Automatic Proof of Strong Secrecy for Security Protocols

By Bruno Blanchet

Originally presented at:
IEEE Symposium on Security and Privacy
Oakland, California, May 2004

Presented by David Fink

Outline

1. **Secrecy vs. Strong Secrecy**
   - Definitions, prior work, advantages
2. **Formal Spec: Applied Pi Calculus**
   - Generic/abstract cryptography with constructors/destructors
   - Model supports most cryptographic primitives, including probabilistic variants.
   - Uses unbounded parallel composition, unbounded name creation.
3. **Verification (semi)algorithm**
   - Algorithmic translation to Horn clauses representing deduction rules for processes and adversary for given protocol.
   - Uses resolution with free selection to derive forbidden clause.
   - *Main Contribution*: Unification predicate, *testunif*(*p*,*p*') to test for distinguishability of terms *p*,*p*'.
4. **Decidability and Efficiency**
   - Very efficient implementation in Prolog: **ProVerif** tool.
   - However, in general proof technique may not terminate.
   - Termination proven for tagged protocols.

Standard Secrecy

Traditional secrecy is defined over traces if the Dolev-Yao model:

Assuming perfect cryptography, for any term s, s remains secret iff no possible (possibly infinite) trace results in the adversary learning s.

$$\{A,N_A\}_{PK_I} \quad \{A,N_A\}_{PK_B}$$
$$\{N_A,N_B\}_{PK_A} \quad \{N_B\}_{PK_I}$$
$$\overline{\qquad\qquad\qquad\qquad}$$
$$N_B$$

Secrecy is nondeducability

Defined over an arbitrary secret and over all possible traces.

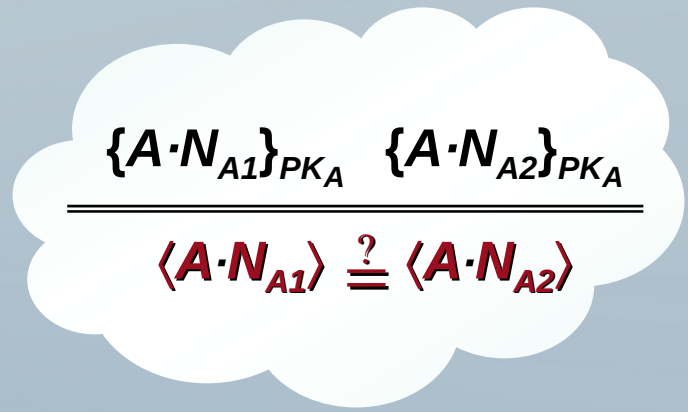
Weaknesses

- *Deducibility of single secrets cannot model partial information leaks or distinguishability of ciphertexts.*
- *Automated proof of secrecy is already undecidable for arbitrary protocols[1], even with bounded sessions and message length[2], so more powerful model no "worse" computationally.*

1. Even and Goldreich 1983, Heintze & Tygar 1996
2. Durgin, Lincoln, Mitchell & Scedrov, "Undecidability of Bounded Security Protocols", 1999.

Strong Secrecy

Strong secrecy requires that ciphertexts containing *possibly* different secrets are indistinguishable from each other by the adversary. Thus, processes whose messages differ only in their ciphertext contents are also indistinguishable.



Strong secrecy is observational equivalence with respect to different secret values; i.e. the adversary cannot see if or when a secret encrypted value changes.

Comparative Advantages

- *Strong secrecy subsumes standard secrecy – it is a strictly more difficult property to satisfy.*
- *More refined: can model implicit flow detection (detecting differing process behaviors depending on differing ciphertexts).*
- *It is a type of process equivalence, so we gain compositionality, aiding proof automation.*
- *Closer to computational model of secrecy.*

Background Work

This paper builds on the following by Blanchet and others:

1. Abadi & Blanchet, “Analyzing Security Protocols with Secrecy Types and Logic Programs” (*POPL 2002*).
   - *Introduced typed constructors and destructors in the Pi calculus to model abstract cryptography. Proved equivalence between this calculus and untyped logic programs (Prolog). Presented verification algorithm as logic program.*
2. Blanchet & Podelski “Verification of Cryptographic Protocols: Tagging Enforces Termination” (*FOSSACS 2003*)
   - *Proved that the verification algorithm described above terminates for tagged protocols, proving preservation of secrecy is decidable.*

☞ *More on tagged protocols later…*

Extended Pi Calculus

Elsewhere called Applied Pi Calculus

Grammar:

M,N ::=	**Terms (Messages)**	
	x, y, z	variables
	a, b, c, k, s	names
	$f(M_1,\ldots ,M_n)$	constructor application
P,Q ::=	**Processes**	
	0	nil
	$P \mid Q$	parallel composition
	$!P$	replication
	$(\nu a)P$	restriction (free variable instantiation)
	$M\langle\bar{N}\rangle.P$	output
	$M(x).P$	input
	let x = g$(M_1,\ldots ,M_n)$ in P *else* Q	destructor application
	if $M = N$ *then* P *else* Q	conditional

Constructors & Destructors

- **Constructor *f* used to build terms:** $f(M_1, \cdots, M_n)$
  - Used for encryption, digital signatures, hashing, etc.
- **Destructor *g* used to break down and analyze terms:**

$$\text{let } x = g(M_1, \cdots, M_n) \text{ in } P \text{ else } Q$$

  - Destructor process can be seen as a reduction:

$$g(M_1, \cdots, M_n) \Rightarrow M$$

  - Decryption, signature, verification, etc.
  - Destructor arguments must be grounded

- ***Constructors and destructors can be public or private.***
- ***Note that constructors and destructors are abstract. Any specific cryptosystem can be instantiated here.***

From (Con/De)structors to (En/De)cryption

Public key encryption as constructor:

- Public key generation constructor : $pk(N)$ *(N is private)*
- Encryption constructor: $pencrypt(M,N)$
- Probabilistic PK encryption : $pencrypt_{prob}(M,N,R)$
- Example : $(\nu r)\bar{c}\langle pencrypt_{prob}(M,pk_A,r)\rangle$

Public key decryption as destructor:

- Decryption destructor : $pdecrypt(M',N')$
- Example : $pdecrypt(pencrypt_{prob}(M, pk(N), R), N) \Rightarrow M$

Constructors and destructors are similarly defined for symmetric key encryption, digital signatures, hash functions, MAC's, with or without probability.

Some Definitions

- *fv*(*P*) and *fn*(*P*) are the free variables and names in P, respectively.
- *P* is a **closed** process iff

$$|fv(P)| + |fn(P)| = 0$$

- A *term* is closed (or ground) if it has no free variables, and a *substitution* is closed if its image consists only of closed terms.
- Process equivalence ($\equiv$) and process reduction relations ($\Rightarrow$) are only defined over closed processes.

Example: Corrected, Simplified Denning-Sacco

This protocol is designed to accomplish secure key exchange and maintain the secrecy of x, a value chosen by B, so it is free variable:

$$A \rightarrow B : \{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B} \quad \textit{(k is a fresh secret)}$$
$$B \rightarrow A : \{x\}_k$$

In the process algebra:

$$P_A(pk_A, sk_A) \triangleq c(x\_pk_B).(\nu k)(\nu r)$$
$$\bar{c}\langle pencrypt_{prob}(sign((pk_A, x\_pk_B, k), sk_A), x\_pk_B, r)\rangle.$$
$$c(x').let\ s = sdecrypt(x', k)\ \text{in}\ 0$$

$$P_B(pk_B, sk_B, pk_A) \triangleq c(y).let\ \ y' = pdecrypt_{prob}(y, sk_B)\ \ in$$
$$let\ (= pk_A, = pk_B, k)\ =\ checksign(y', pk_A)\ in\ \bar{c}\langle sencrypt(x, k)\rangle$$

$$P_0 \triangleq (\nu sk_A)(\nu sk_B)\ let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in$$
$$\bar{c}\langle pk_A\rangle\bar{c}\langle pk_B\rangle.(!P_A(pk_A, sj_A)|!P_B(pk_B, sk_B, pk_A))$$

We want to prove the strong secrecy of *x*.

Strong Secrecy, Formally

Adversary as Context: We consider the adversary as an evaluation context, built from [], C|P, P|C, and (νa)C. It can run unbounded parallel sessions with unbounded data.

Observational Equivalence: Two processes P and Q are observationally equivalent, denoted $P \approx Q$, if no adversarial evaluation context exists that would allow the adversary to decide $P \neq Q$.

Strong Secrecy: A process P_0 preserves the strong secrecy of its free variables iff for all closed substitutions σ and σ' of domain $fv(P_0)$, $\sigma P_0 \approx \sigma' P_0$.

Intuition: No pair of different ground substitutions of free variables results in observably different behavior by P_0.

Proof Technique - Abstraction

In order to reuse prior automated proof technique, the adversary is defined as a process (or set of processes) running in parallel with protocol processes.

- **Intuition**: Each reduction step of process P_0 is *independent* of the values of its secrets.
- **Condition 1**: The success or failure of communications is independent of the secrets.
- **Condition 2**: The success or failure of destructor applications is independent of the secrets.

Proposition: Proof Obligations

Let process P_0' be derived from P_0 by substituting distinct free names from a set ***Secr*** for free variables of P_0, and let Q be any adversary s.t. *fn*(Q) ý ***Secr*** = þ

Condition 1: P_0' | Q does not communicate over a channel in ***Secr***.

- *If it did, adversary could see that communication succeeded. This assures that P_0' does not leak distinguishing secrets.*

Condition 2: If P_0' | Q executes a destructor application *let x = g(M_1,…,M_n) in Q' else R'* that succeeds for some value in ***Secr***, it succeeds for all values in ***Secr***.

- *If not, adversary could could distinguish between Q' executing for some values of the secrets and R' executing for others.*

If Conditions 1 and 2 hold, P_0 preserves the strong secrecy of its fv(P_0).

Horn Clause Representation

Automation of the proof is similar to Abadi & Blanchet's POPL 2002 paper "Analyzing Security Protocols with Secrecy Types".

The algorithm is based on an a reduction to Horn clauses, which encode the deductive rules.

Starting from closed process P_0. Each restriction $(\nu a)P_0$ has a different name a. In order to distinguish between different copies of P_0, each replication of P_0 has a unique *session identifier* associated with it.

Horn clause terms, called *patterns*, are generated from the following grammar:

p ::=	*x,y,z*	variable
	e	element of **EVar**
	x	element of **Secr**
	a[p_1,...,p_n]	name
	f(p_1,...,p_n)	constructor application

Horn Clause Represenatation

Name creation $(\nu a)P_0$ under replication is replaced **name function** $a\,[p_1,...,p_n]$. If the name is free (unbound), the function arity is 0. Bound names are represented by name function of arity equal to the number of inputs, destructor applications, and replications above it. The use of name functions eliminates unbounded names under replication.

The Horn clauses use the following predicates (facts)

att(p)	:	attacker may have p
mess(p,p')	:	message p' may be appear on channel p
com(p)	:	attacker may communicate on channel p
testunif(p,p')	:	unification test ← **key addition**
bad	:	derivable iff strong secrecy does not hold

Testunif

*Testunif is specific to strong secrecy. It detects when a destructor application (usually a decryption) succeeds for some values of the secrets in **Secr** but not for others.*

Let p, p' be closed patterns, ***Secr*** be a set of secret, unbound names, and **EVar** be a set of constants disjoint from ***Secr***.

Testunif(p, p′) is true iff:

1. *p* and *p'* can be unified – there exists closed substitution σ from domain ***Secr*** ∪ ***EVar***, such that σ***Secr*** does not contain bound names and $\sigma p = \sigma p'$.
2. *p* and *p′* cannot otherwise be unified – no closed substitution σ' from domain ***EVar*** exists such that $\sigma' p = \sigma' p'$.

Therefore Testunif returns true iff the adversary can't distinguish between encrypted secrets without knowing a secret already.

Testunif can be used to check that Condition 2 holds in the proof.

Rules

Attacker Rules: Encoding of Dolev-Yao derivation as nine rules, but adds new rules for deriving bad.

- **EVar**($(N_1,\ldots,N_n)$) is a substitution of variables for values in ***EVar***.
- $att(x_1) \wedge \ldots \wedge att(x_n) \wedge testunif((x_1,\ldots x_n), \textbf{EVar}((N_1,\ldots,N_n))) \Rightarrow$ bad
- If $x \in$ ***Secr***, then $com(x) \Rightarrow$ bad.

Protocol rules: Standard encoding from prior paper, ensuring fresh session identifier is added for each replicated process.

Solving Algorithm (sketch)

Resolution with Free Selection:

R: Rule H: Hypothesis F: Fact C: Conclusion

$$\frac{H \Rightarrow C \quad F \wedge H' \Rightarrow C'}{\sigma H \wedge \sigma H' \Rightarrow \sigma C'}$$

Selection function chooses which rules to apply at any point. Rules define protocol and adversary actions. See paper for details.

- Selection function **sel** used to pick which rule to apply. Picks hypothesis not of the form *att(x)* or *testunif(p. p')* if possible, or the conclusion otherwise. Tries to avoid resolving on fact *att(x*).
- A set of simplification steps used to decide if testunif(*x,x'*) holds : testunif does not depend on clauses.
- Repeat rule selection function / rule application / clause set simplification until a **fixpoint** is achieved.
- If fixpoint includes bad, strong secrecy fails. If not, strong secrecy is proven.

Correctness / Incompleteness

Theorem 1: The clause *bad* is derivable from the input clauses iff the algorithm generates it.

Theorem 2: The algorithm does not generate *bad* iff the protocol preserves strong secrecy of its free vars.

- ***Limitation***: The algorithm may not terminate for all inputs.
  - *Fixpoint may not be found.*
  - *Not surprising – this is automated theorem proving.*

Decidable Subclass: Decidablility proven for ***tagged protocols*** – protocols which syntactically disambiguate all encrypted subterms in a protocol.

- *Extension of proof from previous paper.*

Tagged Protocols

A **tagged protocol** in the Applied Pi Calculus is a process P_0 with the following properties:

- All communication occurs over a single public channel.
- Each constructor in a tagged protocol adds a **unique tag** (constant name) to each distinct constructor:

$$f(t, M_1,\ldots,M_N)$$

- Every (honest) destructor (*let x = g(…) in P else Q*) must first check for **tag equality** before proceeding. If the tag is not equal, the process must end (fail-stop):

$$\textbf{let } y = 1th_n(x) \textbf{ in if } y = t \textbf{ then } P' \textbf{ else } 0$$

  - *Extension in paper admits weaker model that handles non-empty case for error-handling: protocol should still not make progress in "else" process.*

- Long term secrets are atomic constants – secrets are not only not lost, but not created by the protocol.

Tagged Protocols

An interesting restricted class of protocols:

- Guarantees that ***intent*** of each encrypted term is unambiguous.
- Eliminates need to consider messages with unbounded length.
- Prevents all type-flaw attacks
- Used to prove:
  - **Completeness of model checking** (*Gavin Lowe, 1998*)
  - **Decidability of secrecy** (*Blanchet & Podelski 2003, Ramaujam & Suresh 2003, using a very different formalism*).

Most security protocols can be modified to become tagged protocols.

Research question: Which protocols **cannot** be tagged? Interesting subclass? *Diffie-Hellman*, for one.

Results

1. Corrected Denning-Sacco:

$$A \rightarrow B : \{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B}$$
$$B \rightarrow A : \{x\}_k$$
$$A \rightarrow B : \{x'\}_k$$

   - *Preserves strong secrecy of x and x' only if encryption is probabilistic **or** if we add tags c_0, c_0' to messages 2 and 3.*

2. JFKi: preserves strong secrecy of the initiator.
   - *Proof uses extensions to the proof system given in latest technical report version of this paper.*

Conclusion

- Formal definition of strong secrecy allows for the extension of prior results in proof automation via logic programming to be applied to strong secrecy.
- Automated translation from Applied Pi Calculus to Horn clauses.
- Resolution algorithm proves or disproves strong secrecy. May not terminate on all protocols. Terminates for tagged protocols.
- Does not consider weakening the term algebra (malleability, weak keys, RSA, etc.)
- Free tool available: **Proverif**
  - See http://www.di.ens.fr/~blanchet/crypto-eng.html