The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm

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

ProVerif is a widely used security protocol verifier. Internally, ProVerif uses an abstract representation of the protocol by Horn clauses and a resolution algorithm on these clauses, in order to prove security properties of the protocol or to find attacks. In this paper, we present an overview of ProVerif and discuss some specificities of its resolution algorithm, related to the particular application domain and the particular clauses that ProVerif generates. This paper is a short summary that gives pointers to publications on ProVerif in which the reader will find more details.

1 Introduction

The verification of security protocols is a very active research area since the 1990's. Security protocols are ubiquitous: Internet (in particular, the TLS protocol used for `https://` connections), WiFi, mobile phones, credit cards, ... Their design is notoriously error-prone, and errors are not detected by testing: they appear only when an adversary tries to attack the protocol. Therefore, it is important to formally verify them.

In order to formalize security protocols, one needs a mathematical model for them. One typically considers an active adversary, which can listen to messages sent on the network, compute its own messages, and send them to the network as if they came from honest participants. To facilitate the automatic verification of protocols, most protocol verifiers consider the symbolic model of cryptography, also called "Dolev-Yao model" [18, 15]. In this model, cryptographic primitives, such as encryption, are considered as ideal black-boxes, represented by function symbols; messages are modeled by terms on these primitives; and the adversary is restricted to apply defined primitives. This is also called the perfect cryptography assumption: the only way for the adversary to decrypt a message is to use the decryption function with the correct key. In such a model, one of the main tasks of protocol verification consists in computing the knowledge of the adversary, that is, the set of terms that the adversary can obtain. This is still non-trivial as this set is typically infinite, but it is much simpler than reasoning about bitstrings and probabilities as in cryptographic proofs. The two most widely used symbolic protocol verifiers are probably ProVerif [11] and Tamarin [17]. For more details on the field of protocol verification, we refer the reader to the surveys [10, 6]. In this paper, we focus on the protocol verifier ProVerif, which can be downloaded from `https://proverif.inria.fr`. We present an overview of ProVerif in the next section and focus on its Horn clause resolution algorithm in Section 3.

2 ProVerif

As illustrated in Figure 1, ProVerif takes as input a description of the protocol to verify in an extension of the pi calculus with function symbols to represent cryptography, a dialect of the applied pi calculus [3, 2].

G.W. Hamilton, T. Kahsai, M. Proietti (Eds.):
Joint International Workshops on
Horn Clauses for Verification and Synthesis (HCVS 2022)
Verification and Program Transformation (VPT 2022)
EPTCS 373, 2022, pp. 14–22, doi:10.4204/EPTCS.373.2

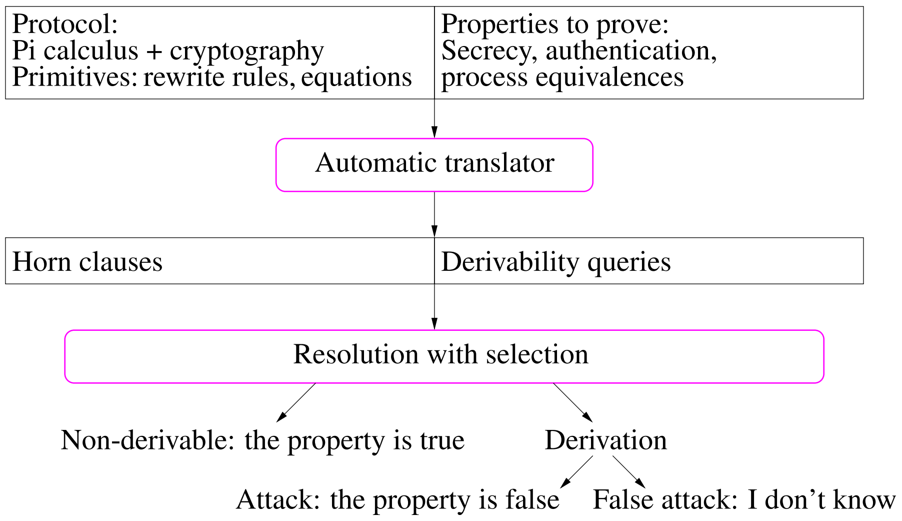
Protocol:
Pi calculus + cryptography
Primitives: rewrite rules, equations
Properties to prove:
Secrecy, authentication,
process equivalences
Automatic translator
Horn clauses
Derivability queries
Resolution with selection
Non-derivable: the property is true
Derivation
Attack: the property is false
False attack: I don't know

Figure 1: The structure of ProVerif

This language can be seen as a small, domain-specific, programming language for security protocols. Additionally, one needs to define the cryptographic primitives, using rewrite rules or equations. For instance, shared-key encryption can be represented by a free function symbol senc that takes as argument a message and a key and returns the corresponding ciphertext. The corresponding decryption function sdec can be defined by a rewrite rule

$$\mathsf{sdec}(\mathsf{senc}(m,k),k) \to m$$

which means that, when we decrypt a ciphertext $\mathsf{senc}(m,k)$ with the correct key k, we obtain the cleartext m. Free function symbols like senc are named *constructors* while function symbols defined by rewrite rules are named *destructors*.

ProVerif also takes as input the security properties to prove, named queries, which can be:

- *secrecy* properties [1]: the adversary cannot compute certain values;
- *authentication* properties [9]: if some participant Alice thinks she talks to Bob, then she really talks to Bob. Authentication is formalized by correspondence properties [20], of the form: if some event has been executed, then some other event has been executed. Events can represent that Alice concluded the protocol apparently with Bob, or that Bob started the protocol, apparently with Alice.
- *equivalence* properties [12]: the adversary cannot distinguish two protocols. Equivalence properties are a powerful notion to specify security properties (such as anonymity and privacy), but they are also difficult to verify. ProVerif can verify only a strong notion of equivalence, named *diff-equivalence*, between protocols that have the same structure but differ only by the messages they exchange.

ProVerif proves such security properties thanks to an internal representation of the protocol by Horn clauses, obtained by an automatic translation from the protocol and the cryptographic primitives. Simplifying as much as possible, the clauses use as main predicate att: $\mathsf{att}(M)$ means that the adversary may know the term M. Some clauses represent computations by the adversary:

- For each constructor f of arity n, the clause

  $$\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$$

  is generated, representing that the adversary can compute $f(x_1, \ldots, x_n)$ by applying f when it has $x_1, \ldots, x_n$. For instance, for shared-key encryption senc, the following clause is generated:

  $$\mathsf{att}(m) \wedge \mathsf{att}(k) \Rightarrow \mathsf{att}(\mathsf{senc}(m,k)) \qquad (\mathsf{senc})$$

- For each destructor g, defined by a rewrite rule $g(M_1, \ldots, M_n) \to M$, the clause

  $$\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \Rightarrow \mathsf{att}(M)$$

  is generated, representing that the adversary can compute M when it has $M_1, \ldots, M_n$, by applying g. For instance, for shared-key decryption sdec, defined by $\mathsf{sdec}(\mathsf{senc}(m,k),k) \to m$, the following clause is generated:

  $$\mathsf{att}(\mathsf{senc}(m,k)) \wedge \mathsf{att}(k) \Rightarrow \mathsf{att}(m) \qquad (\mathsf{sdec})$$

  If the adversary has the ciphertext $\mathsf{senc}(m,k)$ and key k, it can obtain the cleartext m by decryption.

Other clauses represent the protocol itself: if a principal A has received the messages $M_1, \ldots, M_n$ and sends the message M, the following clause is generated:

$$\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \Rightarrow \mathsf{att}(M)\,.$$

Indeed, if the adversary has M_1, ..., M_n, it can send them to A, which is going to reply with M. The adversary intercepts this message and thus obtains M. For instance, consider the following protocol, inspired by the Denning-Sacco key distribution protocol [14]:

Message 1. $A \to B : \mathsf{penc}(\mathsf{sign}(k, sk_A), pk_B)$
Message 2. $B \to A : \mathsf{senc}(\mathsf{s}, k)$

The symbol penc represents public-key encryption, sign represents a signature. In this protocol, A generates a fresh key k and aims to share it with B. A signs k with her secret key sk_A and encrypts it under B's public key pk_B. Only B can decrypt this message; then B verifies A's signature and obtains the key k. In the second message, B uses this key k to encrypt a secret s under k, using shared-key encryption.

In this protocol, upon receipt of a message of the form $\mathsf{penc}(\mathsf{sign}(y, sk_A), pk_B)$, B replies with the message $\mathsf{senc}(\mathsf{s}, y)$, so the generated clauses include

$$\mathsf{att}(\mathsf{penc}(\mathsf{sign}(y, sk_A), pk_B)) \Rightarrow \mathsf{att}(\mathsf{senc}(\mathsf{s}, y))\,.$$

This clause represents that the adversary sends $\mathsf{penc}(\mathsf{sign}(y, sk_A), pk_B)$ to B, and intercepts his reply $\mathsf{senc}(\mathsf{s}, y)$.

ProVerif also translates the security properties to prove into derivability queries on the generated Horn clauses. For instance, in order to prove that s is secret, ProVerif proves that $\mathsf{att}(\mathsf{s})$ is *not* derivable from the generated Horn clauses. Intuitively, the adversary is then unable to compute s. More generally, a security property is proved when no instance of a certain fact F is derivable from the clauses. ProVerif uses a resolution algorithm (detailed in the next section) in order to determine whether some instance of F is derivable from the clauses or not.

When an instance of F is derivable from the clauses, the derivation is the witness of an attack. However, the Horn clause representation introduces an abstraction: mainly, the Horn clauses can be applied any number of times in a derivation [8], but that does not always correspond to what happens in the protocol, since some protocol steps may be applicable only once, for instance. Because of this abstraction, a derivation, that is, an attack at the Horn clause level, does not always correspond to an attack at the protocol level. ProVerif therefore uses an attack reconstruction algorithm [4] in order to reconstruct an attack at the protocol level from the derivation. When this algorithm succeeds in finding an attack, the security property is definitely false. When this attack reconstruction algorithm fails, the derivation is a so-called "*false attack*"; in this case, we do not know whether the security property holds or not (see Example 1 below for an example).

As Horn clauses can be used for resolution an unbounded number of times by default, this abstraction allows ProVerif to prove properties for an unbounded number of sessions of the protocol, an undecidable problem [16]. However, because of this abstraction, ProVerif is not complete: it does not always succeed in deciding whether a security property holds or not. Moreover, in general, the resolution algorithm may not terminate. In practice, ProVerif is still precise and efficient for many examples of protocols.

3 The resolution algorithm

The resolution algorithm of ProVerif is based on resolution with free selection [5]. A selection function selects one literal in each clause, and the algorithm performs resolution upon selected literals, that is, from two clauses $R = H \Rightarrow C$ and $R' = F' \wedge H' \Rightarrow C'$ where the conclusion C is selected in R and the hypothesis F' is selected in R', the algorithm generates the clause $H\sigma \wedge H'\sigma \Rightarrow C'\sigma$, where σ is the most general unifier of C and F'. Intuitively, this clause is obtained by using R to infer $C\sigma = F'\sigma$ from $H\sigma$, and then using R' to infer $C'\sigma$ from $F'\sigma$ and $H'\sigma$. These resolution steps are performed between all clauses until a fixpoint is reached, that is, no new clause can be added. Finally, among the clauses in this fixpoint, only the clauses in which the conclusion is selected are kept.

The following theorem states the soundness and completeness of resolution with free selection. It is a particular case of [9, Lemma 2].

Theorem 1 *The set of clauses obtained by resolution with free selection derives the same facts as the initial clauses.*

The key idea is to choose the selection function to avoid resolving upon facts of the form $\mathsf{att}(x)$ for a variable x, because such facts unify with any fact $\mathsf{att}(M)$, yielding many resolution steps and non-termination. Hence a basic selection function selects some hypothesis not of the form $\mathsf{att}(x)$ if possible, and the conclusion when all hypotheses are of the form $\mathsf{att}(x)$.

ProVerif uses standard optimizations of resolution provers (elimination of subsumed clauses, of tautologies, ...) [9]. We do not detail those further. However, it also uses less standard, domain-specific optimizations and extensions. We sketch a few of them below, with references of papers in which more details can be found.

Data constructors [9] Data constructors are constructors f that come with associated projections π_i^f defined by $\pi_i^f(f(x_1,\dots,x_n)) = x_i$ for $1 \leq i \leq n$. Such constructors model data structures that appear in protocols; projections allow to obtain the elements of the structure. For such constructors, we have clauses

$$\mathsf{att}(x_1) \wedge \dots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1,\dots,x_n)) \qquad \text{(for } f\text{)}$$

$$\mathsf{att}(f(x_1,\dots,x_n)) \Rightarrow \mathsf{att}(x_i) \qquad \text{(for } \pi_i^f\text{)}$$

Therefore, $\mathsf{att}(f(x_1,\dots,x_n))$ is equivalent to $\mathsf{att}(x_1) \wedge \dots \wedge \mathsf{att}(x_n)$. So we can simplify clauses by decomposing data constructors, except in the two clauses above: we replace $\mathsf{att}(f(x_1,\dots,x_n))$ with $\mathsf{att}(x_1) \wedge \dots \wedge \mathsf{att}(x_n)$ in hypotheses of clauses and we replace clauses $H \Rightarrow \mathsf{att}(f(x_1,\dots,x_n))$ with n clauses $H \Rightarrow \mathsf{att}(x_i)$. (This transformation corresponds to resolving each clause R with the clauses for f and for π_i^f. However, the difference with an ordinary resolution step is that the initial clause R can be removed. Only the transformed clause is kept.)

Blocking predicates [9] Blocking predicates are predicates that occur in the hypothesis of clauses and on which we do not resolve: the selection function never selects them. Hence, they remain in the hypothesis of clauses until the end of the resolution algorithm. ProVerif proofs are valid for any definition of these predicates. For instance, they can be used to model conditions that could not be explicitly defined in ProVerif, such as conditions on real numbers. They are also very useful in order to prove correspondence properties, as explained in [9, Section 4]: we represent events that we wish to prove using a blocking predicate; the presence of these blocking events in the hypotheses of clauses at the end of resolution shows that they must have been executed in order to reach the conclusion of these clauses.

Disequations [12] ProVerif also supports disequality constraints modulo the equations that define the cryptographic primitives, of the form $\forall x_1,\dots,x_n.M \neq N$. These disequality constraints may occur in the hypothesis of clauses. They are handled in the resolution algorithm using simplifications explained in [12].

Natural numbers [13] ProVerif supports natural numbers, which can be used to represent counters: it supports constant natural numbers, addition and subtraction of a variable and a constant, comparisons, and tests whether a term is a natural number or not. Natural numbers are implemented in clauses with constraints $\mathsf{is\_nat}(M)$ (M is a natural number), $\neg\mathsf{is\_nat}(M)$ (M is not a natural number), and $M \geq N + n$ where n is a constant natural number. The latter constraints are simplified using the Bellman-Ford algorithm [7].

Temporal correspondence queries [13] ProVerif supports correspondence queries that use events of the form $\mathbf{event}(M)@i$, where i is a temporal variable: $\mathbf{event}(M)@i$ means that event M happened at step i. One can then compare temporal variables with each other: for instance $\mathbf{event}(A(x))@i$ && $\mathbf{event}(B(x))@j \Longrightarrow i < j$ means that if event $A(x)$ happens at step i and $B(x)$ happens at step j then $i < j$, so event $A(x)$ happens before event $B(x)$. These queries are encoded using special natural number constraints $i < j$ and $i \leq j$.

Precise actions [13] Precise actions allow us to avoid false attacks that come from the repeated usage of Horn clauses in a derivation, as illustrated by the following example.

Example 1 *Consider a protocol in which the participant A sends the messages* $\mathsf{senc}(k_1,k)$, $\mathsf{senc}(k_2,k)$, *and* $\mathsf{senc}(s,(k_1,k_2))$, *and the participant B receives one message x and sends its decryption under k, where k,* k_1, k_2 *are keys initially secret and s is a secret. B acts as a decryption oracle under k, but only once. Therefore, the adversary can obtain either* k_1 *or* k_2 *by decrypting either* $\mathsf{senc}(k_1,k)$ *or* $\mathsf{senc}(k_2,k)$ *using B, but it cannot obtain both* k_1 *and* k_2*, hence it cannot decrypt* $\mathsf{senc}(s,(k_1,k_2))$ *and cannot obtain s.*

The clauses for this protocol include:

$$\begin{array}{llll} \mathsf{att}(\mathsf{senc}(k_1,k)) & \mathsf{att}(\mathsf{senc}(k_2,k)) & \mathsf{att}(\mathsf{senc}(s,(k_1,k_2))) & \text{for } A \\ \mathsf{att}(\mathsf{senc}(y,k)) \Rightarrow \mathsf{att}(y) & & & \text{for } B \\ \mathsf{att}(x) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}((x,y)) & & & \text{for the pair} \end{array}$$

as well as the clauses (senc) *and* (sdec) *for encryption and decryption. The clause for B can be applied any number of times in a derivation. Using this clause, we derive* $\mathsf{att}(k_1)$ *from* $\mathsf{att}(\mathsf{senc}(k_1,k))$ *and* $\mathsf{att}(k_2)$ *from* $\mathsf{att}(\mathsf{senc}(k_2,k))$*. Then we derive* $\mathsf{att}((k_1,k_2))$ *by the clause for the pair, and* $\mathsf{att}(s)$ *using* $\mathsf{att}(\mathsf{senc}(s,(k_1,k_2)))$ *and the clause for* sdec*. Therefore, at the Horn clause level, the adversary can obtain s: this is a false attack, due to the repeated usage of the clause for B, which does not match the specification of the protocol. With these clauses, ProVerif is unable to prove secrecy of s. This false attack can be avoided by tagging* `precise` *the input that receives message x in B.*

Precise actions are implemented as a particular axiom, as we explain below.

Restrictions, axioms, lemmas [13] Syntactically, restrictions, axioms, and lemmas are particular correspondence queries. However, they play a different role:

- Restrictions restrict the set of traces on which queries are proved: queries are proved only on traces that satisfy the restrictions.
- Axioms are properties that hold on all considered traces, but are not proved by ProVerif: they are assumed. For instance, they are useful to use properties that are proved manually and that ProVerif cannot prove.
- Lemmas are proved by ProVerif, and then assumed in the proof of subsequent lemmas and queries.

ProVerif uses restrictions, axioms, and already proved lemmas in order to strengthen clauses generated during resolution, as follows. Suppose that a proved lemma (or axiom or restriction) is $\bigwedge_i F_i \Longrightarrow \phi$ and that resolution generates a clause $H \Rightarrow C$ such that for all i, $F_i\sigma \in H$ or $F_i\sigma = C$. Then we know that $\bigwedge_i F_i\sigma$ holds, hence by the lemma, $\phi\sigma$ holds as well. Therefore, we add $\phi\sigma$ in the hypothesis of the clause, yielding $H \wedge \phi\sigma \Rightarrow C$.

Example 2 (Example 1 continued) *Precise actions are implemented using axioms. After each precise input that receives message x, we generate a fresh name occ and execute the event* $\mathsf{Precise}(occ,x)$*, and we add the axiom*

$$\textbf{event}(\mathsf{Precise}(occ,x_1)) \ \&\& \ \textbf{event}(\mathsf{Precise}(occ,x_2)) \Longrightarrow x_1 = x_2 \qquad (\mathsf{Precise})$$

This axiom means that, if events $\mathsf{Precise}(occ,x_1)$ *and* $\mathsf{Precise}(occ,x_2)$ *are executed with the same value of occ (hence after the same execution of the precise input), then* $x_1 = x_2$*, that is, the received message x is the same.*

In Example 1, there is a single execution of the precise input, the input of B, so a single name occ. Therefore, the axiom shows that a single message x can be processed by B.

During resolution with the precise input, the following clause is generated

$$\textbf{event}(\mathsf{Precise}(occ,\mathsf{senc}(k_1,k))) \wedge \textbf{event}(\mathsf{Precise}(occ,\mathsf{senc}(k_2,k))) \Rightarrow \mathsf{att}(s)$$

which means that if B has been used to decrypt $\mathsf{senc}(k_1,k)$ *(event* $\mathsf{Precise}(occ,\mathsf{senc}(k_1,k))$ *has been executed) and* $\mathsf{senc}(k_2,k)$ *(event* $\mathsf{Precise}(occ,\mathsf{senc}(k_2,k))$ *has been executed), then the adversary obtains s. Using the axiom* (Precise)*, this clause is transformed into*

$$\mathbf{event}(\mathsf{Precise}(occ,\mathsf{senc}(k_1,k))) \wedge \mathbf{event}(\mathsf{Precise}(occ,\mathsf{senc}(k_2,k))) \wedge \mathsf{senc}(k_1,k) = \mathsf{senc}(k_2,k) \Rightarrow \mathsf{att}(s)$$

and hence removed because $\mathsf{senc}(k_1,k) \neq \mathsf{senc}(k_2,k)$*. That avoids the false attack that we had previously and ProVerif can now prove the secrecy of s.*

Proofs by induction [13] In order to perform proofs by induction, we use as lemma the property that we want to prove itself, but on a strict prefix of the trace. Since intuitively the hypothesis of a clause happens strictly before the conclusion, when we apply an inductive lemma to a clause, its hypothesis must be proved using only the hypothesis of the clause, not the conclusion of the clause.

All extensions of [13] are recent extensions of ProVerif designed and implemented mainly by Vincent Cheval. As explained in [13, Section 4], he also optimized ProVerif considerably, by revising many of its algorithms. In particular, using Schulz's idea of feature vertex indexing [19] allowed him to speed up subsumption tests considerably and indexing clauses in a prefix tree allowed him to speed up the resolution steps themselves.

4 Conclusion

Horn clauses are a powerful mechanism to reason about terms, hence about security protocols in the symbolic model. As shown in Section 3, tuning the resolution algorithm with specific clause simplifications and transformations allows one to implement many optimizations and extensions that are essential for reasoning about protocols.

Acknowledgment This work was partly supported by ANR TECAP (decision number ANR-17-CE39-0004-03).

References

[1] Martín Abadi & Bruno Blanchet (2005): *Analyzing Security Protocols with Secrecy Types and Logic Programs*. Journal of the ACM 52(1), pp. 102–146, doi:10.1145/1044731.1044735. Available at `https://bblanche.gitlabpages.inria.fr/publications/AbadiBlanchetJACM7037.html`.

[2] Martín Abadi, Bruno Blanchet & Cédric Fournet (2017): *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. Journal of the ACM 65(1), pp. 1:1–1:41, doi:10.1145/3127586. Available at `https://bblanche.gitlabpages.inria.fr/publications/AbadiBlanchetFournetJACM17.html`.

[3] Martín Abadi & Cédric Fournet (2001): *Mobile Values, New Names, and Secure Communication*. In: *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, ACM Press, New York, NY, pp. 104–115, doi:10.1145/360204.360213.

[4] Xavier Allamigeon & Bruno Blanchet (2005): *Reconstruction of Attacks against Cryptographic Protocols*. In: *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 140–154, doi:10.1109/CSFW.2005.25. Available at `https://bblanche.gitlabpages.inria.fr/publications/AllamigeonBlanchetCSFW05.html`.

[5] L. Bachmair & H. Ganzinger (2001): *Resolution Theorem Proving*. In A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, chapter 2, 1, North Holland, pp. 19–100, doi:10.1016/B978-044450813-3/50004-7.

[6] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao & Bryan Parno (2021): *SoK: Computer-Aided Cryptography*. In: *IEEE Symposium on Security and Privacy (S&P'21)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 777–795, doi:10.1109/SP40001.2021.00008. Available at `https://bblanche.gitlabpages.inria.fr/publications/BarbosaetalOakland21.html`.

[7] Richard Bellman (1958): *On a routing problem*. *Quarterly of Applied Mathematics* 16, pp. 87–90, doi:10.1090/qam/102435.

[8] Bruno Blanchet (2005): *Security Protocols: From Linear to Classical Logic by Abstract Interpretation*. *Information Processing Letters* 95(5), pp. 473–479, doi:10.1016/j.ipl.2005.05.011. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetIPL05.html`.

[9] Bruno Blanchet (2009): *Automatic Verification of Correspondences for Security Protocols*. *Journal of Computer Security* 17(4), pp. 363–434, doi:10.3233/JCS-2009-0339. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetJCS08.html`.

[10] Bruno Blanchet (2012): *Security Protocol Verification: Symbolic and Computational Models*. In Pierpaolo Degano & Joshua Guttman, editors: *First Conference on Principles of Security and Trust (POST'12)*, *Lecture Notes in Computer Science* 7215, Springer, Berlin, Heidelberg, pp. 3–29, doi:10.1007/978-3-642-28641-4_2. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetETAPS12.html`.

[11] Bruno Blanchet (2016): *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*. *Foundations and Trends in Privacy and Security* 1(1–2), pp. 1–135, doi:10.1561/3300000004. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetFnTPS16.html`.

[12] Bruno Blanchet, Martín Abadi & Cédric Fournet (2008): *Automated Verification of Selected Equivalences for Security Protocols*. *Journal of Logic and Algebraic Programming* 75(1), pp. 3–51, doi:10.1016/j.jlap.2007.06.002. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetAbadiFournetJLAP07.html`.

[13] Bruno Blanchet, Vincent Cheval & Véronique Cortier (2022): *ProVerif with lemmas, induction, fast subsumption, and much more*. In: *IEEE Symposium on Security and Privacy (S&P'22)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 205–222, doi:10.1109/SP46214.2022.00013. Available at `https://bblanche.gitlabpages.inria.fr/publications/BlanchetEtAlSP22.html`.

[14] Dorothy E. Denning & Giovanni Maria Sacco (1981): *Timestamps in Key Distribution Protocols*. *Communications of the ACM* 24(8), pp. 533–536, doi:10.1145/358722.358740.

[15] Danny Dolev & Andrew C. Yao (1983): *On the Security of Public Key Protocols*. *IEEE Transactions on Information Theory* IT-29(12), pp. 198–208, doi:10.1109/TIT.1983.1056650.

[16] Nancy Durgin, Patrick Lincoln, John C. Mitchell & Andre Scedrov (2004): *Multiset Rewriting and the Complexity of Bounded Security Protocols*. *Journal of Computer Security* 12(2), pp. 247–311, doi:10.3233/JCS-2004-12203.

[17] Simon Meier, Benedikt Schmidt, Cas Cremers & David Basin (2013): *The TAMARIN prover for symbolic analysis of security protocols*. In Natasha Sharygina & Helmut Veith, editors: *International Conference on Computer-Aided Verification (CAV'13)*, *Lecture Notes in Computer Science* 8044, Springer, pp. 696–701, doi:10.1007/978-3-642-39799-8_48.

[18] Roger M. Needham & Michael D. Schroeder (1978): *Using Encryption for Authentication in Large Networks of Computers*. *Communications of the ACM* 21(12), pp. 993–999, doi:10.1145/359657.359659.

[19] Stephan Schulz (2013): *Simple and Efficient Clause Subsumption with Feature Vector Indexing*. In Maria Paola Bonacina & Mark E. Stickel, editors: *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, *Lecture Notes in Artificial Intelligence* 7788, Springer, Berlin, Heidelberg, pp. 45–67, doi:10.1007/978-3-642-36675-8_3.

[20] Thomas Y. C. Woo & Simon S. Lam (1993): *A Semantic Model for Authentication Protocols*. In: *Proceedings IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, Los Alamitos, CA, pp. 178–194, doi:10.1109/RISP.1993.287633.