier (Bhargavan et al)
  - FS2PV: **F#** to ProVerif, applied to TLS and TPM (Bhargavan et al)
  - JavaSpi: **Java** to ProVerif (Avalle et al)
  - Web-spi: **web** security mechanisms (Bansal et al)

# 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 papers since then.

# Linking the symbolic and the computational models: application

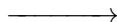
- **Indirect approach** to automating computational proofs:

1. Automatic symbolic  
protocol verifier



proof in the  
symbolic model

2. Computational  
soundness



proof in the  
computational model

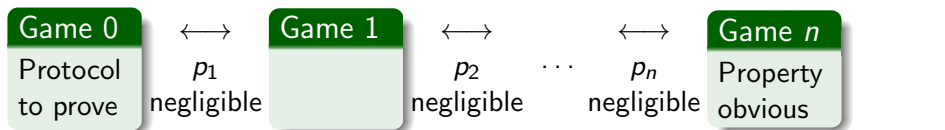
# Advantages and limitations

- + 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]

# Direct computational proofs

Following Shoup and Bellare&Rogaway, the proof is typically a sequence of games:

- 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 usually 0 for this game.)





# Mechanizing proofs by sequences of games

CertiCrypt, <http://certicrypt.gforge.inria.fr/>

- 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

EasyCrypt, <https://www.easycrypt.info/trac/>:

- Successor of CertiCrypt
- Less human effort: give games and hints on how to prove indistinguishability
- Relies on SMT solvers

Idea also followed by Nowak et al.

# CryptoVerif, <http://cryptoverif.inria.fr>

- **Computational** security protocol verifier.
- Proves **secrecy** and **correspondence** properties.
- Provides a **generic** method for specifying properties of **cryptographic primitives**, which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, ...
- Works for  **$N$  sessions** (polynomial in the security parameter).
- Gives a bound on the **probability** of an attack (exact security).
- Has **automatic** and **manually guided** modes.
- Can generate **OCaml implementations** (joint work with David Cadé).

# Process calculus for games

Games are formalized in a **process calculus**:

- It is adapted from the pi calculus.
- The semantics is **purely probabilistic** (no non-determinism).
- All processes run in **polynomial time**:
  - polynomial number of copies of processes,
  - length of messages on channels bounded by polynomials.

This calculus is inspired by:

- the calculus of [Lincoln, Mitchell, Mitchell, Scedrov, 1998],
- the calculus of [Laud, 2005].

# Example

$$A \rightarrow B : e = \{x'_k\}_{x_k}, \text{mac}(e, x_{mk}) \quad x'_k \text{ fresh}$$

$$Q_0 = \mathbf{in}(\text{start}, ()); \mathbf{new} \ x_r : \text{keyseed}; \mathbf{let} \ x_k : \text{key} = \text{kgen}(x_r) \mathbf{in}$$

$$\mathbf{new} \ x'_r : \text{mkeyseed}; \mathbf{let} \ x_{mk} : \text{mkey} = \text{mkgen}(x'_r) \mathbf{in} \mathbf{out}(c, ());$$

$$(Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} \mathbf{in}(c_A, ()); \mathbf{new} \ x'_k : \text{key}; \mathbf{new} \ x''_r : \text{coins};$$

$$\mathbf{let} \ x_m : \text{bitstring} = \text{enc}(k2b(x'_k), x_k, x''_r) \mathbf{in}$$

$$\mathbf{out}(c_A, x_m, \text{mac}(x_m, x_{mk}))$$

$$Q_B = !^{i' \leq n} \mathbf{in}(c_B, x'_m : \text{bitstring}, x_{ma} : \text{macstring});$$

$$\mathbf{if} \ \text{check}(x'_m, x_{mk}, x_{ma}) \ \mathbf{then}$$

$$\mathbf{let} \ i_{\perp}(k2b(x''_k)) = \text{dec}(x'_m, x_k) \ \mathbf{in} \ \mathbf{out}(c_B, ())$$

# Arrays

The variables defined in repeated processes (under a replication) are **arrays**, with one cell for each execution, to remember the values used in each execution.

These arrays are indexed with the execution number  $i$ ,  $i'$ .

$$Q_A = !^{i \leq n} \mathbf{in}(c_A, ()); \mathbf{new } x'_k[i] : \mathit{key}; \mathbf{new } x''_r[i] : \mathit{coins};$$

$$\mathbf{let } x_m[i] : \mathit{bitstring} = \mathit{enc}(k2b(x'_k[i]), x_k, x''_r[i]) \mathbf{in}$$

$$\mathbf{out}(c_A, x_m[i], \mathit{mac}(x_m[i], x_{mk}))$$

Arrays replace lists generally used by cryptographers.

They avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.

# Indistinguishability as observational equivalence

Two processes (games)  $Q_1$ ,  $Q_2$  are **observationally equivalent** when the adversary has a negligible probability of distinguishing them:

$$Q_1 \approx Q_2$$

The adversary is represented by an acceptable evaluation context  $C$  (essentially, a process put in parallel with the considered games).

- Observational equivalence is an equivalence relation.
- It is **contextual**:  $Q_1 \approx Q_2$  implies  $C[Q_1] \approx C[Q_2]$  where  $C$  is any acceptable evaluation context.

# Proof technique

We transform a game  $G_0$  into an observationally equivalent one using:

- **observational equivalences**  $L \approx R$  given as **axioms** and that come from security properties of primitives. These equivalences are used inside a context:

$$G_1 \approx C[L] \approx C[R] \approx G_2$$

- **syntactic transformations**: simplification, expansion of assignments, ...

We obtain a **sequence of games**  $G_0 \approx G_1 \approx \dots \approx G_m$ , which implies  $G_0 \approx G_m$ .

If some equivalence or trace property holds with overwhelming probability in  $G_m$ , then it also holds with overwhelming probability in  $G_0$ .

# MACs: security definition

A MAC scheme:

- (Randomized) key generation function *mkgen*.
- MAC function *mac*(*m*, *k*) takes as input a message *m* and a key *k*.
- Checking function *check*(*m*, *k*, *t*) such that
 
$$\text{check}(m, k, \text{mac}(m, k)) = \text{true}.$$

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the mac.

More formally, an adversary  $\mathcal{A}$  that has oracle access to *mac* and *check* has a negligible probability to forge a MAC (UF-CMA):

$$\max_{\mathcal{A}} \Pr[\text{check}(m, k, t) \mid k \xleftarrow{R} \text{mkgen}; (m, t) \leftarrow \mathcal{A}^{\text{mac}(\cdot, k), \text{check}(\cdot, k, \cdot)}]$$

is negligible, when the adversary  $\mathcal{A}$  has not called the *mac* oracle on message *m*.



# MACs: intuitive implementation

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- so when checking a MAC with  $check(m, k, t)$  and  $k \stackrel{R}{\leftarrow} mkgen$  is used only for generating and checking MACs, the check can succeed **only if  $m$  is in the list (array) of messages whose  $mac$  has been computed** by the protocol
- so we can replace a check with an array lookup:  
if the call to  $mac$  is  $mac(x, k)$ , we replace  $check(m, k, t)$  with

**find  $j \leq N$  suchthat defined( $x[j]$ )  $\wedge$   
 $(m = x[j]) \wedge check(m, k, t)$  then true else false**

# MACs: formal implementation

$$\text{check}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \mathbf{true}$$

$$!^{N''} \mathbf{new} \ r : \text{mkeyseed}; ($$

$$!^N (x : \text{bitstring}) \rightarrow \text{mac}(x, \text{mkgen}(r)),$$

$$!^{N'} (m : \text{bitstring}, t : \text{macstring}) \rightarrow \text{check}(m, \text{mkgen}(r), t))$$

$$\approx !^{N''} \mathbf{new} \ r : \text{mkeyseed}; ($$

$$!^N (x : \text{bitstring}) \rightarrow \text{mac}'(x, \text{mkgen}'(r)),$$

$$!^{N'} (m : \text{bitsting}, t : \text{macstring}) \rightarrow$$

$$\mathbf{find} \ j \leq N \ \mathbf{suchthat} \ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge$$

$$\text{check}'(m, \text{mkgen}'(r), t) \ \mathbf{then} \ \mathbf{true} \ \mathbf{else} \ \mathbf{false})$$

The prover understands such specifications of primitives.

They can be reused in the proof of many protocols.

# Proof strategy: advice

- CryptoVerif tries to apply **all equivalences** given as axioms, which represent security assumptions.

It transforms the left-hand side into the right-hand side of the equivalence.

- If such a **transformation succeeds**, the obtained game is then simplified, using in particular equations given as axioms.
- When these **transformations fail**, they may return syntactic transformations to apply in order to make them succeed, called **advised transformations**.

CryptoVerif then applies the advised transformations, and retries the initial transformation.

# Applications

- 16 “Dolev-Yao style” protocols that we study in the computational model. CryptoVerif proves all correct properties except in one case.
- 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)
- Avionic protocols (ARINC 823, ICAO9880 3rd edition)

# Conclusion and future work

- The automatic prover **ProVerif** works in the **symbolic** model. It is essentially mature; improve its documentation and interface.
- The automatic prover **CryptoVerif** works in the **computational** model. Much work still to do:
  - Improvements to the game transformations and the proof strategy.
  - Handle more cryptographic primitives (stateful encryption, ...)
  - Extend the input language (loops, mutable variables, ...)
  - Make more case studies.