Finite models for formal security proofs

Jean Goubault-Larrecq

Finite Models for Formal Security Proofs*

Jean Goubault-Larrecq
goubault@lsv.ens-cachan.fr
LSV, ENS Cachan, CNRS, INRIA
61, avenue du président Wilson 94230 Cachan, France
Tel: +33-1 47 40 75 68 Fax: +33-1 47 40 75 21

June 9, 2021

Abstract

First-order logic models of security for cryptographic protocols, based on variants of the Dolev-Yao model, are now well-established tools. Given that we have checked a given security protocol π using a given first-order prover, how hard is it to extract a formally checkable proof of it, as required in, e.g., common criteria at the highest evaluation level (EAL7)? We demonstrate that this is surprisingly hard in the general case: the problem is non-recursive. Nonetheless, we show that we can instead extract finite models $\mathcal{M}$ from a set S of clauses representing π, automatically, and give two ways of doing so. We then define a model-checker testing $\mathcal{M} \models S$, and show how we can instrument it to output a formally checkable proof, e.g., in Coq. Experience on a number of protocols shows that this is practical, and that even complex (secure) protocols modulo equational theories have small finite models, making our approach suitable.

Keywords: Dolev-Yao model, formal security proof, finite model, tree automaton, $\mathcal{H}_1$, inductionless induction.

1 Introduction

So far, automated verification of cryptographic protocols in models in the style of Dolev and Yao [36] has been considered under a variety of angles: (un)decidability results [37, 49], practical decision procedures [65, 84, 6], extension to security properties other than secrecy and authentication (e.g., [20]), to protocols requiring equational theories, to soundness with respect to computational models (e.g., [56] for the latter two points), in particular.

We consider yet another angle: producing formally checkable proofs of security, automatically. There is indeed a more and more pressing need from the industrial community, as well as from national defense authorities, to get not just Boolean answers

*Partially supported by project PFC ("plateforme de confiance"), pôle de compétitivité System@tic Paris-région Ile-de-France. Part of this work was done during RNTL project EVA, 2000-2003.

(secure/possibly insecure), but also formal proofs, which could be checked by one of the established tools, e.g., Isabelle [70] or Coq [12]. This is required in Common Criteria certification of computer products at the highest assurance level, EAL 7 [50], a requirement that is becoming more and more common for security products. For example, the PFC initiative ("trusted platform") of the French pôle de compétitivité System@tic will include a formal security model and formal proofs for its trusted Linux-based PC platforms. Producing formal proofs for tools such as Isabelle or Coq is also interesting because of their small trusted base, and defense agencies such as the French DGA would appreciate being able to extract formal Coq proofs from Blanchet's ProVerif tool [15].

It is certainly the case that hand-crafted formal proofs (e.g., [17, 71]) provide such formally checkable certificates. Isabelle's high degree of automation helps in this respect, but can we hope for full automation as in ProVerif, and having formal proofs as well? It is the purpose of this paper to give one possible answer to that question.

One note of caution: We shall concentrate on the core of the problem, which, as we shall see, is related to a model-checking problem on sets of Horn clauses representing the protocol, the security assumptions, the intruder model, and the security properties to be proved. We consider such Horn clauses to be a sufficient, albeit rather low-level, specification language for security protocols in this paper. It is more comfortable to specify protocols and properties in higher-level languages such as ProVerif's calculus. To extend our work to such calculi, we would need not only a translation from the calculus to Horn clause sets (e.g., ProVerif already relies on one), but also a way of lifting proofs of security in the Horn clause model back to proofs of security on protocols expressed in the calculus. While we don't expect this latter to be difficult per se, we won't consider the question in this paper. As a matter of fact, all the sets of Horn clauses that we shall give as examples were produced by hand. We hope however that our way of specifying security protocols, assumptions, properties and the intruder model as Horn clauses will be clear enough. This will be explained at length in Section 3 and later.

1.1 Outline

We explore related work in Section 2, then describe our security model, à la Dolev-Yao, in Section 3. We really start in Section 4, where we show that our problem reduces to a form of model-checking, which is unfortunately undecidable in general. To solve this, we turn to finite models, expanding on Selinger's pioneering idea [77]. We observe that, although representing finite models explicitly is usually practical, it is sometimes cumbersome, and that such models are sometimes hard to find. Surprisingly, larger, finite models in the form of alternating tree automata are sometimes easier to find: we examine such models in Section 6. We then show how we can model-check clause sets against both kinds of models in Section 7. Finally, we argue that the approach is equally applicable to some security protocols that require equational theories in Section 8, and we conclude in Section 9. Our claims are supported by several practical case studies.

2 Related Work

Many frameworks and techniques have been proposed to verify security protocols in models inspired from Dolev and Yao [36]. It would be too long to cite them all. However, whether they are based on first-order proving [84, 27, 15], tree automata [65], set constraints [6], typing [1], or process algebra [4, 3], one may fairly say that most of these frameworks embed into first-order logic. It is well-known that tree automata are subsumed by set constraints, and that set constraints correspond to specific decidable classes of first-order logic. This was first observed by Bachmair, Ganzinger, and Waldmann [9]. Some modern typing systems for secrecy are equivalent to a first-order logic presentation [2], while safety properties of cryptographic protocols (weak secrecy, authentication) presented as processes in a process algebra are naturally translated to first-order logic [2], or even to decidable classes of first-order logic such as $\mathcal{H}_1$ [68].

In all cases, the fragments of first-order logic we need can be presented as sets of Horn clauses. Fix a first-order signature, which we shall leave implicit. Terms are denoted s, t, u, v, ..., predicate symbols P, Q, ..., variables X, Y, Z, ... We assume there are finitely many predicate symbols. Horn clauses C are of the form $H \Leftarrow \mathcal{B}$ where the *head* H is either an atom or $\bot$, and the *body* $\mathcal{B}$ is a finite set $A_1, \ldots, A_n$ of atoms. If $\mathcal{B}$ is empty ($n = 0$), then $C = H$ is a *fact*. For simplicity, we assume that all predicate symbols are unary, so that all atoms can be written $P(t)$. This is innocuous, as k-ary relations $P(t_1, \ldots, t_k)$ can be faithfully encoded as $P(c(t_1, \ldots, t_k))$ for some k-ary function symbol c; we shall occasionally take the liberty of using some k-ary predicates, for convenience. We assume basic familiarity with notions of free variables, substitutions σ, unification, models, Herbrand models, satisfiability and first-order resolution [8]. It is well-known that satisfiability of first-order formulae, and even of sets of Horn clauses, is undecidable. We shall also use the fact that any satisfiable set S of Horn clauses has a least Herbrand model. This can be defined as the least fixpoint $\mathrm{lfp}\, T_S$ of the monotone operator $T_S(I) = \{A\sigma \mid A \Leftarrow A_1, \ldots, A_n \in S, A\sigma \text{ ground}, A_1\sigma \in I, \ldots, A_n\sigma \in I\}$. If $\bot \in \mathrm{lfp}\, T_S$, then S is unsatisfiable. Otherwise, S is satisfiable, and $\mathrm{lfp}\, T_S$ is a set of ground atoms, which happens to be the least Herbrand model of S.

We shall concentrate on *reachability* properties (i.e., weak secrecy) in this paper, without equational theories for the most part. While this may seem unambitious, remember that our goal is not to *verify* cryptographic protocols but to extract *formally checkable proofs* automatically, and one of our points is that this is substantially harder than mere verification. We shall deal with equational theories in Section 8, and claim that producing formally checkable proofs is not much harder than in the non-equational case. We will not deal with strong secrecy, although this reduces to reachability, up to some abstraction [16]. Weak and strong secrecy are, in fact, close notions under reasonable assumptions [31].

We also concentrate on security proofs in *logical* models, derived from the Dolev-Yao model [36]. Proofs in *computational* models would probably be more relevant. E.g., naive Dolev-Yao models may be computationally unsound [64]. However, some recent results show that symbolic (Dolev-Yao) security implies computational security in a number of frameworks, usually provided there are no key cycles at least, and modulo properly chosen equational theories on the symbolic side. See e.g. [51], or

[79]. The latter is a rare example of a framework for developing formal proofs (e.g., in Coq or Isabelle) of *computational soundness* theorems. The search for such theorems is hardly automated for now; yet, we consider this to be out of the scope of this paper, and concentrate on Dolev-Yao-like models.

The starting point of this paper is Selinger's fine paper "Models for an Adversary-Centric Protocol Logic" [77]. Selinger observes that security proofs (in first-order formulations of weak secrecy in Dolev-Yao-like models) are *models*, in the sense of first-order logic. To be a bit more precise, a protocol π encoded as a set of first-order Horn clauses S is secure if and only if S is *consistent*, i.e., there is no proof of false $\bot$ from S. One may say this in a provocative way [41] by stating that a proof of security for π is the *absence* of a proof for (the negation of) S. Extracting a formal Coq proof from such an absence may then seem tricky. However, first-order logic is *complete*, so if S is consistent, it must be *satisfiable*, that is, it must have a model. Selinger then observed that you could prove π secure by exhibiting a model for S, and demonstrated this by building a small, finite model (5 elements) for the Needham-Schroeder-Lowe public-key protocol [66, 59]. We shall demonstrate through several case studies that even complex protocols requiring rather elaborate equational theories can be proved secure using finite models with only few elements. However, we shall observe that even such models may be rather large, and harder than expected to check.

The idea of proving properties by showing the consistency of a given formula F, i.e, showing that $\neg F$ has no proof, is known as *proof by consistency* [53], or *inductionless induction* [58, 24]. Note that the formal Coq proofs we shall extract from models of S, using our tool `h1mc`, are proofs of security for π that work by (explicit) induction over term structure. The relationship between inductionless and explicit induction was elicited by Comon and Nieuwenhuis [26], in the case of first-order logic with equality and induction along the recursive path ordering.

We shall use two approaches to extracting a formal proof of security from a finite model. The first one is based on explicit enumeration. The second is an approach based on model-checking certain classes of first-order formulae F against certain classes of finite models $\mathcal{M}$, i.e., on testing whether $\mathcal{M} \models F$. There is an extensive body of literature pertaining to this topic, see e.g. the survey by Dawar [32]. One particular (easy) result we shall recall is that model-checking first-order formulae against finite models, even of size 2, is **PSPACE**-complete. Many results in this domain have focused on fixed-parameter tractability, and to be specific, on whether model-checking was hard with respect to the size of the model, given a fixed formula as parameter. Even then, the parametrized model-checking problem is $\mathbf{AW}[*]$-complete, and already $\mathbf{W}[k]$-hard for Π_k formulae. This will be of almost no concern to us, as our formulae F will grow in general faster than our models.

Since our presentation at CSF'08 [43], we learned that Matzinger [62] had already designed, in 1997, what is essentially our model-checking algorithm of Section 7, restricted to non-alternating automata, with an explicitly defined strategy for rule application, a different presentation, none of the optimizations that are in fact required in practice, and no report of an implementation.

None of the works cited above addresses the question of extracting a model from a failed proof attempt. Tammet worked on this for resolution proofs [81]. The next step, producing formally checked, inductive proofs from models, seems new. In one

of our approaches, finite models will be presented in the form of tree automata, and formally checking models in this case essentially amounts to producing formal proofs of computations on tree automata. This was pioneered by Rival and the author [74]; the procedure of Section 7 is several orders of magnitude more efficient.

3 A Simple Protocol Model, à la Dolev-Yao

Our first-order model for protocols is close to Blanchet's [14], to Selinger's [77], and to a number of other works. While the actual model is not of paramount importance for now, we need one to illustrate our ideas. Also, models in the style presented here will behave nicely in later sections.

Blanchet uses a single predicate $\mathtt{att}$, so that $\mathtt{att}(M)$ if and only if M is known to the Dolev-Yao attacker. We shall instead use a family of predicates $\mathtt{att}_i$, where i is a *phase*, to be able to model key and nonce corruption (more below). The facts that the Dolev-Yao attacker can encrypt, decrypt, build lists, read off any element from a list, compute successors and predecessors are axiomatized by the Horn clauses of Figure 1. We take the usual Prolog convention that identifiers starting with capital letters such as M, K, A, B, X, are variables, while uncapitalized identifiers such as $\mathtt{sym}$, $\mathtt{crypt}$, $\mathtt{att}$ are constants, function or predicate symbols. We let $\mathtt{crypt}(M, K)$ denote the result of symmetric or asymmetric encrypting M with key K, and write it $\{M\}_K$ for convenience. The key $\mathtt{k}(\mathtt{sym}, X)$ is the symmetric key used in session X; the term $\mathtt{session}_i(A, B, N_a)$ will denote any session between principals A and B sharing the nonce N_a, while in phase i; we shall also use $\mathtt{k}(\mathtt{sym}, [A, S])$ to denote long-term symmetric keys between agents A and S. The key $\mathtt{k}(\mathtt{pub}, X)$ denotes agent X's long-term public key, while $\mathtt{k}(\mathtt{prv}, X)$ is X's private key. Lists are built using $\mathtt{nil}$ and $\mathtt{cons}$; we use the ML notation $M_1 :: M_2$ for $\mathtt{cons}(M_1, M_2)$, and $[M_1, M_2, \ldots, M_n]$ for $M_1 :: M_2 :: \ldots :: M_n :: \mathtt{nil}$. We use $\mathtt{suc}$ to denote the successor function $\lambda n \in \mathbb{N} \cdot n + 1$, as used in our running example, the Needham-Schroeder symmetric key protocol [66].

$$\mathtt{att}_i(\{M\}_K) \Leftarrow \mathtt{att}_i(M), \mathtt{att}_i(K) \tag{1}$$
$$\mathtt{att}_i(M) \Leftarrow \mathtt{att}_i(\{M\}_{\mathtt{k}(\mathtt{pub},X)}), \mathtt{att}_i(\mathtt{k}(\mathtt{prv}, X)) \tag{2}$$
$$\mathtt{att}_i(M) \Leftarrow \mathtt{att}_i(\{M\}_{\mathtt{k}(\mathtt{prv},X)}), \mathtt{att}_i(\mathtt{k}(\mathtt{pub}, X)) \tag{3}$$
$$\mathtt{att}_i(M) \Leftarrow \mathtt{att}_i(\{M\}_{\mathtt{k}(\mathtt{sym},X)}), \mathtt{att}_i(\mathtt{k}(\mathtt{sym}, X)) \tag{4}$$
$$\mathtt{att}_i(\mathtt{nil}) \tag{5}$$
$$\mathtt{att}_i(M_1 :: M_2) \Leftarrow \mathtt{att}_i(M_1), \mathtt{att}_i(M_2) \tag{6}$$
$$\mathtt{att}_i(M_1) \Leftarrow \mathtt{att}_i(M_1 :: M_2) \tag{7}$$
$$\mathtt{att}_i(M_2) \Leftarrow \mathtt{att}_i(M_1 :: M_2) \tag{8}$$
$$\mathtt{att}_i(\mathtt{suc}(M)) \Leftarrow \mathtt{att}_i(M) \tag{9}$$
$$\mathtt{att}_i(M) \Leftarrow \mathtt{att}_i(\mathtt{suc}(M)) \tag{10}$$

Figure 1: Intruder capabilities

This protocol, whose purpose is to establish a fresh, secret session key K_{ab} between two agents, Alice (A) and Bob (B), using a trusted third party (S), is shown in Figure 2. It has the convenient property that there is a well-known attack against it, so that the key K_{ab} that Bob will end up having is possibly known to the attacker, while the keys K_{ab} that S sent and that Alice received will remain secret. Note that all three keys K_{ab} may be different.

1.	$A \longrightarrow S : A, B, N_a$
2.	$S \longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3.	$A \longrightarrow B : \{K_{ab}, A\}_{K_{bs}}$
4.	$B \longrightarrow A : \{N_b\}_{K_{ab}}$
5.	$A \longrightarrow B : \{N_b + 1\}_{K_{ab}}$

Figure 2: The Needham-Schroeder symmetric-key protocol

The protocol itself is modeled in a simple way, originally inspired from strand spaces [82], and similarly to Blanchet [14]. Each agent's role is modeled as a sequence of (receive, send) pairs. Given any such pair (M_1, M_2), we add a Horn clause of the form $\mathtt{att}_i(M_2) \Leftarrow \mathtt{att}_i(M_1)$. This denotes the fact that the attacker may use the agent's role to his profit by sending a message M_1 of a form that the agent will accept, and learning M_2 from the agent's response. Accordingly, the protocol rules for the Needham-Schroeder symmetric key protocol are shown in Figure 3. We use Blanchet's trick of abstracting nonces by function symbols applied to the free parameters of the session, so that $\mathtt{na}_i(A, B)$ denotes N_a, depending on the identities A and B of Alice and Bob respectively and the phase i, and $\mathtt{nb}_i(K_{ab}, A, B)$ denotes N_b, depending on the phase i, the received key K_{ab}, and identities A and B (all three being variables, by our convention). In clause (15), representing the fact that Alice receives $\{N_b\}_{K_{ab}}$ (message 4 of Figure 2) to send $\{N_b+1\}_{K_{ab}}$ (message 5), we use an auxiliary predicate $\mathtt{alice\_key}_i$ to recover Alice's version of K_{ab}, received in message 2. We also define a predicate $\mathtt{bob\_key}_i$ in (17) to recover Bob's version of K_{ab} after message 5.

$$\mathtt{att}_i([A, B, \mathtt{na}_i(A, B)]) \Leftarrow \mathtt{agent}(A), \mathtt{agent}(B) \tag{11}$$

$$\mathtt{att}_i(\{[N_a, B, k_{ab}, \{[k_{ab}, A]\}_{k_{bs}}]\}_{k_{as}})) \Leftarrow \mathtt{att}_i([A, B, N_a]) \tag{12}$$

$$\text{where } k_{ab} = \mathtt{k}(\mathtt{sym}, \mathtt{session}_i(A, B, N_a),$$

$$k_{bs} = \mathtt{k}(\mathtt{sym}, [B, \mathtt{s}]), k_{as} = \mathtt{k}(\mathtt{sym}, [A, \mathtt{s}])$$

$$\mathtt{att}_i(M) \Leftarrow \mathtt{att}_i(\{[\mathtt{na}_i(A, B), B, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}) \tag{13}$$

$$\mathtt{att}_i(\{\mathtt{nb}_i(K_{ab}, A, B)\}_{K_{ab}}) \Leftarrow \mathtt{att}_i(\{[K_{ab}, A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])}) \tag{14}$$

$$\mathtt{att}_i(\{\mathtt{suc}(N_b)\}_{K_{ab}}) \Leftarrow \mathtt{att}_i(\{N_b\}_{K_{ab}}), \mathtt{alice\_key}_i(A, K_{ab}) \tag{15}$$

$$\mathtt{alice\_key}_i(A, K_{ab}) \Leftarrow \mathtt{att}_i(\{[\mathtt{na}_i(A, B), B, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}) \tag{16}$$

$$\mathtt{bob\_key}_i(B, K_{ab}) \Leftarrow \mathtt{att}_i(\{\mathtt{nb}_i(K_{ab}, A, B)\}_{K_{ab}}) \tag{17}$$

Figure 3: Protocol rules

$\texttt{agent(a)}$ $\texttt{agent(b)}$ $\texttt{agent(i)}$ $\texttt{agent(s)}$

Figure 4: Agents

The fact that variables such as A, B are used throughout for agent identities, instead of actual agent identities (for which we reserve the constants `a`, `b`, `s`, and `i` for the attacker), is due to the fact that we wish to model unboundedly many sessions of the protocol in parallel. E.g., (11) states that any pair of agents A, B may initiate the protocol and emit message 1 of Figure 2. We assume that the only possible agents are Alice (`a`), Bob (`b`), the trusted third-party (`s`), and the Dolev-Yao attacker `i`. Since we only deal with secrecy, considering this many agents is sufficient [27].

This way of modeling protocols has been standard at least since Blanchet's seminal work [14]. One should however note that this is only a safe approximation of the protocol, not an exact description as in [27] for example. In particular, our encoding forgets about the relative orderings of messages. In particular, if the intruder sends some message M to A, then A uses M to compute another message M' to B, then our model will make $\texttt{att}_i(M')$ true. This means that M' will be known to the intruder forever, so that replay attacks are accounted for. This can also be taken to mean, as a much stranger consequence, that we estimate that the intruder will have known M' all the time in the past as well, including at the times it was preparing the first message M. Closer inspection reveals that what we are really modeling here is the fact that several sessions of the same protocol can run in parallel, asynchronously: the intruder sends M to A in the first session, A sends back M' to B in the first session, then the intruder uses this M' to compute another message M_1 sent to A, who starts a second session and sends M_1' to B, and so on. For further discussion on the implications of this way of modeling protocols, see Blanchet's paper, cited above [14]. Blanchet also discusses why this is a *safe* approximation (Section 2.5), i.e., is there is any attack at all, then there will be a proof of $\bot$ from our clause set.

This approximation is precise enough in most cases. Some cases where it is not include protocols with timestamps (see Section 5), protocols that combine two sub-protocols, one where long-term keys are established, and a second one where these are used to exchange short-term keys, and finally protocols where we assume that some secrets can be corrupted over time. We model all three situations by distinguishing two *phases* $i = 1, 2$. More are possible. The phase i labels the `att` predicate, i.e., we have two predicates $\texttt{att}_1$ and $\texttt{att}_2$ instead of just one predicate `attacker` as in [14].

Let us deal with the case of secret corruption. Intuitively, phase 1 represents sessions that are old enough that the old session keys $\texttt{k}(\texttt{sym}, \texttt{session}_1(A, B, Na))$ may have been guessed or discovered by the intruder. This is (again) a conservative approximation: we estimate that all old secrets (in phase 1) are compromised, although only some or even none of them may have been actually compromised. On the other hand, no secret in phase 2 is compromised—unless the protocol itself leaks them. To model phases, we only need a few more clauses, shown in Figure 5: (18) states that the intruder has memory, and remembers all old messages from phase 1 in phase 2, while the

$$\mathtt{att}_2(M) \Leftarrow \mathtt{att}_1(M) \tag{18}$$
$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_1(A, B, N_a))) \tag{19}$$
$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{na}_1(A, B))) \tag{20}$$
$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{nb}_1(K_{ab}, A, B))) \tag{21}$$

Figure 5: Phases

$$\mathtt{att}_i(X) \Leftarrow \mathtt{agent}(X)$$
$$\mathtt{att}_i(\mathtt{k}(\mathtt{pub}, X)) \qquad \mathtt{att}_i(\mathtt{k}(\mathtt{prv}, \mathtt{i}))$$

Figure 6: The attacker's initial knowledge

other clauses state that all old session keys, as well as all old nonces, are compromised. This is similar, e.g., to Paulson's Oops moves [71].

Figure 6 lists our security assumptions, i.e., what we estimate the attacker might know initially: all agent identities are known, as well as all public keys $\mathtt{k}(\mathtt{pub}, X)$, and the attacker's own private key $\mathtt{k}(\mathtt{prv}, \mathtt{i})$—whatever the phase. Note that talking about public and private keys in this protocol, which only uses symmetric keys, is overkill. We include them to illustrate the fact that the model is not limited to symmetric key encryption, and public-key protocols would be encoded just as easily.

Finally, Figure 7 lists our security goals, or rather their negated forms. Note that we are only concerned with the security of phase 2 data, since phase 1 is compromised by nature. Negation comes from the fact that a formula G is a consequence of a set S of clauses such as those listed above if and only if $S, \neg G$ is inconsistent. E.g., (22) is the negation of $\exists N_a \cdot \mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_2(\mathtt{a}, \mathtt{b}, N_a)))$, and corresponds to asking whether the secret key K_{as}, as generated by the trusted third-party in current sessions, can be inferred by the attacker. (23) asks whether there is a key K_{ab} that would be both known to the attacker, and is plausibly accepted by Alice ($\mathtt{a}$) as its new symmetric key; we again use the auxiliary predicate $\mathtt{alice\_key}_2$. Finally, (24) asks whether there is a key K_{ab} as could be used in the final check of the protocol by Bob (message 5 of Figure 2), and that would be, in fact, compromised.

$$\bot \Leftarrow \mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_2(\mathtt{a}, \mathtt{b}, N_a))) \tag{22}$$
$$\bot \Leftarrow \mathtt{att}_2(K_{ab}), \mathtt{alice\_key}_2(\mathtt{a}, K_{ab}) \tag{23}$$
$$\bot \Leftarrow \mathtt{att}_2(K_{ab}), \mathtt{bob\_key}_2(\mathtt{b}, K_{ab}) \tag{24}$$

Figure 7: (Negated) security goals

Call, somewhat abusively, the *protocol* π the collection comprised of the cryptographic protocol itself, the (Dolev-Yao) security model, the security assumptions and

the security goals. Let S_π be the corresponding clause set. The clause set S_{NS} denoting the symmetric-key Needham-Schroeder protocol NS is then the union of the clauses in Figure 1 ($i = 1, 2$), Figure 3 ($i = 1, 2$), Figure 6 ($i = 1, 2$), Figure 4, Figure 5, and Figure 7.

Unsurprisingly, running a first-order prover against S_{NS} reveals a possible attack against Bob. E.g., SPASS v2.0 [86] finds that the above set of clauses is inconsistent, with a small resolution proof, where only 309 clauses were derived, in 0.07 seconds on a 2.4 GHz Intel Centrino Duo class machine. Examining the proof reveals that the attack is actual. This is the well-known attack where the attacker uses an old message 3 from a previous session (for which K_{ab} is now known), and replays it to Bob. The attacker can then decrypt message 4, since he knows K_{ab}, and Bob will accept message 5 as confirmation.

Removing the failing security goal (24) produces a consistent set of clauses S_{NS}^{safe}: so there is no attack on the other two security goals. This seems to be out of reach for SPASS (at least without any specific option): after 10 minutes already, SPASS is lost considering terms with 233 nested applications of the successor function `suc`; we decided to stop SPASS after 4h 10 min, where this number had increased to 817. However, our own tool `h1`, from the `h1` tool suite [40], shows both that there is a plausible attack against Bob and definitely no attack against Alice or the trusted third-party, in 0.68 s; `h1` works by first applying a canonical abstraction to the given clause set S [42, Proposition 3], producing an approximation S' in the decidable class $\mathcal{H}_1$ [68, 84]; then `h1` decides S' by the terminating resolution algorithm of [42]. We refer to this paper for details. We shall return to this approach in Section 6.

4 Undecidability

An intuitive idea to reach our goal, i.e., producing formal proofs from a security proof discovered by a tool such as ProVerif, SPASS or `h1`, is to instrument it so as to return a trace of its proof activity, which we could then convert to a formal proof. However, this cannot be done. As illustrated on S_{NS}^{safe}, the protocol, without the security goal (24), is secure because we *cannot* derive any fact of the form $\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_2(\mathtt{a}, \mathtt{b}, n_a)))$ for any term n_a, and there is no term k_{ab} such that both $\mathtt{att}_2(k_{ab})$ and $\mathtt{alice\_key}_2(\mathtt{a}, k_{ab})$ would be derivable. In short, security is demonstrated through the *absence* of a proof.

It would certainly be pointless to instrument ProVerif, SPASS or `h1` so as to document everything it *didn't* do. However, these tools all work by saturating the input clause set S representing the protocol π to get a final clause set S_∞, using some form of the resolution rule, and up to specific redundancy elimination rules. To produce a formally checkable security proof of the protocol π—in case no contradiction is derived from S—, what we can therefore safely assume is: (A) S_∞ is consistent, (B) S_∞ is entailed by S, and (C) S_∞ is saturated up to redundancy (see [8]).

Bruno Blanchet kindly reminded me that point (C) could in principle be used to produce a formal proof that π is secure. We have to: (a) prove formally that the saturation procedure is complete, in the sense that whenever S_∞ is saturated up to redundancy, and every clause in S is redundant relative to S_∞, then S is consistent; and: (b) pro-

duce a formal proof that S_∞ is indeed saturated up to redundancy. Task (b) is complex, and complexity increases with the sophistication of the saturation strategy; realize that even the mundane task of showing, in Isabelle or Coq, that two given literals do not unify requires some effort. Moreover, S_∞ is in general rather large, and task (b) will likely produce long proofs. Task (a) is rather formidable in itself. However, a gifted and dedicated researcher might be able achieve as much, as suggested to me by Cédric Fournet, while the effort in achieving (b) is likely to be comparable to the one we put into our tool `h1mc` (Section 7).

We believe that the most serious drawback of this approach is in fact non-maintainability: (a) and (b) have to be redone for each different saturation procedure, i.e., for different tools, or when these tools evolve to include new redundancy elimination rules or variants of the original resolution rule.

This prompts us to use only points (A) and (B) above, not (C). Fortunately, and this is one of the points that Selinger makes [77], a clause set is consistent if and only if it has a *model*. We may therefore look for models of S as witnesses of security for π. While Selinger proposes this approach to check whether π is secure, it can certainly be used to fulfill our purpose: assume that we know that S is consistent, typically because ProVerif, SPASS or `h1`, has terminated on a clause set S_∞ that is saturated under some complete set of logical rules (forms of resolution in the cited provers) and which does not contain the empty clause $\bot$; then our tasks reduces to answering two questions: (1) how can we extract a model from a saturated set of clauses S_∞ not containing $\bot$? (2) given a model $\mathcal{M}$ that acts as a certificate of satisfiability, hence as a certificate of security for π, how do we convert $\mathcal{M}$ to a formal Coq proof?

Question (2) is not too hard, at least in principle: produce a proof that $\mathcal{M}$ satisfies each clause C in S, by simply enumerating all possible values of each free variable in C, and checking that this always yields "true". For larger models, we shall see that we can instead build a model-checking algorithm to check whether $\mathcal{M}$ satisfies S (in notation, $\mathcal{M} \models S$), and keep a trace of the computation of the model-checker. Then we convert this trace into a formal proof. We shall see how to do this in Section 7.

Question (1) is easy, but ill-posed, because we did not impose any restriction on the format the model should assume. (Note that we don't know whether $\mathcal{M}$ is finite, in particular in the cases of SPASS and ProVerif.) The answer is that S_∞ is itself a perfectly valid description of a model, namely the unique least Herbrand model $\mathrm{lfp}\, T_{S_\infty}$ of S_∞ (I owe this simple remark to Andreas Podelski, personal communication, 1999). What this model lacks, at least, is being effective: there is in general no way of testing whether a given ground atom A holds or not in this model. In our case, the important result is the following, which shows that we cannot in general even test whether $\mathcal{M} \models S$, where $\mathcal{M} = \mathrm{lfp}\, T_{S_\infty}$, contradicting our goal (2).

Proposition 4.1 *The following problem is undecidable: Given a satisfiable set of first-order Horn clauses S_∞, and a set of first-order Horn clauses S, check whether the least Herbrand model of S_∞ satisfies S. This holds even if S contains just one ground unit clause, and S_∞ contains only three clauses.*

Proof. By [34], the satisfiability problem for clause sets S_1 consisting of just three clauses $p(fact)$, $p(left) \Leftarrow p(right)$, and $\bot \Leftarrow p(goal)$ is undecidable. Take S_∞ to

consist of the clauses $p(fact)$, $p(left) \Leftarrow p(right)$, and $q(*) \Leftarrow p(goal)$, where q is a fresh predicate symbol and $*$ a fresh constant. Take S to contain just the clause $q(*)$.

Note that S_∞, as a set of definite clauses, is satisfiable. We claim that S_1 is unsatisfiable if and only if $\mathrm{lfp}\, T_{S_\infty}$ satisfies S. If S_1 is unsatisfiable, then $\bot \in \mathrm{lfp}\, T_{S_1} = T_{S_1}(\mathrm{lfp}\, T_{S_1})$. By definition of T_{S_1}, and since $\bot \Leftarrow p(goal)$ is the only clause of S_1 with head $\bot$, there is a ground instance $p(goal\ \sigma)$ in $\mathrm{lfp}\, T_{S_1}$. Now $\mathrm{lfp}\, T_{S_1} = \bigcup_{n\in\mathbb{N}} T^n_{S_1}(\emptyset)$, since the T_{S_1} operator is Scott-continuous. By an easy induction on n (which, intuitively, is proof length), every atom of the form $p(t)$ in $T^n_{S_1}(\emptyset)$ is in $T^n_{S_\infty}(\emptyset)$. So $p(goal\ \sigma)$ is in $\mathrm{lfp}\, T_{S_\infty}$, whence $q(*)$ is in the least Herbrand model of S_∞, i.e., the latter satisfies S. Conversely, if $\mathrm{lfp}\, T_{S_\infty}$ satisfies S, that is, $q(*)$, by similar arguments we show that it must satisfy some instance $p(goal\ \sigma)$, which is then in $\mathrm{lfp}\, T_{S_1}$, so that S_1 is unsatisfiable. □

Despite the similarity, this theorem is not a direct consequence of Marcinkowski and Pacholski's result [61], that the Horn clause implication problem $C_1 \models C_2$ is undecidable. Recall that $C_1 \models C_2$ whenever every model of C_1 satisfies C_2. Indeed, this is not equivalent to (not entailed by, to be precise) the fact that the least Herbrand model of C_1 satisfies C_2.

Replacing the ground unit clause $q(*)$ of S above by $\mathtt{att}_1(*)$ shows that:

Corollary 4.2 *The following problem is undecidable: given a satisfiable set of first-order Horn clauses S_∞, check whether* $\mathrm{lfp}\, T_{S_\infty}$ *is a model of a first-order formulation of a cryptographic protocol π. This holds even if π contains absolutely no message exchange (i.e., the number of protocol steps is zero), has only one phase, the initial knowledge of the intruder consists of just one ground message $*$, the Dolev-Yao intruder has no deduction capability at all (i.e., we don't include any of the rules of Figure 1), and the number of security goals is zero.*

To mitigate this seemingly devastating result, recall that SPASS and ProVerif use variants of resolution, and the clause sets S_∞ produced by SPASS or ProVerif are saturated up to redundancy. SPASS uses sophisticated forms of ordered resolution with selection and sorts, while ProVerif uses two restrictions of resolution. "Saturated up to redundancy" [8] means that every conclusion of the chosen resolution rule with premises in S_∞ is either already in S_∞, or redundant with respect to S_∞, e.g., subsumed by some clause in S_∞. It is well-known that, for all variants of resolution that can be shown complete by Bachmair and Ganzinger's forcing technique [8], the models of a set S_∞ that is saturated up to redundancy are exactly the models of the subset $S_{prod} \subseteq S_\infty$ of all the so-called *productive* clauses of S_∞. In particular, for Horn clauses, $\mathrm{lfp}\, S_{prod} = \mathrm{lfp}\, S_\infty$. For example, the first phase of the ProVerif algorithm uses a form of resolution with selection, where all literals of the form $\mathtt{att}_i(M)$ are selected in clause bodies. The effect is that the clauses of S_{prod} cannot contain any literal of the form $\mathtt{att}_i(M)$ in their body. It is then a trivial observation that Proposition 4.1 still holds with S_∞ replaced by S_{prod} (just make p different from $\mathtt{att}_i$). However, this first phase is not a complete procedure in itself. Ordered resolution with selection [8], as well as the kind of resolution used in SPASS [85] are complete. Using the former for example, S_{prod} consists of clauses where no atom is selected in any clause body, and the head is maximal with respect to the chosen stable, well-founded ordering $\succ$. Even so, this does not help in general:

Proposition 4.3 *Proposition 4.1 and Corollary 4.2 still hold if S_∞ is replaced by a set S_{prod} of productive clauses, again even of cardinality 3.*

Proof. Modify the construction of S_∞ slightly, and take it to consist of $p(\mathtt{c}, fact)$, $p(\mathtt{f}(X), left) \Leftarrow p(X, right)$, and $q(*) \Leftarrow p(X, goal)$. Let $\succ$ be defined by $q(M) \succ p(N)$ for every terms M and N, and $p(M, N) \succ p(M', N')$ if and only if M' is a proper subterm of M. Select no atom in any clause body. Then $S_{prod} = S_\infty$ is a set of productive clauses. As in Proposition 4.1, S_1 is unsatisfiable if and only if $\mathrm{lfp}\, T_{S_{prod}} \models q(*)$. □

5 Explicit, Finite Models

There is a much simpler solution: directly find *finite* models $\mathcal{M}$ of the set S of clauses representing the protocol π. This won't enable us to verify protocols that are secure because S is satisfiable, but not finitely satisfiable. But again Selinger's early experiments [77] suggest that this is perhaps not a problem in practice. To wit, remember that there is a 5 element model for Selinger's encoding of the Needham-Schroeder-Lowe public-key protocol. In fact, our encoding of the 7-message Needham-Schroeder-Lowe protocol has a 4 element model, found by Koen Claessen's tool Paradox. As for our running example, our tool `h1` finds a 46 element model for S_{NS}^{safe} (see Section 3), but there is also a 4 element model (see below).

There are certainly protocols which could only be shown secure using techniques requiring infinite models. In particular, this is likely for parametric verification of *recursive* protocols—where by parametric we mean that verification should conclude for all values of an integer parameter n, typically the number of participants or the number of rounds. Solving first-order clause sets representing such protocols was addressed by Küsters and Truderung [57]. Examples of such protocols are Bull and Otway's recursive authentication protocol [18], or the IKA protocols [80]. Note however that both are flawed [76, 73], so that S would in fact be unsatisfiable in each case.

Finding finite models of first-order clause sets is a sport, and is in particular addressed in the finite model category of the CASC competition at annual CADE conferences. Paradox [21] is one such model-finder, and won the competition in 2003, 2004, 2005, 2006. Paradox v2.3 finds a 4 element model for S_{NS}^{safe} (see Section 3), in 1.6 s. Due to the algorithm used by Paradox, this also guarantees that there is no 3 element model. We have also tested other model finders, such as Mace4 [63] or the experimental tool Darwin [10]. None returned on any of the examples that we tested them on, in a time limit of two hours (and sometimes more).

Paradox represents finite models in the obvious way, as tables representing the semantics of functions, and truth-tables representing predicates. Call these *explicitly presented* models. The explicitly presented model found by Paradox on S_{NS}^{safe} has 4 element !1, !2, 3, and !4. All identities `a`, `b`, `i`, `s` have value !1; this is also the value of `nil`, `prv` and `pub`, while the value of `sym` is !2. The $\mathtt{att}_1$ predicate holds of value !1 only, while $\mathtt{att}_2$ holds of !1 and !2 only. The table for encryption is shown in Figure 8: that `crypt` applied to !2 and !1 yields !2 then means that encrypting a message learned between phase 1 and phase 2 (with value !2) with a key that was already known in

crypt	!1	!2	!3	!4
!1	!1	!1	!4	!1
!2	!2	!1	!4	!4
!3	!3	!4	!4	!4
!4	!3	!2	!2	!2

Figure 8: `crypt`, in Paradox's 4 element model

phase 1 (with value !1) cannot be known in phase 1 but will have been learned by the time we enter phase 2 (i.e., it has value !2). It is also useful to think of these values as pairwise disjoint *types*: messages of type !1 are those known in phase 1, messages of type !2 are those known in phase 2 but unknown in phase 1. The other values (or types) are harder to interpret. Both !3 and !4 can be thought as types of messages that will remain secret even after phase 2. The only difference between them is that encrypting messages of type !4 with data of type !2 (known in phase 2 but not before) will produce ciphertexts that are known in phase 2 (of type !2), while messages of type !3 are safer, in the sense that encrypting them with data of type !2 yield ciphertexts of type !4, which remain unknown even in phase !2 (but don't encrypt them twice).

Model-checking clause sets S against such small models $\mathcal{M}$, represented as tables, i.e., checking whether $\mathcal{M} \models S$, is straightforward, and works in polynomial time, assuming the number of free variables in each clause of S is bounded: let k be the largest number of free variables in clauses of S, n the number of elements in $\mathcal{M}$, then for each clause C in S, enumerate the at most n^k tuples ρ of values for the variables of C, then check that C under ρ is true. Call one such step of verification that C holds under ρ a *check*. In the example of Section 3, $k = 4$, there are 50 clauses with at most 5 free variables: a conservative estimate shows that we need at most $50 \times 4^5 = 51\ 200$ checks. A precise computation shows that we need $8.4^0 + 11.4^1 + 17.4^2 + 8.4^3 + 4.4^4 + 2.4^5 = 3\ 908$ checks.

However, the assumption that the number of free variables is bounded is important in the latter paragraph. In general, using the same construction that the one showing that model-checking first-order formulae against finite models is **PSPACE**-complete, we obtain:

Lemma 5.1 *Checking whether $\mathcal{M} \models S$, where $\mathcal{M}$ is an explicitly presented finite model and S is a set of Horn clauses, is* **coNP***-complete, even when $\mathcal{M}$ is restricted to 2-element models and S contains just one positive, unit clause.*

Proof. We show that checking $\mathcal{M} \not\models S$ is **NP**-complete. Membership in **NP** is easy: guess an unsatisfied clause C in S and values for its free variables. Conversely, we show that the problem is **NP**-hard already when $\mathcal{M}$ is the two-element model $\{0, 1\}$, with one predicate `true`, which holds of 1 but not of 0. We also assume term constants `t` (denoting 1), `f` (denoting 0), `and` (denoting logical conjunction), `or` (denoting logical disjunction), `not` (denoting negation). We are now ready to reduce SAT: let the input be a set S_0 of propositional clauses on a vector $\vec{A}$ of propositional variables, seen as

1.	$A \longrightarrow S : A, B$
2.	$S \longrightarrow A : \{K_b, B\}_{K_s^{-1}}$
3.	$A \longrightarrow B : \{N_a, A\}_{K_b}$
4.	$B \longrightarrow S : B, A$
5.	$S \longrightarrow B : \{K_a, A\}_{K_s^{-1}}$
6.	$B \longrightarrow A : \{N_a, N_b, B\}_{K_a}$
7.	$A \longrightarrow B : \{N_b\}_{K_b}$

Figure 9: The 7-Message NSL Protocol

a conjunction $F(\vec{A})$. Build a first-order term $F^*(\vec{A})$, where now the variables in $\vec{A}$ are seen as term variables, by replacing ands ($\wedge$) by `and`, ors ($\vee$) by `or`, negations ($\neg$) by `not`, and so on, in $F(\vec{A})$. Let S consist of the unique positive unit clause $\texttt{true}(\texttt{not}(F^*(\vec{A})))$. Clearly $\mathcal{M} \models S$ if and only if S_0 is not satisfiable. □

What this lemma illustrates, and what practice confirms, is that it is not so much the number k of elements of the model that counts, or even the number of entries in its tables, but what we called the number of *checks* needed. Both the number of entries in the tables and the number of checks can be exponentially large. However, the approach is, as we shall see, practical.

We have conducted an experiment on several secrecy protocols. Results are to be found in Figure 10, and we shall comment on the protocols shortly. Most were found in the Spore library [78]. The only exceptions are EKE and EAP-AKA (see below). The reader should be aware that the proportion of secrecy protocols that are in fact secure is small, and, despite our trying to avoid vulnerable protocols, we actually lost some time experimenting with some other protocols that eventually turned out not to be secure. (E.g., although the Kao-Chow protocol [52] is well-known to be vulnerable, the Kao Chow Authentication v.3 protocol is not reported as vulnerable in SPORE; however we found out that it was subject to an attack.)

The NS row is our example S_{NS}^{safe}, while the amended NS row is a corrected version [67] that satisfies all required security properties. Paradox always finds smallest possible models, since it looks for models of size k for increasing values of k. On the other hand, `h1` is a resolution prover that decides the class $\mathcal{H}_1$, all of whose satisfiable formulae have finite models; the models extracted are in particular not minimal in any way. We report figures found by `h1` so as to appreciate how even small models in terms of number of elements (e.g., 57 for the amended NS protocol) are in fact large in practice (e.g., 188 724 entries—we actually report a number of transitions in a deterministic tree automaton describing the model, as explained in Section 6, and this is a *lower bound* on the actual number of entries), and require many checks (e.g., more than one billion).

The NSL7 row is the 7-message version of the Needham-Schroeder-Lowe protocol, checking that the secrecy of the exchanged messages is preserved, instead of mutual authentication. See Figure 9. Contrarily to the above protocols, this is an asymmetric key protocol. The messages 1, 2, 4 and 5, which are usually left out of models of this

Protocol	Paradox				h1			
	Time	#elts	#entries	#checks	Time	#elts	#entries	#checks
NS	1.62s	4	824	3 908	0.70s	46	217 312	430 10^6
amended NS [67]	($\geq$ 30 872s)	(≥ 5)	–	–	1.71s	57	188 724	1.245 10^9
NSL7 [67, 59]	4.85s	4	2 729	2 208	8.03s	over-approximated		
Yahalom [72]	3 190s	6	5 480	38 864	4.82s	≥ 57		$\geq$ 2.46 10^9
Kerberos [19]	17.87s	5	1 767	5 518	0.94s	57	7 952	84.5 10^6
X.509 [78]	3 395s	4	142 487	12 670	0.44s	≥ 29		$\geq$ 228.5 10^6
EAP-AKA [7]	54.3s	3	2 447	1 457	1.93s	72	22 550	7.74 10^9
EKE [11]	0.44s	4	3 697	4 632	1.88s	48	16 016	64.5 10^6

Figure 10: Model sizes

protocol, are meant for A and B to get their peer's respective public keys K_b and K_a from the server S. This is a rare example where the standard approximation strategy of `h1` fails (without added tricks), and `h1` does not conclude that the protocol is safe; Paradox finds a 4 element model, showing it is indeed safe.

The Yahalom row is Paulson's corrected version of the Yahalom protocol [72]. While it is found secure by `h1` in 4.8 s, the model found (in implicit form, see Section 6) is so big that we have been unable to convert it to an explicit representation in 2 GB of main memory using our determinizer `pldet`. However, note that `h1` *did* find a model—it is just too big to print in an explicit form. The same thing happened on the X.509 row.

The Kerberos row is Burrows, Abadi and Needham's [19, Section 6] simplified version (4 messages) of the Kerberos protocol. This is also known as the shared key Denning-Sacco protocol [33], with Lowe's modification [60], and is a variant of NS where nonces are replaced by timestamps. We model timestamps as two constants $\mathtt{t}_1$ and $\mathtt{t}_2$, where $\mathtt{t}_1$ is used by honest agents in phase 1 and $\mathtt{t}_2$ in phase 2. In other words, we use the safe approximation that all old timestamps are equated, all current timestamps are equated, but we do draw a distinction between old and current timestamps. We also add clauses $\mathtt{att}_i(\mathtt{t}_j)$ for all $i, j \in \{1, 2\}$, meaning that all timestamps are known to the intruder at all times.

The X.509 row is the so-called "BAN modified version of CCITT X.509 (3)", as referenced in the SPORE database [78]. Several other versions of the X.509 protocol are vulnerable. This particular version is a 3-message protocol that uses nonces and asymmetric cryptography, and no timestamp.

The EAP-AKA row is the extensible authentication protocol (EAP), with authentication and key agreement (AKA), from the AVISPA repository [7]. This is developed from the UMTS AKA authentication and key agreement protocol, see Figure 11. This is meant for a server S and a so-called peer P to agree on a session key for encryption $CK = \mathtt{f}_3(SK, N_s)$, and a session key for integrity $IK = \mathtt{f}_4(SK, N_s)$, where SK is a long-term secret between S and P, N_s is a nonce generated by S at step 3, and $\mathtt{f}_3$ and $\mathtt{f}_4$ are key generation functions. We model SK as $\mathtt{k}(\mathtt{sym}, [S, P])$. The protocol also uses request ids and response ids, which we model as constants `request_id` and `respond_id`, a final signal `success`, a network address identifier NAI for P, modeled as $\mathtt{nai}(P)$, another key generation function $\mathtt{f}_5$, two so-called authentication

1. $S \longrightarrow P : \texttt{request\_id}$
2. $P \longrightarrow S : \texttt{respond\_id}, NAI$
3. $S \longrightarrow P : N_s, AT\_AUTN$
   where $AT\_AUTN = (\{Sqn\}_{AK}, \texttt{f}_1(SK, Sqn, Ns))$
   $AK = \texttt{f}_5(SK, Ns)$
4. $P \longrightarrow S : AT\_RES, h(h(NAI, IK, CK), AT\_RES)$
   where $AT\_RES = \texttt{f}_2(SK, Ns)$
5. $S \longrightarrow P : \texttt{success}$

Figure 11: The EAP-AKA Protocol

1. $A \longrightarrow B : \texttt{enc}(K_a, P_{ab})$
2. $B \longrightarrow A : \texttt{enc}(\{R\}_{K_a}, P_{ab})$
3. $A \longrightarrow B : \{N_a\}_R$
4. $B \longrightarrow A : \{N_a, N_b\}_R$
5. $A \longrightarrow B : \{N_b\}_R$

Figure 12: The EKE protocol

functions $\texttt{f}_1$ and $\texttt{f}_2$, a hash function $\texttt{h}$, and a sequence number Sqn, which we model as $\texttt{session}_i(NAI, S, P)$ in phase i, thus merging all old sequence numbers, and all current sequence numbers, keeping old and current sequence numbers distinct. We test whether P's and S's version of the two keys CK and IK are secret. Secrecy is not guaranteed for P's keys in this model, where several current sessions may have the same sequence number. However, the keys of S are definitely secret. This is what our models for EAP-AKA establish.

Finally, the EKE row is an experiment on Bellovin and Merritt's encrypted key exchange protocol [11, Section 2.1], see Figure 12. The new ingredients here are as follows. First, $\texttt{enc}$ and $\texttt{dec}$ denote encryption and decryption through a *cipher*, i.e., we have not only $\texttt{dec}(\texttt{enc}(M, K), K) = M$ but also $\texttt{enc}(\texttt{dec}(M, K), K) = M$; the latter means that every message can be thought of as the result of the encryption of some message. In particular, the clauses for EKE should be understood modulo an equational theory, generated by the latter two equations. It is however to precompile these equations into the remaining clauses, so that only Horn clauses without equations remain, by computing all superpositions [8] in a preprocessing step. It turns out that for such an equational theory, this terminates. A similar trick is used by Blanchet in his tool ProVerif to compile his rules [14]. In message 1, the public key K_a and its private key K_a^{-1} are generated fresh, and P_{ab} is a shared password between A and B. R is a fresh nonce in message 2, as is N_a in message 3 and N_b in message 4. The final shared key, which should be secret, is R. We naturally assume that all passwords used in phase 1 are known to the attacker in phase 2.

Although this protocol may seem short, this is the one that requires the most

clauses: 124, compared to 46 for X.509, 49 for EAP-AKA or Yahalom, 50 for NS or NSL7, 51 for Kerberos, and 55 for amended NS. The reason is another peculiarity of this protocol: we need to model the fact that P_{ab} is a *weak secret*, i.e., one whose values we can feasibly enumerate. Modeling resistance against dictionary attacks, as done by Corin *et al.* [28], is out of reach of our simple style of models. Instead, we model the weaker, but in fact adequate enough, property of *resistance against time-memory trade-off attacks*. The latter [46] are the most effective form of dictionary attacks. We model the resistance of a weak secret P to these attacks by checking that there are no two messages M_1, M_2 that are *testable* and such that $M_2 = \mathtt{enc}(M_1, P)$. A message is testable if and only if, intuitively, some part of it (but not necessarily all of it) is knowable by the intruder. The idea is that time-memory trade-offs will enumerate all (known) messages M_1 and test whether encryption with P yields any recognizable pattern; or enumerate all known M_2 and test whether decryption with P yields any recognizable pattern. Intuitively, the difference with general resistance against dictionary attacks [28] is that we only allow tests on P of the form $\mathcal{C}[\mathtt{enc}(M, P)]$ or $\mathcal{C}[\mathtt{dec}(M, P)]$ for some public context $\mathcal{C}$ and some public term M; in particular, P only occurs once in these tests.

We model testability through the $\mathtt{testable}_i$ predicate (in phase i) defined in Figure 13. Note that any known message is testable, that a pair $\mathtt{cons}(M_1, M_2)$ is testable if and only if one of M_1, M_2 is (such a pair is known if and only if *both* M_1 and M_2 are). All other clauses model testability using encryption and decryption.

While this is probably the seemingly most complex problem of our set, it is in fact one of the easiest to solve: see Figure 10.

Note that while `h1` returns exact answers in a matter of seconds, except for NSL7 on which it thinks there may be some attacks, Paradox finds models but usually takes much more time. An extreme example is Yahalom, where the 6 element model is found in 53 min, or X.509, with 56 min, or amended NS, where we ran out of patience after 8 hours $1/4$ (the only thing we know is that the least model contains at least 5 elements here).

From an explicitly presented finite model $\mathcal{M}$, as returned by Paradox, it is in easy to extract a formally checkable proof. In Coq, we declare an inductive type of values of $\mathcal{M}$, e.g., $\mathbf{Inductive}\ M : \mathbf{Set} := \mathtt{v1} : M \mid \mathtt{v2} : M \mid \mathtt{v3} : M \mid \mathtt{v4} : M$ for a 4 element model. Then, define all function and predicate symbols by their semantics. E.g., `crypt` (Figure 8) would be described by:

$\mathbf{Definition}\ \mathtt{crypt}(m : M)(k : M) : M :=$
$\mathbf{match}\ m, k\ \mathbf{with}$
$\mathtt{v1}, \mathtt{v1} \Rightarrow \mathtt{v1} \mid \mathtt{v1}, \mathtt{v2} \Rightarrow \mathtt{v1} \mid \mathtt{v1}, \mathtt{v3} \Rightarrow \mathtt{v4} \mid \mathtt{v1}, \mathtt{v4} \Rightarrow \mathtt{v1}$
$\mid \mathtt{v2}, \mathtt{v1} \Rightarrow \mathtt{v2} \mid \mathtt{v2}, \mathtt{v2} \Rightarrow \mathtt{v1} \mid \mathtt{v2}, \mathtt{v3} \Rightarrow \mathtt{v4} \mid \mathtt{v2}, \mathtt{v4} \Rightarrow \mathtt{v4}$
$\mid \mathtt{v3}, \mathtt{v1} \Rightarrow \mathtt{v3} \mid \mathtt{v3}, \mathtt{v2} \Rightarrow \mathtt{v4} \mid \mathtt{v3}, \mathtt{v3} \Rightarrow \mathtt{v4} \mid \mathtt{v3}, \mathtt{v4} \Rightarrow \mathtt{v4}$
$\mid \mathtt{v4}, \mathtt{v1} \Rightarrow \mathtt{v3} \mid \mathtt{v4}, \mathtt{v2} \Rightarrow \mathtt{v2} \mid \mathtt{v4}, \mathtt{v3} \Rightarrow \mathtt{v2} \mid \mathtt{v4}, \mathtt{v4} \Rightarrow \mathtt{v2}$
`end`.

and $\mathtt{att}_2$ would be described by:

$\mathbf{Definition}\ \mathtt{att}_2(m : M) : \mathbf{Prop} :=$
$\mathbf{match}\ m\ \mathbf{with}\ \mathtt{v1} \Rightarrow \mathtt{True} \mid \mathtt{v2} \Rightarrow \mathtt{True} \mid \_ \Rightarrow \mathtt{False}\ \mathtt{end}.$

$$
\begin{array}{rcl}
\texttt{testable}_i(M) & \Leftarrow & \texttt{att}_i(M) \\
\texttt{testable}_i(\texttt{cons}(M_1, M_2)) & \Leftarrow & \texttt{testable}_i(M_1) \\
\texttt{testable}_i(\texttt{cons}(M_1, M_2)) & \Leftarrow & \texttt{testable}_i(M_2) \\
\texttt{testable}_i(\texttt{suc}(M)) & \Leftarrow & \texttt{testable}_i(M) \\
\texttt{testable}_i(M) & \Leftarrow & \texttt{testable}_i(\texttt{suc}(M)) \\
 & & \textit{Testing by decrypting with a known key:} \\
\texttt{testable}_i(\texttt{crypt}(M, \texttt{k}(\texttt{pub}, K))) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{k}(\texttt{prv}, K)) \\
\texttt{testable}_i(\texttt{crypt}(M, \texttt{k}(\texttt{prv}, K))) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{k}(\texttt{pub}, K)) \\
\texttt{testable}_i(\texttt{crypt}(M, \texttt{k}(\texttt{sym}, K))) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{k}(\texttt{sym}, K)) \\
 & & \textit{Testing by encrypting with a known key:} \\
\texttt{testable}_i(M) & \Leftarrow & \texttt{testable}_i(\texttt{crypt}(M, K)), \texttt{att}_i(K) \\
 & & \textit{Testing key by decrypting known message:} \\
\texttt{testable}_i(\texttt{k}(\texttt{prv}, K)) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{crypt}(M, \texttt{k}(\texttt{pub}, K))) \\
\texttt{testable}_i(\texttt{k}(\texttt{pub}, K)) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{crypt}(M, \texttt{k}(\texttt{prv}, K))) \\
\texttt{testable}_i(\texttt{k}(\texttt{sym}, K)) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{crypt}(M, \texttt{k}(\texttt{sym}, K))) \\
 & & \textit{Testing key by encrypting known message:} \\
\texttt{testable}_i(X) & \Leftarrow & \texttt{testable}_i(\texttt{crypt}(M, X)), \texttt{att}_i(M) \\
 & & \textit{Testing by cipher decryption:} \\
\texttt{testable}_i(\texttt{enc}(M, K)) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(K) \\
\texttt{testable}_i(M) & \Leftarrow & \texttt{testable}_i(\texttt{dec}(M, K)), \texttt{att}_i(K) \\
 & & \textit{Testing by cipher encryption:} \\
\texttt{testable}_i(M) & \Leftarrow & \texttt{testable}_i(\texttt{enc}(M, K)), \texttt{att}_i(K) \\
\texttt{testable}_i(\texttt{dec}(M, K)) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(K) \\
 & & \textit{Testing key by cipher-encrypting known message:} \\
\texttt{testable}_i(X) & \Leftarrow & \texttt{testable}_i(\texttt{enc}(M, X)), \texttt{att}_i(M) \\
\texttt{testable}_i(X) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{dec}(M, X)) \\
 & & \textit{Testing key by cipher-decrypting known message:} \\
\texttt{testable}_i(X) & \Leftarrow & \texttt{testable}_i(\texttt{dec}(M, X)), \texttt{att}_i(M) \\
\texttt{testable}_i(X) & \Leftarrow & \texttt{testable}_i(M), \texttt{att}_i(\texttt{enc}(M, X))
\end{array}
$$

Figure 13: Testability

Protocol	Checking Time	#lines
NS	3.29s	1 038
NSL7	1.76s	4 415
Yahalom	36.6s	14 646
Kerberos	2.57s	2 584
X.509	11.01s	35 472
EAP-AKA	4.42s	3 763
EKE	1.99s	5 023

Figure 14: Checking Explicit Models with Coq

The *size* of such a description is proportional to what we called the number of entries above. Proofs of clauses C from the clause set S are then very short: if C contains k free variables, we write its proof in Coq's tactic language as:

$$\mathbf{intro}\ x_1; \mathbf{case}\ x_1; \ldots \mathbf{intro}\ x_k; \mathbf{case}\ x_k; \mathbf{simpl}; \mathbf{tauto}.$$

The effect of this command line is to enumerate all n^k assignments of values to variables. This not only takes time proportional to the number of checks (the #checks columns in Figure 10), but also produces a proof term of size proportional to it.

We conclude that the explicitly presented models approach is practical, however only for small models. While this approach is applicable for the 3 to 6 element models that Paradox found in Figure 10, it is completely unrealistic for the models found by `h1`, whether representable explicitly (NS, amended NS) or not (Yahalom). Note that the MACE algorithm underlying Paradox is doubly exponential in the number n of elements of the model. In practice, the largest models we have discovered with Paradox contained 7 elements. However, when this works, this works well, despite Lemma 5.1.

6 Large Models, and Tree Automata

There are several reasons why we would like to find a more efficient method for producing formally checkable proofs. This will be solved in Section 7. As it stands, the strategy of Section 5 does not scale up. That is, it does not apply to security proofs that would require finite models larger than 6 elements. And there are a few reasons why we would like some larger finite models.

The first one is that Dolev-Yao *secrecy* properties are in fact simple to prove. Remember that the 4 element model that Paradox found for S_{NS}^{safe} mapped each intruder identity to the same value, !1. No such model can ever be used to prove authentication properties, where we need to make a distinction between identities. This phenomenon is already illustrated on Paulson's corrected version of the Yahalom protocol [72], whose security depends on checking the identity of an agent included in a message.

A second reason is that the *style* of protocol specification that we used in Section 3 makes it more likely that secure protocols have small models, but we may need other styles in other applications. One may describe our style as *stateless*: agents only

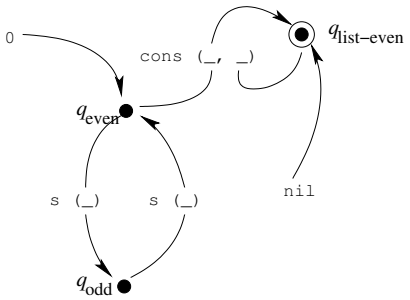

Figure 15: A tree automaton for lists of even numbers

remember past values, not because we have modeled a local state containing all values of their internal variables, but because they are given back to them in received messages. For example, look at message 2 of Figure 2: Alice receives $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$ from the trusted third party. The corresponding clause is (13) (see Figure 3), where Alice expects a message of the form $\{[\mathtt{na}_i(A, B), B, K_{ab}, M]\}_{\mathtt{k(sym},[A,\mathtt{s}])}$. While freshness is checked by verifying that the nonce part N_a is of the form $\mathtt{na}_i(\ldots)$, Blanchet's clever trick of parametrizing $\mathtt{na}_i$ by some free parameters forces this term to match only if A was indeed the intended recipient (viz. the occurrence of A in the key $\mathtt{k}(\mathtt{sym}, [A, \mathtt{s}])$), and to remember whom A wanted to talk to (viz. the two occurrences of B must match). Other, more precise, protocol verification tools employ *stateful* models, whereby each agent maintains a state vector consisting of its local program counter, and all values of its variables (see [17] for an early example). This is needed in verifying protocols where sessions must be sequential, e.g., for the Otway-Rees protocol [69], which is secure if sessions are sequential, but insecure if sessions can be run in parallel [22]. We have played with such a model, and found it satisfiable both with `h1` (with a 54 element model, in 1.1 s) and with Paradox (with a 4 element model, in 227 s). However, the fact that state vectors have high arity (up to 9) entails that, while function tables only require 143 entries—for the 4 element model—, *predicate* entries require 706 716.

We can only expect to need even larger models when considering composition of protocols, or Web services [13], or cryptographic APIs [30], in order of increased complexity. However, note that the number of elements in the model is fairly independent of the *size* of the protocol.

Our model-checking technique will be able to check the larger models found by `h1` (see Figure 10). Some of it rests on intuitions on how we decide $\mathcal{H}_1$ by resolution [42], and the relationship to tree automata.

Tree automaton are best explained as graph-like structures, more precisely as certain hypergraphs. Figure 15 displays one tree automata, which we take as example. We take 0 to denote zero, $s(t)$ to denote the successor of t in $\mathbb{N}$, i.e., $t + 1$ (so that the number n is encoded as $s(\ldots(s(0)))$, with n copies of s), `nil` to denote the empty list, and `cons` to be the binary list constructor (so that the list $[1, 2, 4]$ is encoded as $\mathtt{cons}(1, \mathtt{cons}(2, \mathtt{cons}(4, \mathtt{nil})))$). The states of this tree automaton are q_{even}, q_{odd}, and

$q_{\text{list-even}}$. The transitions are hyperedges from n states $q_1, \ldots, q_n$ to another state q, labeled by an n-ary function symbol f. Graphically, we represent this as an arrow going from $f(\_, \ldots, \_)$ to q, and lines from each state q_i to the corresponding underscore $\_$ in the label. The idea is that if t_1 is a ground term recognized at q_1, and ..., and t_n is a ground term recognized at q_n, then $f(t_1, \ldots, t_n)$ should be recognized at q. For example, in the tree automaton above, 0 is recognized at q_{even} (this is the case $n = 0$, where there are in fact no source state), so $s(0)$ is recognized at q_{odd}, $s(s(1))$ is recognized at q_{even}. We usually define the set of terms *recognized at* a state q as those obtained by finitely many applications of such transitions. We let the reader check that, in the example above, q_{even} recognizes the even natural numbers, q_{odd} the odd natural numbers, and $q_{\text{list-even}}$ the lists of even numbers.

The Horn clause format allows one to express the semantics of tree automata directly. Turn each state q into a unary predicate symbol, and read $q(t)$ as "t is recognized at q". Then the semantics of each transition is expressed as a Horn clause. In the example above, write the following:

$$q_{\text{even}}(O) \qquad q_{\text{even}}(s(X)) \Leftarrow q_{\text{odd}}(X) \qquad q_{\text{odd}}(s(X)) \Leftarrow q_{\text{even}}(X)$$
$$q_{\text{list-even}}(\texttt{cons}(X, Y)) \Leftarrow q_{\text{even}}(X), q_{\text{list-even}}(Y) \qquad q_{\text{list-even}}(\texttt{nil})$$

Then observe that this does not just give the semantics of the tree automaton, but in fact completely describes it. Accordingly, define a *tree automaton* as a finite set of *tree automaton clauses*, defined as being of the form $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$, where $X_1, \ldots, X_n$ are pairwise distinct; such clauses are just tree automaton transitions from $P_1, \ldots, P_n$ to P.

One can generalize the notion of acceptance at a state to any satisfiable set of Horn clauses: for each satisfiable set S of Horn clauses, and each predicate symbol P (i.e., each state P), let $L_P(S)$ be the set of ground terms t such that $P(t)$ is in the least Herbrand model of S. $L_P(S)$ is the *language* recognized at *state* P. When S is a tree automaton, this coincides with the usual definition of the set of terms recognized at P.

This connection between tree automata and Horn clauses was pioneered by Frühwirth *et al.* [38]; there, $L_P(S)$ is called the *success set* for P. This connection was then used in a number of papers: see the comprehensive textbook [25], in particular Section 7.6 on tree automata as sets of Horn clauses.

Tree automata clauses can be generalized right away to alternating tree automata [25, Chapter 7]. Call *ϵ-block* any finite set of atoms of the form $P_1(X), \ldots, P_m(X)$ (with the same X, and $m \geq 0$); it is *non-empty* iff $m \geq 1$. We abbreviate ϵ-blocks as $B(X)$ to make the variable X explicit. We shall also write B for the set $\{P_1, \ldots, P_m\}$. *Alternating automaton clauses* are of the form:

$$P(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad B_1(X_1), \ldots, B_k(X_k) \tag{25}$$

where $B_1(X_1)$, ..., $B_k(X_k)$ are ϵ-blocks, and X_1, ..., X_k are pairwise distinct. It is harder to find a graphical rendition of such clauses. One can think of them as giving the additional power to compute *intersections* $\bigcap_{P \in B_i} L_P(S)$ of recognizable languages: for a term $f(t_1, \ldots, t_k)$ to be recognized at state P, one must find a clause (25) such that t_1 is recognized at *all* the states in B_1, and ..., and t_k is recognized at all the states in B_k.

For technical reasons, we shall also consider *universal clauses*, of the form $P(X)$. These are meant to state that every term is recognized at state P.

We define *alternating tree automata* as any finite set S of alternating automaton clauses and universal clauses. (The standard definition [25] does not include universal clauses; on a fixed first-order signature Σ, a universal clause $P(X)$ may be replaced by the clauses $P(f(X_1, \ldots, X_k)) \Leftarrow P(X_1), \ldots, P(X_k)$, where f ranges over Σ.) Tree automata are the special case without universal clauses, and where ϵ-blocks contain at most one atom.

Given any clause set S, `h1` first applies a canonical abstraction [42, Proposition 3] to get a clause set S' in the decidable class $\mathcal{H}_1$ [68, 84], and such that S is satisfiable whenever S' is. Then `h1` saturates S' by ordered resolution with selection [42], getting a saturated set S_∞. The point is that the subset $S_{prod} \subseteq S_\infty$ of productive clauses that `h1` returns is always an alternating tree automaton [42, Proposition 9]. Determinizing S_{prod} can be done by a standard powerset construction, and we have implemented this in the tool `pldet`, also a part of the `h1` tool suite [40]. The states of the determinized automaton $Det(S_{prod})$ are sets of states of S_{prod}, i.e., sets of predicate symbols.

We shall assume that the following procedure is used to define $Det(S_{prod})$, which builds new states on demand. Initially, the set Q of states, and the set of transitions of $Det(S_{prod})$, are empty. Then, while there is a function symbol f, say of arity k, and k states $q_1, \ldots, q_k$ already constructed in Q such that (†) $q = \{P \mid (\exists P(X) \in S_{prod})$ or $\exists (P(f(X_1, \ldots, X_k)) \Leftarrow B_1(X_1), \ldots, B_k(X_k)) \in S_{prod} \cdot \forall i \cdot B_i \subseteq q_i\}$ is non-empty, add q to Q, and add the transition $q(f(X_1, \ldots, X_k)) \Leftarrow q_1(X_1), \ldots, q_k(X_k)$ to $Det(S_{prod})$. Call this *the powerset construction*. It is well-known that the powerset construction has the property that the language $L_q(Det(S_{prod}))$ of the state $q = \{P_1, \ldots, P_n\}$ in $Det(S_{prod})$ is exactly the intersection $\bigcap_{P \in q} L_P(S_{prod}) \setminus \bigcup_{P \notin q} L_P(S_{prod})$. The fact that states q are built on demand also implies that $L_q(Det(S_{prod})) \neq \emptyset$ for all q.

The connection with finite models was done by Kozen [54], who observed that complete deterministic tree automata were just finite models. (In fact, Kozen *defined* them this way.) There is a transition from the tuple of states $(q_1, \ldots, q_m)$ to q labeled f, i.e., a clause $q(f(X_1, \ldots, X_m)) \Leftarrow q_1(X_1), \ldots, q_m(X_m)$ in the clausal representation of the automaton, if and only if the semantics of f maps the tuple of *values* $(q_1, \ldots, q_m)$ to q. That is, the states of a complete deterministic automaton are the values of the corresponding finite model.

The example tree automaton of Figure 15 is deterministic, but not complete. One gets an equivalent complete deterministic automaton by adding a new, catch-all state $-$, and adding all missing transitions to $-$. This results in a rather messy drawing. However, we can describe it as a finite model as indicated above:

	s	
$0 : q_{\text{even}}$	q_{even}	q_{odd}
	q_{odd}	q_{even}
$\texttt{nil} : q_{\text{list-even}}$	$q_{\text{list-even}}$	$-$
	$-$	$-$

`cons`	q_{even}	q_{odd}	$q_{\text{list-even}}$	$-$
q_{even}	$-$	$-$	$q_{\text{list-even}}$	$-$
q_{odd}	$-$	$-$	$-$	$-$
$q_{\text{list-even}}$	$-$	$-$	$-$	$-$
$-$	$-$	$-$	$-$	$-$

The powerset construction is easier to understand in this light. For every f satisfying (†) above, instead of adding the transition $q(f(X_1, \ldots, X_k)) \Leftarrow q_1(X_1), \ldots, q_k(X_k)$

to $Det(S_{prod})$, add the *table entry* $f(q_1, \ldots, q_k) = q$ to the model. This requires one to write q into the $(q_1, \ldots, q_k)$ entry of table f, possibly after extending all tables with entries for the value q, in case q is fresh. Additionally, tables for predicates are given as truth-tables; for each predicate P, this is defined in $Det(S_{prod})$ so that P holds of state q if and only if $P \in q$.

We can now explain how we estimated the size of models returned by `h1` in Figure 10: we ran `pldet`, which builds $Det(S_{prod})$, and we counted states (values) and transitions (table entries).

Finally, while our model-checking technique will work on alternating tree automata, it will in particular work on finite models encoded as alternating tree automata (which will necessarily be deterministic); i.e., each entry in a table, stating that f applied to values $(v_1, \ldots, v_m)$ should yield value v, will give rise to a tree automaton clause $\texttt{is\_}v(f(X_1, \ldots, X_m)) \Leftarrow \texttt{is\_}v_1(X_1), \ldots, \texttt{is\_}v_m(X_m)$, where there is one $\texttt{is\_}v$ predicate for each value v; the truth-table of each predicate P is encoded as the collection of clauses $P(X) \Leftarrow \texttt{is\_}v(X)$, where v ranges over the values that satisfy P in the model. While this won't decrease the size of the description of the model in Coq—still proportional to #entries—, our model-checker will have the opportunity to find proofs that are shorter than the #checks steps needed in enumeration proofs. E.g., our model-checker will produce the obvious proof that $P(X) \Leftarrow P(X)$ holds (in any model), without enumerating all possible values for X.

Finally, we loop the loop and observe that model-checking against $Det(S_{prod})$ or against our old friend $\mathrm{lfp}\, T_{S_{prod}}$ are the same thing:

Lemma 6.1 *Let S_{prod} be an alternating tree automaton. For any set S of first-order clauses, $Det(S_{prod}) \models S$ if and only if $\mathrm{lfp}\, T_{S_{prod}} \models S$.*

Proof. Say that a value v in a model $\mathcal{M}$ is *definable* iff v is the denotation of some ground term. A model is *fully complete* if and only if all its values are definable. Clearly, $\mathrm{lfp}\, T_{S_{prod}}$ is fully complete, as every value is its own denotation. $Det(S_{prod})$ is also fully complete, since every value (state) q in $Det(S_{prod})$ is the denotation of any ground term in $L_q(Det(S_{prod}))$, and this is non-empty by construction.

For any ground term t, observe that $Det(S_{prod}) \models P(t)$ if and only if t is in $\bigcup_{q/P \in q} L_q(Det(S_{prod})) = \bigcup_{q/P \in q} \left(\bigcap_{P'/P' \in q} L_{P'}(S_{prod}) \setminus \bigcup_{P'/P' \notin q} L_{P'}(S_{prod}) \right) = L_P(S_{prod})$, where the latter equality is by standard set reasoning. That is, $Det(S_{prod}) \models P(t)$ if and only if $\mathrm{lfp}\, T_{S_{prod}} \models P(t)$. It follows that $Det(S_{prod}) \models F$ if and only if $\mathrm{lfp}\, T_{S_{prod}} \models F$ for every universal closed formula F: this is by structural induction on F, using the easy fact that, whenever $\mathcal{M}$ is fully complete, $\mathcal{M} \models \forall X \cdot G(X)$ if and only if $\mathcal{M} \models G(t)$ for every ground term t. Since every set S of first-order clauses is a universal sentence (taking into the implicit universal quantifications over free variables), we conclude. □

7 Model-Checking Against Alternating Tree Automata

Since $Det(S_{prod})$ can have exponential size in the size of S_{prod}, one may say that alternating tree automata are *compact representations* of possibly large finite models.

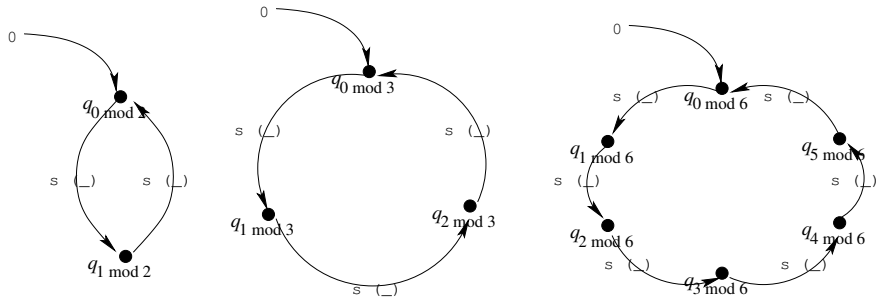

Figure 16: A tree automaton for numbers modulo 2, 3, and 6

We describe how to model-check S against $\mathcal{M} = Det(S_{prod})$ efficiently in practice. But compactness has its toll:

Proposition 7.1 *Checking whether $\mathcal{M} \models S$, where $\mathcal{M} = Det(S_{prod})$ is compactly represented by an alternating tree automaton S_{prod}, and S is a set of Horn clauses, is* **EXPTIME***-complete. It is* **EXPTIME***-hard already if S_{prod} is a (non-alternating) automaton, and S only contains one positive, unit clause.*

Proof. Let n be the number of predicates in S_{prod}, S, k be the largest number of variables in a clause C of S, α the largest symbol arity. Note that we don't *require* to compute $Det(S_{prod})$. However, computing it yields the desired upper bound: $Det(S_{prod})$ can be computed in time exponential in the size of S_{prod}, producing a model with at most 2^n states, and tables with at most $2^{n\alpha}$ entries. We then enumerate up to $(2^n)^k = 2^{nk}$ tuples ρ of values for variables. For each, we can check whether C holds under ρ in exponential time on a Turing machine (we need exponential time to travel along exponential-sized tables stored on the tapes).

Conversely, non-deterministic tree automaton universality is **EXPTIME**-complete [25, Section 1.7, Theorem 14]. This is the problem of checking whether, given a (non-alternating) tree automaton S_{prod} and a state P, $L_P(S_{prod})$ is the set of all ground terms. This is the same as checking $\mathrm{lfp}\, T_{S_{prod}} \models S$, where S only contains the clause $P(X)$, hence to $Det(S_{prod}) \models S$ by Lemma 6.1. □

7.1 Model-Checking Against Automata, Step by Step

We first explain the idea of our model-checking algorithm on an example. We use the tree automaton of Figure 16 as model $\mathcal{M}$. Note that this is no longer a deterministic tree automaton, since 0 is recognized at three distinct states. The names of states should make the automaton self-explanatory; e.g., $q_{2 \bmod 6}$ recognizes exactly the numbers that are equal to 2 modulo 6.

Imagine we would like to check that $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{0 \bmod 2}(s(Z)), q_{1 \bmod 3}(s(s(Z)))]$, where Z is implicitly universally quantified in $q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{0 \bmod 2}(s(Z)), q_{1 \bmod 3}(s(s(Z)))$. Intuitively, this states that if $Z{+}1$ is even ($= 0 \bmod 2$) and $Z + 2 = 1 \bmod 3$, then $Z + 3 = 2 \bmod 6$.

We may first look at all the ways that the model can make $Z+1$ be even. There is only one way to do so in $\mathcal{M}$, i.e., there is only one way that $s(Z)$ be recognized at $q_{0 \bmod 2}$, namely by using the unique transition from $q_{1 \bmod 2}$ to $q_{0 \bmod 2}$; as an automaton clause, this is $q_{0 \bmod 2}(s(X)) \Leftarrow q_{1 \bmod 2}(X)$. Then Z must have been recognized at $q_{1 \bmod 2}$, i.e., Z must be odd. So we are left with checking that $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z), q_{1 \bmod 3}(s(s(Z)))]$. In general, to model-check $\mathcal{M} \models [H \Leftarrow \mathcal{B}, P(f(t_1, \ldots, t_n))]$, where $\mathcal{B}$ is any set of atoms, we shall look for all alternating automaton clauses $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$, with the same P and f, in the alternating tree automaton describing the model $\mathcal{M}$: replacing X_1 by t_1, ..., X_n by t_n, this describes all the ways that $f(t_1, \ldots, t_n)$ can be recognized at P; then it remains to check that $\mathcal{M} \models [H \Leftarrow \mathcal{B}, B_1(t_1), \ldots, B_n(t_n)]$ for all such clauses. This will be formalized in the $(-P, f\ Elim)$ rule below: see Figure 18. The $(-P, f\ Elim)$ works on more complex judgments, for reasons we shall explain shortly. Also, the above discussion assumed that there was no universal clause $P(X)$ in $\mathcal{M}$; otherwise, we shall also use another rule $(-Univ)$ (Figure 17), which simplifies the problem of checking $\mathcal{M} \models [H \Leftarrow \mathcal{B}, P(f(t_1, \ldots, t_n))]$ to $\mathcal{M} \models [H \Leftarrow \mathcal{B}]$: in this case indeed, *every* term is recognized at P.

Returning to our example, we again apply $(-P, f\ Elim)$ to reduce the verification of $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z), q_{1 \bmod 3}(s(s(Z)))]$ to $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z), q_{0 \bmod 3}(s(Z))]$ (if $Z+2 = 1 \bmod 3$, then $Z+1 = 0 \bmod 3$), then to $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$ (... and $Z = 2 \bmod 3$). We have simplified the body of the clause as much as we could in this way.

Now look at the head, $q_{2 \bmod 6}(s(s(s(Z))))$ ("$Z+3 = 2 \bmod 6$"). In a similar way, we realize that $Z+3$ can only be equal to 2 modulo 6 if $Z+2 = 1 \bmod 6$, so we are left with checking $\mathcal{M} \models [q_{1 \bmod 6}(s(s(Z))) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$. In general, and assuming as above that there is no universal clause $P(X)$ in $\mathcal{M}$ (otherwise we shall prefer to use rule $(+Univ)$ of Figure 17), to model-check $\mathcal{M} \models [P(f(t_1, \ldots, t_n)) \Leftarrow \mathcal{B}]$, we shall look for all alternating automaton clauses in $\mathcal{M}$ whose head starts with $P(f(\ldots))$. Let $P_i(f(X_1, \ldots, X_n)) \Leftarrow B_{i1}(X_1), \ldots, B_{in}(X_n)$ be these clauses, $1 \leq i \leq p$. Now $P(f(t_1, \ldots, t_n))$ holds in $\mathcal{M}$ if and only if the disjunction $\bigvee_{i=1}^{p}(B_{i1}(t_1) \wedge \ldots \wedge B_{in}(t_n))$ holds in $\mathcal{M}$. This is the familiar *Clark completion* from logic programming [23]. It then remains to check that $\mathcal{M} \models [\bigvee_{i=1}^{p}(B_{i1}(t_1) \wedge \ldots \wedge B_{in}(t_n)) \Leftarrow \mathcal{B}]$. However, the latter is far from being a clause in general. So, in Figure 18 below, we shall first convert the formula $\bigvee_{i=1}^{p}(B_{i1}(t_1) \wedge \ldots \wedge B_{in}(t_n)) \Leftarrow \mathcal{B}$ into a conjunction of clauses. This will be our rule $(+P, f\ Elim)$.

This is also the rule that forces us to consider not just Horn clauses, but general clauses. Imagine that, in our example, there had been two clauses with head of the form $q_{2 \bmod 6}(s(\ldots))$: the clause $q_{2 \bmod 6}(s(X)) \Leftarrow q_{1 \bmod 6}(X)$ we used above, plus another one, say $q_{2 \bmod 6}(s(X)) \Leftarrow P(X)$. Then using $(+P, f\ Elim)$ would reduce checking $\mathcal{M} \models [q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$ to checking $\mathcal{M} \models [q_{1 \bmod 6}(s(s(Z))) \vee P(s(s(Z))) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$. The latter formula is not Horn, and we shall therefore need to define our model-checking procedures so that it takes general, possibly non-Horn clauses C as input, and checks whether $\mathcal{M} \models C$. (We shall in fact need a bit more again, in the form of histories Γ, see below.)

Let us return to our, unmodified, example. We must check whether it holds that

$\mathcal{M} \models [q_{1 \bmod 6}(s(s(Z))) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$. Using $(+P, f\ Elim)$ twice, we reduce this to the problem of checking $\mathcal{M} \models [q_{5 \bmod 6}(Z) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)]$. Now this clause is something we shall call an ϵ-clause below, i.e., one without a function symbol: on these, we cannot apply either $(-P, f\ Elim)$ or $(+P, f\ Elim)$. However, any ground term that we may plug in for Z must be of the form 0 or $s(t)$ for some ground term t. So we only have to check the two clauses obtained by replacing Z by 0 and by $s(Z_1)$ respectively, namely $\mathcal{M} \models [q_{5 \bmod 6}(0) \Leftarrow q_{1 \bmod 2}(0), q_{2 \bmod 3}(0)]$ and $\mathcal{M} \models [q_{5 \bmod 6}(s(Z_1)) \Leftarrow q_{1 \bmod 2}(s(Z_1)), q_{2 \bmod 3}(s(Z_1))]$. This is what rule $(-P\ Elim)$ does in Figure 18, with a few added twists (in particular, it only applies when there is an atom in the body of the clause to model-check, and uses this as a guide as to which shapes of Z should actually be considered, looking at the model $\mathcal{M}$.) The first clause is easy to check: a single application of $(-P, f\ Elim)$ reduces it to no clause at all (in informal terms, $0 \neq 1 \bmod 2$, so the body of the clause is false, hence the clause itself is vacuously true). Applying $(-P, f\ Elim)$ and $(+P, f\ Elim)$ for as long as we can on the second one eventually leads us to checking the ϵ-clause $\mathcal{M} \models [q_{4 \bmod 6}(Z_1) \Leftarrow q_{0 \bmod 2}(Z_1), q_{1 \bmod 3}(Z_1)]$. Repeating the process, we are led to consider model-checking the following clauses, of which we have kept only the ϵ-clauses:

$$
\begin{array}{lcll}
\mathcal{M} & \models & [q_{5 \bmod 6}(Z) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)] & \\
\mathcal{M} & \models & [q_{4 \bmod 6}(Z_1) \Leftarrow q_{0 \bmod 2}(Z_1), q_{1 \bmod 3}(Z_1)] & \text{(which we have just arrived at)} \\
\mathcal{M} & \models & [q_{3 \bmod 6}(Z_2) \Leftarrow q_{1 \bmod 2}(Z_2), q_{0 \bmod 3}(Z_2)] & \\
\mathcal{M} & \models & [q_{2 \bmod 6}(Z_3) \Leftarrow q_{0 \bmod 2}(Z_3), q_{2 \bmod 3}(Z_3)] & \\
\mathcal{M} & \models & [q_{1 \bmod 6}(Z_4) \Leftarrow q_{1 \bmod 2}(Z_4), q_{1 \bmod 3}(Z_4)] & \\
\mathcal{M} & \models & [q_{0 \bmod 6}(Z_5) \Leftarrow q_{0 \bmod 2}(Z_5), q_{0 \bmod 3}(Z_5)] & \\
\mathcal{M} & \models & [q_{5 \bmod 6}(Z_6) \Leftarrow q_{1 \bmod 2}(Z_6), q_{2 \bmod 3}(Z_6)] & \\
 & \vdots & &
\end{array}
$$

Note that this is looping, as the last ϵ-clause shown is the same as the first one, up to renaming (which is implicit, since all clauses are implicitly universally quantified). When this happens, we stop, and conclude that the last ϵ-clause thus obtained *holds* in $\mathcal{M}$. One may get an intuition of why this must be so as follows. In the sequence of ϵ-clauses above, Z_1 is obtained by assuming that Z denotes a ground term of the form $s(Z_1)$ (see above). Similarly, $Z_1 = s(Z_2)$, $Z_2 = s(Z_3)$, $\ldots$, $Z_5 = s(Z_6)$, so that the term that Z_6 denotes is a proper subterm of that denoted by Z. It follows that, if there were a ground term t for Z that made the first clause, $q_{5 \bmod 6}(Z) \Leftarrow q_{1 \bmod 2}(Z)$, $q_{2 \bmod 3}(Z)$, false in $\mathcal{M}$, then there would be a proper subterm of t for Z_6 that would make the last clause false; i.e., there would be a proper subterm of t for Z that would also make the first clause false. By a classical argument of *descente infinie*, since the subterm ordering is well-founded, this is impossible.

Descente infinie is, at least in classical logic, equivalent to induction. So the Coq proofs we shall produce from this looping argument will be proofs by induction, on the structure of terms.

To formalize this, we keep a *history* Γ of all ϵ-clauses that we have encountered so

$$\frac{\Gamma \vdash C \quad (P \text{ universal})}{\Gamma \vdash C \vee \neg P(t)}\,(-Univ)$$

$$\frac{}{\Gamma, C \vdash C}\,(Loop) \qquad \frac{(P \text{ universal})}{\Gamma \vdash C \vee P(t)}\,(+Univ)$$

Figure 17: Basic model-checking rules

far. Loop-checking is performed by checking whether the current clause is in Γ (see rule $(Loop)$ in Figure 17). Because loop-checking is induction in disguise, one can also see Γ as a collection of induction hypotheses that may be freely applied.

The pair of the clause C to check and the history Γ will be kept in a judgment $\Gamma \vdash C$, and we shall define our model-checking procedure so that $\mathcal{M} \models C$ holds in history Γ if and only if we can derive the judgment $\Gamma \vdash C$ in the system of Figure 17 and Figure 18. In particular, model-checking proceeds by applying rules from the goal, and must therefore be read from conclusions, below, to premises, above.

7.2 The Model-Checking Algorithm, Formalized

The actual definition of our model-checking procedure (Figure 17, Figure 18) is made more concise by relying on a few definitions. Let S_{prod} be an alternating tree automaton. Call a predicate P *universal* in S_{prod} if and only if S_{prod} contains the clause $P(X)$. *Judgments* $\Gamma \vdash C$ are composed of a clause C and a *history* Γ, which is a finite set of ϵ-clauses. An *ϵ-clause*, $E(X)$ is a disjunction of literals of the form $P(X)$ or $\neg P(X)$, with the same variable X; ϵ-blocks are the special case with no negation. All clauses in a judgment are implicitly universally quantified, and do not share variables. Here it is convenient that clauses may be non-Horn, and are written as disjunctions $L_1 \vee L_2 \vee \ldots \vee L_k$. We let S_{prod}/P be the the set of clauses of the form $P(f(X_1, \ldots, X_n)) \Leftarrow \mathcal{B}$ in S_{prod} for some body $\mathcal{B}$ and some function symbol f; $S_{prod}/P, f$ is the set of clauses of the same form, this time with given function f. We write $\vec{t}$ for $t_1, \ldots, t_n$, and $\vec{X}$ similarly in the name of brevity; $[\vec{t}/\vec{X}]$ is the substitution $[t_1/X_1, \ldots, t_n/X_n]$. The notation $E(f(\vec{X}))$, used in rule $(-P\ Elim)$, stands for $E(X)[f(\vec{X})/X]$; this rule is the only one that adds a clause to the history Γ, preparing for an argument by induction. The brace notation used above the premises of this rule means that there are as many premises as there are clauses $P(f(\vec{X})) \vee D$ in S_{prod}/P; similarly for $(-P, f\ Elim)$. In rule $(+P, f\ Elim)$, we enumerate the clauses $P(f(\vec{X})) \Leftarrow \mathcal{B}$ of $S_{prod}/P, f$; $\bigwedge \mathcal{B}$ denotes the conjunction of all atoms in the body $\mathcal{B}$. By cnf, we mean a conjunctive normal form, obtained by distributing ands over ors. The $(Split)$ rule is the only non-deterministic rule, and picks one subclause C_i of $C_1 \vee \ldots \vee C_n$, provided the latter is *block-decomposed*, i.e., C_1, …, C_n are all non-empty and share no free variable. The rules in Figure 18 apply only if no rule from Figure 17 applies. This implies that no universal predicate occurs on the right of $\vdash$.

To produce a Coq proof that $Det(S_{prod}) \models S$, we check that $\vdash C$ for each clause

$$\frac{\overbrace{\Gamma \vdash C \vee D[\vec{t}/\vec{X}]}^{(P(f(\vec{X}))\vee D)\in S_{prod}/P,f}}{\Gamma \vdash C \vee \neg P(f(\vec{t}))}\,(-P, f\ Elim)$$

$$\frac{\overbrace{\Gamma, \forall X \cdot E(X) \vee \neg P(X) \vdash E(f(\vec{X})) \vee D}^{(P(f(\vec{X}))\vee D)\in S_{prod}/P}}{\Gamma \vdash E(X) \vee \neg P(X)}\,(-P\ Elim)$$

$$\frac{\Gamma \vdash C_1 \ \ldots\ \Gamma \vdash C_k}{\Gamma \vdash C \vee P(f(\vec{t}))}\,(+P, f\ Elim)$$
where $\bigwedge_{i=1}^{k} C_i$ is a cnf for
$C \vee \bigvee_{(P(f(\vec{X}))\Leftarrow\mathcal{B})\in S_{prod}/P,f} \bigwedge \mathcal{B}[\vec{t}/\vec{X}]$

$$\frac{\Gamma \vdash C_i \quad (1 \leq i \leq n, n \geq 2)}{\Gamma \vdash C_1 \vee \ldots \vee C_n}\,(Split)$$
where $C_1 \vee \ldots \vee C_n$ is block-decomposed

Figure 18: Model-checking rules, end

C in S. Our tool `h1mc`, also part of the `h1` tool suite [40], looks for a proof ϖ of $\vdash C$ by applying the model-checking rules from the bottom up. The important result here is the following soundness theorem. This is proved by induction on a derivation ϖ of $\vdash C$; apart from this outer induction, the rest of the proof is the skeleton of the Coq proof that `h1mc` extracts from ϖ. Let $\succ$ denote the proper subterm ordering, and observe this is well-founded. Let $\succeq$ be defined by $s \succeq t$ if and only if $s \succ t$ or $s = t$.

Theorem 7.2 (Soundness) *Let $\Gamma = \forall X \cdot E_1(X), \ldots, \forall X \cdot E_m(X)$, and $C = C(X_1, \ldots, X_k)$ be a clause with free variables in $X_1, \ldots, X_k$. If $\Gamma \vdash C$ is derivable using the model-checking rules, then the following formula holds in $\mathrm{lfp}\, T_{S_{prod}}$, where all variables range over ground terms:*

$$\forall X_1, \ldots, X_k \cdot \bigwedge_{\substack{1\leq i\leq k\\ 1\leq j\leq m}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow C(X_1, \ldots, X_k)$$

Proof. By induction over a derivation ϖ of the judgment. We look at the last rule. The cases of $(-Univ)$ and $(+Univ)$ are clear. For $(Loop)$, we observe that C must be of the form $E_j(X_i)$ for some i, j, and we conclude by the antecedent $\forall X \preceq X_i \cdot E_j(X)$.

For $(-P\ Elim)$, let $X_1, \ldots, X_k$ contain at least the variable X free in $E(X) \vee \neg P(X)$. Without loss of generality, let X be X_1. We prove $\forall X_1, \ldots, X_k \cdot \bigwedge_{\substack{1\leq i\leq k\\ 1\leq j\leq m}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow C(X_1, \ldots, X_k) \Rightarrow E(X_1) \vee \neg P(X_1)$ by an auxiliary induction on X_1, well-ordered by $\succ$. (In Coq, we use the **fix** tactic to do this.) Our new induction hypothesis is $(*)\ \forall X \prec X_1 \cdot E(X) \vee \neg P(X)$. We must then show that $E(X_1) \vee \neg P(X_1)$ holds

in $\mathrm{lfp}\, T_{S_{prod}}$. Assume $P(X_1)$ holds: we must show $E(X_1)$. But the only way that $P(t)$ can hold in $\mathrm{lfp}\, T_{S_{prod}}$, for any ground term t, is that t is of the form $f(\vec{t})$, and that there is a clause with head $P(f(\vec{X}))$, say $P(f(\vec{X})) \Rightarrow \mathcal{B}$, in S_{prod}/P, where $\bigwedge \mathcal{B}[\vec{t}/\vec{X}]$ holds in $\mathrm{lfp}\, T_{S_{prod}}$. (In Coq, we use **case** and **inversion**.) We may also write this clause as $P(f(\vec{X})) \vee D$, where D is equivalent to the negation of $\bigwedge \mathcal{B}$. Let $\vec{X}$ be $X_{k+1}, \ldots, X_{k+p}$, and let $E_{m+1}(X)$ be $E(X) \vee \neg P(X)$. By the outer induction on ϖ, we have a proof of $\forall X_2, \ldots, X_k, X_{k+1}, \ldots, X_{k+p} \cdot \bigwedge_{\substack{2 \leq i \leq k+p \\ 1 \leq j \leq m+1}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow E(f(\vec{X})) \vee D$. For $X_1 = f(X_{k+1}, \ldots, X_{k+p})$, we have that every $X \preceq X_{k+i}$ is such that $X \prec X_1$, so we may apply $(*)$. Simple logic then shows that $E(X_2)$ holds. So $\forall X_1, \ldots, X_k \cdot \bigwedge_{\substack{1 \leq i \leq k \\ 1 \leq j \leq m}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow C(X_1, \ldots, X_k) \Rightarrow E(X_1) \vee \neg P(X_1)$ holds in $\mathrm{lfp}\, T_{S_{prod}}$.

Rule $(-P, f\ Elim)$ is justified by the same case analysis, using Coq's **case** and **inversion** tactics, but does not require to introduce any new induction hypothesis into the history. The correctness of $(Split)$ is obvious. Finally, for $(+P, f\ Elim)$, propositional reasoning (using Coq's **tauto** tactic) shows that $\bigwedge_{i=1}^{k} C_k$ implies $C \vee \bigvee_{(P(f(\vec{X})) \Leftarrow \mathcal{B}) \in S_{prod}/P, f} \bigwedge \mathcal{B}[\vec{t}/\vec{X}]$. Using the fact that, for any clause $P(f(\vec{X})) \Leftarrow \mathcal{B}$ in $S_{prod}/P, f$, $\bigwedge \mathcal{B}[\vec{t}/\vec{X}]$ implies $P(f(\vec{t}))$ in $\mathrm{lfp}\, T_{S_{prod}}$, we infer that $C \vee P(f(\vec{t}))$ must also hold in $\mathrm{lfp}\, T_{S_{prod}}$ □

Using Theorem 7.2 and Lemma 6.1, we then obtain:

Corollary 7.3 *If $\vdash C$ is derivable using the model-checking rules for every $C \in S$, then $Det(S_{prod}) \models S$.*

For the sake of efficiency, `h1mc` actually uses a number of extra rules that act as shortcuts in common cases, and which we describe later. As defined in Figure 17 and Figure 18 above, and provided the extra rule $(+Elim)$ of Section 7.4 below is added, these would essentially define Matzinger's procedure [62]. The fact that we do not need the costly rule $(+Elim)$ is already an optimization over Matzinger's procedure, which depends on a subtlety related to the kind of models that `h1` finds: see Section 7.4. However, even this is not enough to make this model-checking algorithm efficient in practice. We shall describe the required optimizations in Sections 7.5 and later.

7.3 Producing Coq Proofs

As we have said above, `h1mc` produces Coq proofs by mimicking the proof of Theorem 7.2, and output corresponding Coq proof arguments. While we have given the bare Coq ingredients in the proof of Theorem 7.2, we illustrate this through the example of Section 7.1. While this is not an example from security, it will be sufficient to explain how `h1mc` generates Coq proofs. Moreover, it will be clear that the resulting Coq proofs are in any case unreadable—the real security argument lies in the model, not in the proof that it is a model.

For basics on Coq, we refer the reader to the Coq'Art [12]. We take some liberties with actual Coq syntax, for readability purposes [83]. While we believe our model-checking algorithm can be made to produce proofs in most standard proof assistants,

the actual details presented in this section definitely rely on Coq's specific ways, and particularly as far as induction proofs are concerned.

First, `h1mc` outputs a definition of all possible ground terms:

$$\textbf{Inductive}\ \texttt{term} : \textbf{Set} := s : \texttt{term} \to \texttt{term} \mid 0 : \texttt{term}$$

and of all clauses in the model, as an inductively defined collection of predicates, taking terms (of type `term` above) and returning formulae (of type **Prop**):

$$\begin{array}{l}\textbf{Inductive}\ q_{0 \bmod 2} : \texttt{term} \to \textbf{Prop} := \\ \quad \texttt{trans\_}q_{0 \bmod 2}\texttt{\_}s_1 : \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{0 \bmod 2}(s(X_1)) \\ \mid \texttt{trans\_}q_{0 \bmod 2}\texttt{\_}0_1 : q_{0 \bmod 2}(0) \\ \textbf{with}\ q_{1 \bmod 2} : \texttt{term} \to \textbf{Prop} := \\ \quad \texttt{trans\_}q_{1 \bmod 2}\texttt{\_}s_1 : \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(X_1) \Rightarrow q_{1 \bmod 2}(s(X_1))\end{array}$$

We omit similar definitions for the other predicates $q_{i \bmod 3}$ ($i \in \{0,1,2\}$) and $q_{i \bmod 6}$ ($0 \leq i \leq 5$). This defines $q_{0 \bmod 2}$ as the least predicate satisfying clauses $\texttt{trans\_}q_{0 \bmod 2}\texttt{\_}s_1$ and $\texttt{trans\_}q_{0 \bmod 2}\texttt{\_}0_1$, simultaneously defining $q_{1 \bmod 2}$ as the least predicate satisfying clause $\texttt{trans\_}q_{1 \bmod 2}\texttt{\_}s_1$. Note that these clauses are Coq incarnations of the corresponding alternating automaton clauses of the model.

Our goal in Section 7.1 was to prove the clause $q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{0 \bmod 2}(s(Z)), q_{1 \bmod 3}(s(s(Z)))$ in this model. Accordingly, `h1mc` will output a proof of the following remark:

$$\textbf{Remark}\ \texttt{rem}_{76} : \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(s(X_1)) \Rightarrow q_{1 \bmod 3}(s(s(X_1))) \Rightarrow q_{2 \bmod 6}(s(s(s(X_1)))).$$

(The funny number "76" results from the numbering scheme that `h1mc` uses, and is not indicative of anything per se. Also, `h1mc` will use the ancillary remark $\texttt{rem}_{76}$ to produce a trivial proof of the actual lemma we are interested in, which only differs by names of variables and order of atoms.)

The remark $\texttt{rem}_{76}$ is proved by using rule $(-P, f\ Elim)$ to examine all the ways that one can derive $q_{0 \bmod 2}(s(X_1))$ in the model. Although one could directly use the **inversion** tactic here, it is more convenient in an automatically derived proof to generate an auxiliary lemma that embodies this instance of inversion. The general form of such a lemma will prove $\forall X_1, \ldots, X_n : \texttt{term} \cdot P(f(X_1, \ldots, X_n)) \Rightarrow \texttt{or}_p(\mathcal{B}_1, \ldots, \mathcal{B}_p)$, where $\mathcal{B}_1, \ldots, \mathcal{B}_p$ are the bodies of the clauses $P(f(X_1, \ldots, X_n)) \Leftarrow \mathcal{B}_i$, $1 \leq i \leq p$, of $S_{prod}/P, f$ in the usual clause notation, and $\texttt{or}_p$ is p-ary disjunction. The latter is defined by `h1mc` as a type with p constructors $\texttt{or}_p\texttt{intro}_i : H_i \Rightarrow \texttt{or}_p(H_1, \ldots, H_i, \ldots, H_p)$, $1 \leq i \leq p$, where $H_1, \ldots, H_p$ are parameter formulae, of type **Prop**. Instead of $\texttt{or}_p(\mathcal{B}_1, \ldots, \mathcal{B}_p)$, it would seem simpler to use the semantically equivalent $\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_p$. However, to do a case analysis on the latter, we would have to use the **elim** tactic $p - 1$ times, whereas the $\texttt{or}_p$ trick allows use to use **elim** just once, and get all cases of the disjunctions in one step.

In our example, `h1mc` produced the following inversion lemma:

$$\textbf{Remark}\ \texttt{rem}_{22} : \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(s(X_1)) \Rightarrow \texttt{or}_1(q_{1 \bmod 2}(X_1)).$$
$$\textbf{Proof}.\ \textbf{intros}.\ \textbf{inversion}\ H.\ \textbf{intros}.\ \textbf{apply}\ \texttt{or}_1\texttt{intro}_1; \textbf{tauto}.\ \textbf{Qed}.$$

The proof of $\texttt{rem}_{76}$ then reads:

$$\textbf{Proof}.\ \textbf{intros}\ X_1.\ \textbf{intro}\ H.\ \textbf{intros}.\ \textbf{elim}\ \texttt{rem}_{22}(X_1, H); \textbf{intros}.\\ \textbf{apply}\ \texttt{rem}_{75}(X_1); \textbf{tauto}.\ \textbf{Qed}.$$

The first **intro** and **intros** tactics introduce $X_1 : \texttt{term}$, the assumption $H : q_{0 \bmod 2}(s(X_1))$, and various other assumptions we don't care about. The goal is now to prove $q_{2 \bmod 6}(s(s(s(X_1))))$. To this end, we apply the **elim** tactic on the inversion lemma $\texttt{rem}_{22}$ applied to X_1 and H (so that $\texttt{rem}_{22}(X_1, H)$ is a proof of $\texttt{or}_1(q_{1 \bmod 2}(X_1))$). In general, if our current proof goal is some formula F, calling **elim** on a proof of $\texttt{or}_p(H_1, \ldots, H_p)$ will subdivide the proof task in p sub-goals. For each i, $1 \leq i \leq p$, the ith subgoal will still be F, only with H_i as added assumption. Here $p = 1$, a seemingly trivial case, where however this mechanism allows us to assert that $q_{1 \bmod 2}(X_1))$ holds, as an extra assumption. To complete the proof of $\texttt{rem}_{76}$, it only remains to prove the corresponding premise of the $(-P, f\ Elim)$ rule, namely $q_{2 \bmod 6}(s(s(s(Z)))) \Leftarrow q_{1 \bmod 2}(Z)$, $q_{0 \bmod 3}(s(Z))$, and to apply it to the case where Z is X_1. The `h1mc` tool completes the proof of the latter clause by a recursive call, producing some other lemma named $\texttt{rem}_{75}$, and uses it as shown above, by invoking the **apply** tactic on $\texttt{rem}_{75}(X_1)$; we then let Coq find the trivial proofs of the assumptions left unproved by doing some elementary propositional reasoning using **tauto**. Accordingly, $\texttt{rem}_{75}$ is declared as follows.

Remark $\texttt{rem}_{75} : \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{1 \bmod 3}(s(s(X_1))) \Rightarrow q_{2 \bmod 6}(s(s(s(X_1))))$.

and is proved in a similar way, using $(-P, f\ Elim)$ and other auxiliary sub-remarks with lower numbers.

After a series of applications of $(-P, f\ Elim)$, `h1mc` will arrive at the following clause, which it will have to prove by using $(+P, f\ Elim)$ instead:

Remark $\texttt{rem}_{72} : \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{2 \bmod 3}(X_1) \Rightarrow q_{2 \bmod 6}(s(s(s(X_1))))$.

Our example is too degenerate to actually show what will happen in this case, and the general case produces hairy proofs. So let's explain the main technical difficulty instead. We use **intro** and **intros** to separate the variables $X_1, \ldots, X_m$ of the clause, and its assumptions $H_1 : A_1, \ldots, H_\ell : A_\ell$, from the head of the clause, here $q_{2 \bmod 6}(s(s(s(X_1))))$. In the general case, this head will be $P(f(t_1, \ldots, t_n)) \vee D$, for some disjunction D of atoms. Look at all the clauses $P(f(X_1, \ldots, X_n)) \Leftarrow \mathcal{B}_j$, $1 \leq j \leq q$, in $S_{prod}/P, f$. Then `h1mc` will, by using $(+P, f\ Elim)$, obtain proofs ϖ_i of the formulae $\forall X_1, \ldots, X_m \cdot A_1 \Rightarrow \ldots \Rightarrow A_\ell \Rightarrow C_i \vee D$, for each i, $1 \leq i \leq k$, where $\bigwedge_{i=1}^{k} C_i$ is a cnf for $\bigvee_{j=1}^{q} \bigwedge \mathcal{B}_j$. So $\varpi_i(X_1, \ldots, X_m, H_1, \ldots, H_\ell)$ will be a proof of $C_i \vee D$ for each i. Now `h1mc` produces a proof ϖ_{Distr} of $C_1 \wedge \ldots \wedge C_k \Rightarrow \mathcal{B}_1 \vee \ldots \vee \mathcal{B}_q$, and uses it to derive a proof of $\mathcal{B}_1 \vee \ldots \vee B_q \vee D$ (under assumptions $X_1 : \texttt{term}, \ldots, X_m : \texttt{term}, H_1 : A_1, \ldots, H_\ell : A_\ell$). An inversion lemma as used above allows `h1mc` to deduce the desired head $P(f(t_1, \ldots, t_n)) \vee D$ from the latter.

The main difficulty is to generate ϖ_{Distr}. First, we know that $C_1 \wedge \ldots \wedge C_k$ is equivalent to $\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_q$ in classical logic, however Coq is based on *intuitionistic* logic. (While we could import the `Classical` module that implements classical

reasoning in Coq, we do not wish to do so.) It turns out that, since none of these propositional formulae involve negation, these two formulae must also be intuitionistically equivalent—something that is obvious from the Kripke semantics of propositional intuitionistic logic.

The second difficulty is complexity-theoretic. We illustrate it through an example that an early version of `h1mc` actually produced in 2003. This example has $q = 13$, $\mathcal{B}_1$ through $\mathcal{B}_5$ are conjunctions of just 2 atoms, while $\mathcal{B}_6$, ..., $\mathcal{B}_{13}$ each contain just one atom. Distributing ands over ors yields a cnf with $2^5 = 32$ clauses C_1, ..., C_{2^5}, each with 13 atoms. It is tempting to let Coq prove $C_1 \wedge \ldots C_{2^5} \Rightarrow \mathcal{B}_1 \vee \ldots \vee \mathcal{B}_{13}$ by invoking **tauto**, however this is hopeless. This is because **tauto**, just like any other reasonable tableaux prover for propositional formulae, will attempt to use the invertible rules of its calculus eagerly. Concretely, this means that **tauto** will do a case analysis over the 13 atoms of C_1; then a case analysis on the 13 atoms of C_2, and similarly on C_3, ..., C_{2^5}. Eventually, the resulting 13^{2^5} clauses are trivial to prove. But no prover we know, including **tauto**, is able to deal with that many clauses. In general, the problem is that, while a cnf $C_1 \wedge \ldots \wedge C_k$ for $\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_q$ is of exponential size already in q, checking this by distributing back the ors over the ands, as all tableaux provers we know do, is of complexity *doubly exponential* in q.

To solve this, it would in principle be best to keep a trace of the operations used to obtain $C_1 \wedge \ldots \wedge C_k$ from $\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_q$, and using this trace to guide a Coq proof that would not rely on **tauto** but would use the elementary tactics **elim**, **split**, **left**, and **right** on $\wedge$ and $\vee$, explicitly. We haven't done so in `h1mc`, as the optimizations presented in Section 7.5 and later, plus a few tricks that eliminate tautological clauses and subsumed clauses among C_1, ..., C_k, happen to suffice in practice.

Let us turn to induction. Eventually, `h1mc` needs to prove the ϵ-clause $q_{5 \bmod 6}(Z) \Leftarrow q_{1 \bmod 2}(Z), q_{2 \bmod 3}(Z)$. To this end, `h1mc` produces:

Remark $\mathtt{rem}_{66} : \forall X_1 : \mathtt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{2 \bmod 3}(X_1) \Rightarrow q_{5 \bmod 6}(X_1)$.
Proof. **fix** H_{ind} 1. **intro**X. **case** X.
  intros X_1; **exact** $\mathtt{rem}_{65}(H_{ind}, X_1)$.
  intro$H$. **elim** $\mathtt{rem}_{35}(H)$. **Qed**.

The key here is the **fix** tactic: **fix** H_{ind} 1 simply adds the whole goal to prove as a new assumption $H_{ind} : \forall X_1 : \mathtt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{2 \bmod 3}(X_1) \Rightarrow q_{5 \bmod 6}(X_1)$. This serves as induction hypothesis. We can then apply it to any proper subterm X_2 of X_1 by invoking $H_{ind}(X_2)$. The extra number "1" informs Coq that subterms should be extracted from first quantified term, here X_1.

The **fix** tactic is rarely used in man-made proofs, because it is error-prone: only when typing the final **Qed** will Coq check that all calls to the induction hypothesis H_{ind} were really applied to proper subterms, and are therefore valid. However, using more standard induction tactics such as **induction** would require us to specify in advance the actual subterm of X_1 that we shall apply our induction hypothesis on; **fix** relieves us from the difficulty.

Once this is done, **intro**X and **case** X rip the formula of its initial universal quantification, renames X_1 as X : `term`, and does a case analysis on the shape of X. The second line of the proof, which invokes $\mathtt{rem}_{65}$, deals with the case where X is of the form $s(X_1)$ for some X_1 : `term`, the third line deals with the case where $X = 0$.

A curious thing in the proof of $\texttt{rem}_{66}$ shown above is that, although it introduces the induction hypothesis H_{ind}, it never uses it directly. Instead, it passes it on to the auxiliary sub-remarks that need it. This is why $\texttt{rem}_{65}$ is invoked with both X_1 and H_{ind} as arguments, so that it can use the latter at all. Accordingly, $\texttt{rem}_{65}$ is declared as:

$$\begin{array}{l}\textbf{Remark}\ \texttt{rem}_{65} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow \\ \qquad \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(s(X_1)) \Rightarrow q_{2 \bmod 3}(s(X_1)) \Rightarrow q_{5 \bmod 6}(s(X_1)).\end{array}$$

where the second line, quantified over X_1, is the actual formula we want to prove, and the formula $\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)$ on the first line is the induction hypothesis that $\texttt{rem}_{65}$ can use.

In general, when `h1mc` has managed to derive a sequent of the form $\Gamma \vdash C$, where Γ consists of the ϵ-clauses $C_1, \ldots, C_k$, it will output a Coq proof of $C_1 \Rightarrow \ldots \Rightarrow C_k \Rightarrow C$. More precisely, it will output a proof of $C_1 \Rightarrow \ldots \Rightarrow C_k \Rightarrow C$, where $C_1, \ldots, C_k$ are the *relevant* induction hypotheses from Γ, i.e., the ones that have really been used in an instance of $(Loop)$ in the given derivation of $\Gamma \vdash C$. This way, instead of carrying up to 6 induction hypotheses as at the end of Section 7.1, `h1mc` will only need one for each of the sub-remarks leading to $\texttt{rem}_{66}$:

$$\begin{array}{ll}\textbf{Remark}\ \texttt{rem}_{60} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow & \\ \qquad \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(X_1) \Rightarrow q_{1 \bmod 3}(X_1) \Rightarrow q_{4 \bmod 6}(X_1). & [\ldots] \\ \textbf{Remark}\ \texttt{rem}_{53} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow & \\ \qquad \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{0 \bmod 3}(X_1) \Rightarrow q_{3 \bmod 6}(X_1). & [\ldots] \\ \textbf{Remark}\ \texttt{rem}_{45} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow & \\ \qquad \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(X_1) \Rightarrow q_{2 \bmod 3}(X_1) \Rightarrow q_{2 \bmod 6}(X_1). & [\ldots] \\ \textbf{Remark}\ \texttt{rem}_{36} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow & \\ \qquad \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{1 \bmod 3}(X_1) \Rightarrow q_{1 \bmod 6}(X_1). & [\ldots] \\ \textbf{Remark}\ \texttt{rem}_{26} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow & \\ \qquad \forall X_1 : \texttt{term} \cdot q_{0 \bmod 2}(X_1) \Rightarrow q_{0 \bmod 3}(X_1) \Rightarrow q_{0 \bmod 6}(X_1). & \end{array}$$

As above, $\texttt{rem}_{26}$ requires an inductive argument on X_1. This eventually leads to the following sub-remark $\texttt{rem}_{15}$, obtained by using $(Loop)$, i.e., by invoking the induction hypothesis. (We have slightly edited its proof, which contained some useless steps.)

$$\begin{array}{l}\textbf{Remark}\ \texttt{rem}_{15} : (\forall X : \texttt{term} \cdot q_{1 \bmod 2}(X) \Rightarrow q_{2 \bmod 3}(X) \Rightarrow q_{5 \bmod 6}(X)) \Rightarrow \\ \qquad \forall X_1 : \texttt{term} \cdot q_{1 \bmod 2}(X_1) \Rightarrow q_{2 \bmod 3}(X_1) \Rightarrow q_{5 \bmod 6}(X_1). \\ \textbf{Proof}.\ \textbf{intro}\ H_{ind}.\ \textbf{intros}\ X_1.\ \textbf{exact}\ H_{ind}(X_1).\ \textbf{Defined}.\end{array}$$

We finish with a subtle point. While all our proofs were terminated by **Qed** until now, the proofs of all sub-remarks that require at least one induction hypothesis, among which not only $\texttt{rem}_{15}$, but also $\texttt{rem}_{26}$, $\texttt{rem}_{36}$, ..., $\texttt{rem}_{60}$ and $\texttt{rem}_{65}$ above, are ended with the **Defined** keyword. This is required to make their proofs *transparent*, as needed by Coq to be able to check that all uses of induction hypothesis indeed apply to *proper* subterms, as discussed above. This check involves traversing the proof terms generated by Coq, along all possible paths from the root of the proof to variables such as H_{ind}: to check the proof term we have given for $\texttt{rem}_{66}$, Coq will have to traverse all

remarks used in its definition, including $\texttt{rem}_{65}$, $\texttt{rem}_{60}$, ..., $\texttt{rem}_{36}$, $\texttt{rem}_{26}$, and $\texttt{rem}_{15}$, where the induction hypothesis is finally used.

It is algorithmically practical to produce such *delocalized* induction proofs, where induction hypotheses are introduced in one lemma ($\texttt{rem}_{66}$) but used in another ($\texttt{rem}_{15}$). However, we must admit that such proofs are not the most readable kind.

7.4 Completeness

The model-checking procedure is also complete, in a subtle sense. We now need to quantify over all signatures Σ that contain all the symbols of S_{prod} and S. While $\mathrm{lfp}\, T_{S_{prod}}$ is a set of ground atoms that is independent of the signature Σ, as a model, it is a subset of the set of all ground atoms, which *does* depend on Σ. To make the dependency on Σ explicit, write this model $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$.

The model-checking procedure now only has the following weak completeness property: if $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models C$ for *every* Σ, then there is a derivation of $\vdash C$. It is easy to see that `h1`, as a resolution algorithm, produces a set S_{prod} satisfying this stronger assumption. This is because resolution algorithms do not depend on the chosen signature, only on the clauses that they work on.

The difference between checking $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models C$ for *every* Σ, or checking $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models C$ just for a given $\Sigma = \Sigma_0$ can be illustrated by considering the case $S_{prod} = \{p(a)\}$ and $S = \{p(X)\}$: we certainly have $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models S$ if Σ only contains a, but this fails otherwise. Note that the soundness Theorem 7.2 is in fact true whatever the signature Σ. This being, hopefully, clarified, we obtain:

Proposition 7.4 (Completeness) *If $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models S$ for every signature Σ containing all the symbols of S_{prod} and S, then one may find a derivation of $\vdash C$ for every $C \in S$, in an effective way.*

Proof. We first claim that, if C_1 holds in $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$ for all Σ, then for any history Γ, some rule applies that has $\Gamma \vdash C_1$ as its conclusion. This is obvious if C_1 contains a universal predicate, in which case $(-Univ)$ or $(+Univ)$ applies. Otherwise, the key observation is that the only way that an atom of the form $P(f(\vec{t}))$ can hold in $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$ is that there is a clause $P(f(\vec{X})) \Leftarrow \mathcal{B}$ in $S_{prod}/P, f$ such that $\bigwedge \mathcal{B}[\vec{t}/\vec{x}]$ holds in S_{prod}. In other words, $P(f(\vec{t}))$ is equivalent to $\bigvee_{(P(f(\vec{X})) \Leftarrow \mathcal{B}) \in S_{prod}/P,f} \bigwedge \mathcal{B}[\vec{t}/\vec{X}]$ in $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$. This is Clark completion [23]. This directly justifies using $(+P, f\ Elim)$ in case C_1 contains a positive atom with non-variable argument, i.e., C_1 is of the form $C \vee P(f(\vec{t}))$. In case C_1 can be written $C \vee \neg P(f(\vec{t}))$, then Clark completion and Boolean reasoning show that all the premises $C \vee D[\vec{t}/\vec{X}]$ of rule $(-P, f\ Elim)$ must hold in $\mathrm{lfp}\, T_{S_{prod}}$.

In all other cases, C_1 is of the form $E_1(X_1) \vee \ldots \vee E_k(X_k)$. If $k \geq 2$, we may apply $(Split)$. If $k = 0$, then C_1 would be false, so the case does not happen. Otherwise, if C_1 contains a negative atom with variable argument, i.e., $C_1 = E(X) \vee \neg P(X)$, a variant of Clark completion (above), using the fact that P is not universal, shows that $P(X)$ is equivalent to $\bigvee_{(P(f(\vec{X})) \Leftarrow \mathcal{B}) \in S_{prod}/P} \bigwedge \mathcal{B}[f(\vec{t})/X]$ in $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$, justifying using $(-P\ Elim)$. In the remaining case, C_1 is a disjunction $P_1(X) \vee \ldots \vee P_n(X)$ of positive atoms with variable arguments; however, for Σ large enough, i.e., containing some

constant a not in S_{prod}, we observe that $P_1(a), \ldots, P_n(a)$ are all false in $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$, contradicting that C_1 is true: so this case does not happen.

Second, we observe that applying $(Split)$ and $(Loop)$ eagerly forces proof search to terminate. This rests on the fact that there can only be finitely many ϵ-clauses, hence also finitely many possible histories Γ, in particular. The missing, easy details are left to the reader. □

We have observed that `h1` produces proofs that are independent on Σ, hence satisfy the assumption of Proposition 7.4. Models produced by Paradox only satisfy $\mathrm{lfp}_\Sigma\, T_{S_{prod}} \models S$ for Σ equal to—no larger than–the signature Σ_0 defined by S. To regain completeness under this weaker assumption, we need an additional rule:

$$\frac{\overbrace{\Gamma, \forall X \cdot E(X) \vdash E(f(\vec{X}))}^{f \in \Sigma_0}}{\Gamma \vdash E(X)}\ (+Elim)$$

whenever $E(X)$ is an ϵ-block consisting only of positive atoms $+P(X)$, and there is one premise for each function symbol f in the given signature Σ_0. This is costly: the only rule that can be applied to derive the premise is $(+P, f\ \ Elim)$, which we had better avoid. We have experimented `h1mc` with the $(+Elim)$ rule on (i.e., using its so-called `-exact-sig` option), and found this not to be competitive relative to the simple-minded approach of Section 5 on models found by Paradox, despite extra algorithmic optimizations in `h1mc` in this case. This seems to be due to the fact that tables are dense, and that `h1mc` still has to enumerate them in some way. (E.g., we have witnessed `h1mc` generate 510 premises in one instance of $(-P\ Elim)$.)

On the other hand, the approach of Figure 17 and Figure 18, i.e., without the $(+Elim)$ rule, *is* effective in all cases where we can find a model using `h1`. We believe this is due to the fact that transitions in alternating tree automata found by `h1` are very sparse, so that, in particular, instances of $(-P\ Elim)$ have very few premises in general. The role of optimizations (see below) is crucial, too.

Figure 19 gives an indication of the size of Coq proofs produced by `h1mc` on the models found by `h1`. We have copied back the #elts, #entries and #checks from Figure 10 for easy reference. Times (rightmost column) are reported as $t_1 + t_2$, where t_1 is the time taken by `h1mc`, and t_2 is the time taken by Coq to check the proof. Note that producing and checking a formal Coq proof of the amended NS protocol, even on the 57 element model found by `h1`, is practical, even though there is probably a smaller model—which we didn't find. It is also rather remarkable that while we haven't been able to determinize S_{prod} in the Yahalom case and in the X.509 case, `h1mc` manages to find a proof in a reasonable amount of time.

7.5 Optimization I: Simulation Testing

A very effective shortcut is as follows. Proving $\Gamma \vdash P(X) \Leftarrow Q(X)$, i.e., proving that $L_Q(S_{prod}) \subseteq L_P(S_{prod})$, can be done in many cases by exhibiting a form of simulation relation between automaton states such that Q simulates P.

First, let $NE(S)$ be the smallest set of predicate symbols such that, for every clause of the form (25) in S, if $B_1 \subseteq NE(S)$ and $\ldots$ and $B_k \subseteq NE(S)$, then $P \in NE(S)$.

	$Det(S_{prod})$			Coq proof		
Protocol	#elts	#entries	#checks	size	#lines	time
NS	46	217 312	430 10^6	0.66 Mb	15 560	0.53+10.73s
amended NS	57	188 724	1.245 10^9	1.40 Mb	31 640	1.90+25.67s
Yahalom	≥ 57		$\geq$ 2.46 10^9	3.50 Mb	60 938	7.34+53.77s
Kerberos	57	7 952	84.5 10^6	1.48 Mb	30 326	2.02+23.97s
X.509	≥ 29		$\geq$ 228.5 10^6	0.97 Mb	20 471	0.95+23.33s
EAP-AKA	72	22 550	7.74 10^9	1.90 Mb	32 229	0.88+43.30s
EKE	48	16 016	64.5 10^6	3.20 Mb	73 683	3.18+89.94s

Figure 19: Coq proofs

Clearly, if $L_P(S) \neq \emptyset$, then $P \in NE(S)$. In fact, if S is a non-deterministic automaton, this yields a decision procedure for non-emptiness: if $P \in NE(S)$ then $L_P(S) \neq \emptyset$. This is not so for alternating automata, for which non-emptiness is **EXPTIME**-complete [25, Theorem 55, Section 7.5]. $NE(S)$ can be computed in polynomial time by a marking algorithm.

We say that R is a *simulation* on the states of S_{prod} if and only if for every clause:

$$P(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad B_1(X_1), \ldots, B_k(X_k) \tag{26}$$

with $P \in NE(S_{prod})$, for every state P' with $P\ R\ P'$, there is a clause:

$$P'(f(X_1, \ldots, X_k)) \quad \Leftarrow \quad B'_1(X_1), \ldots, B'_k(X_k) \tag{27}$$

in S_{prod} with $B_i\ R^\sharp\ B'_i$ for every i, $1 \leq i \leq k$—we let $B\ R^\sharp\ B'$ if and only if for every $Q' \in B'$, there is a $Q \in B$ with $Q\ R\ Q'$.

There is always a largest simulation, which is computable in polynomial time, by a largest fixpoint computation on the set of pairs (P, P') of predicates.

The next two results are probably folklore, at least for non-deterministic automata.

Lemma 7.5 *For any two simulations R and R', $(R; R')$, defined by $P\ (R; R)\ P''$ if and only if $P\ R\ P'$ and $P'\ R'\ P''$ for some $P' \in \mathcal{P}$, is a simulation.*

Proof. First, we claim that: $(*)$ if $P\ R\ P'$, where R is a simulation, and $P \in NE(S_{prod})$, then $P' \in NE(S_{prod})$. This is by structural induction on a proof that $P \in NE(S_{prod})$. Since $P \in NE(S_{prod})$ there must be a clause (26) with $B_1 \subseteq NE(S_{prod})$, ..., $B_k \subseteq NE(S_{prod})$. By definition of a simulation, and since $P \in NE(S_{prod})$, there must be a clause (27) such that $B_i\ R^\sharp\ B'_i$ for every i, $1 \leq i \leq k$. For every $Q' \in B'_i$, there is a $Q \in B_i$ such that $Q\ R\ Q'$. By induction hypothesis, since $Q \in B_i \subseteq NE(S_{prod})$, $Q' \in NE(S_{prod})$. So $B'_i \subseteq NE(S_{prod})$ for every i, $1 \leq i \leq k$. Whence $P' \in NE(S_{prod})$.

Let R and R' be as in the Lemma. Let $P\ (R; R')\ P''$, say $P\ R\ P'\ R\ 'P''$. If $P \notin NE(S_{prod})$, then we are done, so assume $P \in NE(S_{prod})$. For every clause (26) in S_{prod} there is a clause (27) in S_{prod} with $B_i\ R^\sharp\ B'_i$ for every i, $1 \leq i \leq k$. By $(*)$, $P' \in NE(S_{prod})$, so there is a clause $P''(f(X_1, \ldots, X_k)) \Leftarrow B''_1(X_1), \ldots, B''_k(X_k)$ in S_{prod} such that $B'_i\ R'^\sharp\ B''_i$ for every i, $1 \leq i \leq k$. It follows that $B_i\ (R; R')^\sharp\ B''_i$ for every i, showing that $(R; R')$ is a simulation. □

Proposition 7.6 *Let R be the largest simulation. Then R is a quasi-ordering. If $E \supseteq E'$ then $E\ R^\sharp\ E'$. If $E\ R^\sharp\ E'$ then $L_E(S_{prod}) \subseteq L_{E'}(S_{prod})$.*

Proof. First, R is reflexive, because the equality relation is a simulation. To show that R is transitive, we realize that $(R; R)$ is a simulation, by Lemma 7.5, so by maximality $(R; R) \subseteq R$: so R is transitive. That $E \supseteq E'$ implies $E\ R^\sharp\ E'$ is by the definition of $R^\sharp$ and the fact that R is reflexive. The last claim is shown by proving that whenever R is a simulation, then for every ground term $t \in L_E(S_{prod})$, whenever $E \preceq^\sharp E'$ then $t \in L_{E'}(S_{prod})$. This is proved by structural induction on $t = f(t_1, \ldots, t_k)$. Let $E' = \{P'_1, \ldots, P'_m\}$. Since $E \preceq^\sharp E'$, for every j, $1 \leq j \leq m$, there is a $P_j \in E$ such that $P_j \preceq P'_j$. Since $t \in L_E(S_{prod})$, $t \in L_{P_j}(S_{prod})$ for every j, so there is a clause:

$$P_j(f(X_1, \ldots, X_k)) \Leftarrow B_{j1}(X_1), \ldots, B_{jk}(X_k)$$

in S_{prod} such that $t_i \in L_{B_{ji}}(S_{prod})$ for every i, $1 \leq i \leq k$. Since $t \in L_{P_j}(S_{prod})$, $L_{P_j}(S_{prod}) \neq \emptyset$, so $P_j \in NE(S_{prod})$, and because $P_j\ R\ P'_j$, by definition there must be a clause:

$$P'_j(f(X_1, \ldots, X_k)) \Leftarrow B'_{j1}(X_1), \ldots, B'_{jk}(X_k)$$

such that $B_{ji}\ R^\sharp\ B'_{ji}$ for every i, $1 \leq i \leq k$. By induction hypothesis, since $t_i \in L_{B_{ji}}(S_{prod})$, $t_i \in L_{B'_{ji}}(S_{prod})$. So, using the clause above, $t \in L_{P'_j}(S_{prod})$. As j is arbitrary between 1 and m, $t \in L_{E'}(S_{prod})$. □

It follows that, if there is a simulation R with $Q\ R\ P$, then $L_Q(S_{prod}) \subseteq L_P(S_{prod})$. This again compiles into a Coq proof using **fix**, **case** and **inversion**.

7.6 Optimization II: Checking the Abstracted Clauses, not the Original Set

Another `h1`-specific optimization is the following. Remember that `h1` first abstracts the initial clause set S into another clause set S' that falls into the class $\mathcal{H}_1$. Instead of model-checking S directly against $Det(S_{prod})$, we model-check S' instead, then produce a Coq proof that S' implies S. Since S' is obtained from S by some reversed form of resolution, showing that S' implies S is particularly easy.

7.7 Optimization III: Memoization

The final important optimization is that `h1mc` *memoizes* proof attempts. That is, when attempting to derive $\Gamma \vdash C$, it first checks whether it has already derived $\Gamma' \vdash C'$ for some $\Gamma' \subseteq \Gamma$ and some clause C' that subsumes C, i.e., such that $C = C'\sigma \vee D$ for some substitution σ and some subclause D. If so, it reuses the proof of $\Gamma' \vdash C'$ to infer $\Gamma \vdash C$ directly.

Our tool `h1mc` also rests on less important optimizations, which we therefore omit. See the appendices of the full version of the paper [43], available from the author's Web page.

8 Equational Theories

More and more protocols in the literature can only be modeled using equational theories, to represent e.g. bitwise exclusive-or (xor) or modular exponentiation [29]. Our tool `h1` really cannot deal with such equational theories, unless the equations can be eliminated, as we have suggested in the case of EKE in Section 5. This trick generalizes Blanchet's rule compilation trick [14].

However, xor and modular exponentiation are two examples of theories that cannot be dealt with in such a way. While `h1` cannot deal with them, this is in principle easy to Paradox: just add the needed equations as unit clauses. For example, Figure 22 lists axioms for modular exponentiation as used in Diffie-Hellman key agreement [35], where exponents obey an Abelian group law $*$; $\mathtt{g}(M)$ is meant to denote g^M for a fixed generator g. (Following an established tradition in automated deduction, we use $\approx$ for the equality symbol, to distinguish it visually from actual equality.)

It is easy to extend the approach of Section 5 to the equational case. Indeed, to model-check the clause set S against the finite model $\mathcal{M}$, modulo the equational theory E, we only need to model-check $S \cup E$, under the interpretation that $\approx$ is equality. One might let a finite model finder find a model for $S \cup E \cup \mathcal{E}q$, where $\mathcal{E}q$ is the theory of equality (see below) to this end, however this is not needed: any model found by a finite model finder such as Paradox will interpret $\approx$ as equality, so we only have to check $S \cup E$.

Generating Coq proofs from an explicit finite model $\mathcal{M}$ of $S \cup E$, where $\approx$ is equality over $\mathcal{M}$, is done as in Section 5. The only difference has to do with equality. Indeed, $\approx$ cannot be interpreted as Coq's default equality. We illustrate this on a small example. Remember the definition $\mathbf{Inductive}\ \mathtt{term} : \mathbf{Set} := s : \mathtt{term} \to \mathtt{term} \mid 0 : \mathtt{term}$ that we used that we used in Section 7.3, and imagine we want to interpret natural numbers (of type `term`) modulo the equation $s(s(X)) = X$; i.e., modulo 2. Then one can prove in Coq that $s(s(X)) \neq X$ for all X, so Coq equality $=$ cannot be used for our equality modulo 2. (Beginners in Coq should be warned not to attempt to use $\mathbf{Axiom}\ eqn_1 : \forall X : \mathtt{term} \cdot s(s(X)) = X$ to this end. This is a gross misinterpretation of what axioms are, and results in an inconsistency.)

In fact, one should define another type of "terms modulo 2". (Admittedly, in this simple example, one could also cheat and observe that this is just the finite type of bits.) The standard way of doing so in Coq is to use a so-called *setoid type*, i.e., a record type whose first field is the carrier type (e.g., `term`), the second one is an equivalence relation over the carrier type, and the remaining field is a proof that this is indeed an equivalence relation. Several proposals to include so-called *quotient types* in type theories have been considered [48, 47]. Whether they are based on defining actual, new quotient types, or on using setoids, their mere definition requires one to produce proofs of reflexivity, symmetry, and transitivity. Similarly, one also has to show that every function symbol f and every predicate symbol is defined on equivalence classes, independently of their representatives. Moreover, since our intended equality is not Coq's built-in equality, we will have to use a distinct predicate `equal` for our equality.

Accordingly, to check the finite model $\mathcal{M}$ found by Paradox, we have to produce Coq proofs of $S \cup \widetilde{E} \cup \mathcal{E}q$, where $\widetilde{E}$ is the set of clauses $\mathtt{equal}(M, N)$ when $M \approx N$ ranges over the equations of E, and $\mathcal{E}q$ is the theory of equality: for each function sym-

$$\begin{aligned}
\mathtt{att}_i(\mathtt{g}(\mathtt{na}_i(A,B))) &\Leftarrow \mathtt{agent}(A), \mathtt{agent}(B)\\
\mathtt{att}_i(\mathtt{g}(\mathtt{nb}_i(A,B))) &\Leftarrow \mathtt{agent}(A), \mathtt{agent}(B)\\
\mathtt{att}_i(\{\mathtt{one}\}_{\mathtt{g}(\mathtt{na}_i(A,B)*Nb)}) &\Leftarrow \mathtt{att}_i(\mathtt{g}(Nb))\\
\mathtt{att}_2(M) &\Leftarrow \mathtt{att}_1(M)\\
\mathtt{att}_2(\mathtt{na}_1(A,B)) \quad \mathtt{att}_2(\mathtt{nb}_1(A,B)) & \\
\mathtt{att}_2(\mathtt{g}(\mathtt{na}_1(A,B)*\mathtt{nb}_1(A,B))) & \\
\bot &\Leftarrow \mathtt{att}_2(\mathtt{na}_2(\mathtt{a},\mathtt{b})*\mathtt{nb}_2(\mathtt{a},\mathtt{b}))
\end{aligned}$$

Figure 20: Diffie-Hellman protocol rules, phases, and security goal

bol f of arity k, a clause $\mathtt{equal}(f(X_1,\ldots,X_k), f(Y_1,\ldots,Y_k)) \Leftarrow \mathtt{equal}(X_1,Y_1), \ldots, \mathtt{equal}(X_k,Y_k)$, for each predicate symbol P, a clause $P(X) \Leftarrow P(Y), \mathtt{equal}(X, Y)$, and finally the clauses $\mathtt{equal}(X,X)$, $\mathtt{equal}(X,Y) \Leftarrow \mathtt{equal}(Y,X)$ and finally $\mathtt{equal}(X,Z) \Leftarrow \mathtt{equal}(X,Y), \mathtt{equal}(Y,Z)$.

This is easily achieved, using the approach of Section 5. Note that this contrasts with handling equality in automated theorem proving, which can make proof search harder (e.g., $\mathcal{H}_1$ plus equality is undecidable [42, Theorem 11]). But checking them against a finite model is no harder than in the non-equational case, and producing Coq proofs induces no extra difficulty.

We were happily surprised to see that this approach worked fine. Paradox runs slowly, but finds models with few elements on all the secure protocols we have found in the literature again.

8.1 Diffie-Hellman Key Exchange

We start with the small Diffie-Hellman protocol ($A \to B : g^{N_a}$, $B \to A : g^{N_b}$, followed by some message exchange $A \to B : \{1\}_{g^{N_a * N_b}}$), again with old compromised sessions, and more recent sessions.

Precisely, we model the Diffie-Hellman protocol by the clauses in Figure 1 ($i = 1,2$), Figure 21 ($i = 1,2$), Figure 20 ($i = 1,2$), Figure 22 and Figure 4.

The first three clauses of Figure 20 model the protocol itself, both in old and current sessions ($i = 1,2$). The next clause is just (18). The next three clauses model corruption of old values of $N_a = \mathtt{na}_1(A,B)$ and $N_b = \mathtt{nb}_1(A,B)$, together with the old session keys $g^{N_a * N_b} = \mathtt{g}(\mathtt{na}_1(A,B) * \mathtt{nb}_1(A,B))$. Finally, the last clause states that we would like the key $g^{N_a * N_b} = \mathtt{na}_2(\mathtt{a},\mathtt{b}) * \mathtt{nb}_2(\mathtt{a},\mathtt{b})$ shared between Alice ($\mathtt{a}$) and Bob ($\mathtt{b}$) in current sessions to be secret.

Figure 21 shows the additional deduction rules we require. While most of them are standard, one should note the clause $\mathtt{att}_i(\mathtt{g}(X * Y)) \Leftarrow \mathtt{att}_i(\mathtt{g}(X)), \mathtt{att}_i(Y)$, which states that one can get g^{X*Y} from g^X and Y—by computing $(g^X)^Y$. We could have modeled this by adding an equation such as $(g^X)^Y \approx g^{X*Y}$ to Figure 22, but this would have complicated the theory, and would have required us to replace the unary operation $\mathtt{g}(\_)$ by binary exponentiation. The approach we take was used in [45].

$$
\begin{array}{rcl}
\mathtt{att}_i(\mathtt{zero}) & & \mathtt{att}_i(\mathtt{one}) \\
\mathtt{att}_i(\mathtt{g}(X)) & \Leftarrow & \mathtt{att}_i(X) \\
\mathtt{att}_i(\mathtt{g}(X * Y)) & \Leftarrow & \mathtt{att}_i(\mathtt{g}(X)), \mathtt{att}_i(Y) \\
\mathtt{att}_i(X * Y) & \Leftarrow & \mathtt{att}_i(X), \mathtt{att}_i(Y) \\
\mathtt{att}_i(\mathtt{inv}(X)) & \Leftarrow & \mathtt{att}_i(X)
\end{array}
$$

Figure 21: Diffie-Hellman extra intruder deduction rules

$$
X * \mathtt{one} \approx X \quad X * Y \approx Y * X \quad X * (Y * Z) \approx (X * Y) * Z
$$
$$
X * \mathtt{inv}(X) \approx \mathtt{one} \quad \mathtt{g}(\mathtt{zero}) \approx \mathtt{one}
$$

Figure 22: Diffie-Hellman equations

Paradox finds that the common key $g^{N_a * N_b}$ of current sessions is unknown to the intruder in 0.34 s, producing a 3 element model (namely $\mathbb{Z}/3\mathbb{Z}$) with 100 entries. Using the approach of Section 5, we obtain a 641 line Coq proof of the Diffie-Hellman protocol, which is checked in 0.74s.

8.2 The EKE Protocol, Take 2

While we have already used the EKE protocol as example in Section 5, we somehow cheated. Indeed, we removed equations by superposition as a preprocessing step. However, we did not prove that any model of the preprocessed clause set could be converted to one of the original clause set.

We now run Paradox again, this time without preprocessing, and with the equations $\mathtt{dec}(\mathtt{enc}(X, Y), Y) = X$ and $\mathtt{enc}(\mathtt{dec}(X, Y), Y) = X$. Paradox finds a 4-element model in 0.40s (not the same as the one reported in Section 5, though), and the approach of Section 5 yields a $5\ 465$ line Coq proof, which is checked in 2.90s.

8.3 The SKEME Protocol

The SKEME protocol [55] allows two agents to exchange a secret key, and uses Diffie-Hellman exponentiation, plus message authentication codes (macs). Although it is meant to run in several separate phases called SHARE, EXCH, and AUTH, which are meant to be playable independently, so as to avert denial of service attacks, such phases have nothing to do with our phases. We shall call them the *sub-protocols* of SKEME. In particular, we consider that any message exchange from any of the SHARE, EXCH, and AUTH sub-protocols can be played, and even interleaved, during one of the two phases we consider, although some session of both SHARE and EXCH should have been played before AUTH can proceed. The nonces N_a, N_b, and the Diffie-Hellman secrets X_a, X_b that are created fresh in each phase. As before, we consider that these values, as created in phase 1, have been possibly disclosed in phase 2. That the protocol is secure

shows that, as claimed, SKEME has perfect forward secrecy of the final shared key $K_0 = \mathtt{h}(N_a, N_b)$.

As additional symbols, we use a two-place hash functions $\mathtt{h}$, with the Dolev-Yao intruder axiom $\mathtt{att}_i(\mathtt{h}(X,Y)) \Leftarrow \mathtt{att}_i(X), \mathtt{att}_i(Y)$, and a one-place mac function $\mathtt{mac}$, with the axiom $\mathtt{att}_i(\mathtt{mac}(X,Y)) \Leftarrow \mathtt{att}_i(X), \mathtt{att}_i(Y)$.

The three sub-protocols of the SKEME protocol are shown in Figure 23.

SHARE :
1. $A \longrightarrow B : \{A, N_a\}_{K_b}$
2. $B \longrightarrow A : \{N_b\}_{K_a}$
EXCH :
1. $A \longrightarrow B : g^{X_a}$
2. $B \longrightarrow A : g^{X_b}$
AUTH :
1. $A \longrightarrow B : \mathtt{mac}([g^{X_b}, g^{X_a}, A, B], \mathtt{h}(N_a, N_b))$
2. $B \longrightarrow A : \mathtt{mac}([g^{X_a}, g^{X_b}, B, A], \mathtt{h}(N_a, N_b))$

Figure 23: The SKEME Protocol

Paradox finds a 6 element model in 2 218s (37 minutes), and the approach of Section 5 produces a 7 352 line Coq proof, which is checked in 76s.

8.4 The Just-Fast-Keying Protocol, with Responder Security

The penultimate protocol involving an equational theory that we have tested in the JFKr protocol [5]. This one uses the Diffie-Hellman equational theory, plus asymmetric key signatures $\mathtt{sign}(M, A)$ (of message M, using A's private key). Although signatures are assumed without message recovery, the security of JFKr does not depend on signatures hiding the signed message. So, we include a clause stating that the Dolev-Yao intruder may actually be able to recover the message from its signed version.

1. $A \longrightarrow B : \mathtt{h}(N_a), g^{X_a}$
2. $B \longrightarrow A : \mathtt{h}(N_a), N_b, g^{X_b}, \mathtt{grpinfoR}, \mathtt{mac}([g^{X_b}, N_b, \mathtt{h}(N_a), \mathtt{ip}], Hk_b)$
3. $A \longrightarrow B : N_a, N_b, g^{X_a}, g^{X_b}, \mathtt{mac}([g^{X_b}, N_b, \mathtt{h}(N_a), \mathtt{ip}], Hk_b), M, \mathtt{mac}([\mathtt{tagI}, M], K_a)$
where $M = \{A, B, \mathtt{sa}, \mathtt{sign}([\mathtt{h}(N_a), N_b, g^{X_a}, g^{X_b}, \mathtt{grpinfoR}], A)\}_{K_e}$
$K_e = \mathtt{mac}([\mathtt{h}(N_a), N_b, \mathtt{one}], g^{X_a * X_b})$
$K_a = \mathtt{mac}([\mathtt{h}(N_a), N_b, \mathtt{two}], g^{X_a * X_b})$
4. $B \longrightarrow A : M', \mathtt{mac}([\mathtt{tagR}, M'], K_a)$
where $M' = \{B, \mathtt{sa}, \mathtt{sign}([g^{X_b}, N_b, g^{X_a}, \mathtt{h}(N_a)], B)\}_{K_e}$

Figure 24: The JFKr Protocol

The protocol is displayed in Figure 24, where $\mathtt{h}$ is a (unary) hash function and $\mathtt{mac}$ is a binary mac function, axiomatized as in Section 8.3. The constants $\mathtt{grpinfoR}$

1.	$A \longrightarrow B:$	A, B, N_a
2.	$B \longrightarrow S:$	A, B, N_a, N_b
3.	$S \longrightarrow B:$	$N_s,$
		$\mathtt{f}_1(N_s, N_b, A, P_b) \oplus \underbrace{\mathtt{f}_1(N_s, N_a, B, P_a)}_{K},$
		$\mathtt{f}_2(N_s, N_b, A, P_b) \oplus \underbrace{\mathtt{f}_2(N_s, N_a, B, P_a)}_{H_a},$
		$\mathtt{f}_3(N_s, N_b, A, P_b) \oplus \underbrace{\mathtt{f}_3(N_s, N_a, B, P_a)}_{H_b},$
		$\mathtt{g}(K, H_a, H_b, P_b)$
4.	$B \longrightarrow A:$	N_s, H_b
5.	$A \longrightarrow B:$	H_a

Figure 25: Gong's protocol, from SPORE

and `ip` abstract away some relatively unimportant details of the protocol: `grpinfoR` is a record containing information as to the group used in Diffie-Hellman exponentiation, and allows one to check, for example, that g is indeed a primitive element of this group, and that this group has sufficiently high order; `sa` is the so-called security association record; the constant `ip` abstracts away the IP addresses of A and B, which are easy to spoof, and cannot be trusted—so we merge all these addresses into just one constant. Other constants such as `tagI`, `tagR`, `zero`, `one`, `two`, are tags and should typically remain distinct; they are well-known to the Dolev-Yao intruder. The key Hk_b is a long term secret, known to B only. The final, secret key is $K_{ab} = \mathtt{mac}([\mathtt{h}(N_a), N_b, \mathtt{zero}], g^{X_a * X_b})$.

Paradox finds a 3-element model in 524s (8 minutes 44), and the approach of Section 5 produces a 6 335 line Coq proof, which is checked in 47.6s.

8.5 Spore's Version of Gong's Protocol

For a final, even more complicated example, we modeled Gong's protocol [39], or rather the variant from the SPORE repository [78]. This is shown in Figure 25, and uses an operator $\oplus$ (exclusive-or) that is associative, commutative, has a unit 0 and is nilpotent ($M \oplus M \approx 0$). Here $\mathtt{f}_1$, $\mathtt{f}_2$, $\mathtt{f}_3$, $\mathtt{g}$ are one-way functions, P_a is a long-term secret shared between A and S, and similarly for P_b. We omit the clauses, which again include two phases separated by an Oops move revealing all session keys from the first phase. Using Paradox, we have been able to verify that the session key $K = \mathtt{f}_1(N_s, N_a, B, P_a)$ remained secret in current sessions, from the point of view of Alice, Bob and the trusted third-party: Paradox finds a 4 element model in two hours, with 1 774 table entries.

Gong's protocol is based on the equational theory of bitwise exclusive or, shown in Figure 27.

We also need extra intruder deduction rules, shown in Figure 28.

The protocol rules are given in Figure 26. The first five clauses correspond to the

$$
\begin{aligned}
\mathtt{att}_i([A, B, \mathtt{na}_i(A, B)]) &\Leftarrow \mathtt{agent}(A), \mathtt{agent}(B)\\
\mathtt{att}_i([A, B, N_a, \mathtt{nb}_i(A, B, N_a)]) &\Leftarrow \mathtt{att}_i([A, B, N_a])\\
\mathtt{att}_i([\,&\mathtt{ns}_i(A, B, N_a, N_b),\\
&\mathtt{f}_1(\mathtt{ns}_i(A, B, N_a, N_b), N_b, A, \mathtt{p}(B)) \oplus \mathtt{f}_1(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\mathtt{f}_2(\mathtt{ns}_i(A, B, N_a, N_b), N_b, A, \mathtt{p}(B)) \oplus \mathtt{f}_2(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\mathtt{f}_3(\mathtt{ns}_i(A, B, N_a, N_b), N_b, A, \mathtt{p}(B)) \oplus \mathtt{f}_3(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\mathtt{g}(\,\mathtt{f}_1(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\quad\mathtt{f}_2(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\quad\mathtt{f}_3(\mathtt{ns}_i(A, B, N_a, N_b), N_a, B, \mathtt{p}(A)),\\
&\quad\mathtt{p}(B))\\
&\quad\,]) \Leftarrow \mathtt{att}_i([A, B, N_a, N_b])\\
\mathtt{att}_i([N_s, H_b]) &\Leftarrow \mathtt{att}_i([\,N_s,\\
&\qquad\mathtt{f}_1(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus K),\\
&\qquad\mathtt{f}_2(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus H_a,\\
&\qquad\mathtt{f}_3(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus H_b,\\
&\qquad\mathtt{g}(K, H_a, H_b, \mathtt{p}(B))])\\
\mathtt{att}_i(\mathtt{f}_2(N_s, \mathtt{na}_i(A, B), B, \mathtt{p}(A))) &\Leftarrow \mathtt{att}_i([N_s, \mathtt{f}_3(N_s, \mathtt{na}_i(A, B), B, \mathtt{p}(A))])\\
\mathtt{alice\_key}_i(A, \mathtt{f}_1(N_s, \mathtt{na}_i(A, B), B, \mathtt{p}(A))) &\Leftarrow \mathtt{att}_i([N_s, \mathtt{f}_3(N_s, \mathtt{na}_i(A, B), B, \mathtt{p}(A))])\\
\mathtt{bob\_key}_i(B, K) &\Leftarrow \mathtt{att}_i([\,N_s,\\
&\qquad\mathtt{f}_1(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus K),\\
&\qquad\mathtt{f}_2(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus H_a,\\
&\qquad\mathtt{f}_3(N_s, \mathtt{nb}_i(A, B, Na), A, \mathtt{p}(B)) \oplus H_b,\\
&\qquad\mathtt{g}(K, H_a, H_b, \mathtt{p}(B))]),\\
&\quad\mathtt{att}_i(H_a)
\end{aligned}
$$

Figure 26: Gong protocol rules

$$
(X \oplus Y) \oplus Z \approx X \oplus (Y \oplus Z) \quad X \oplus Y \approx Y \oplus X
$$
$$
X \oplus \mathtt{zero} \approx X \quad X \oplus X \approx \mathtt{zero}
$$

Figure 27: Axiomatizing xor

$$
\mathtt{att}_i(\mathtt{zero}) \qquad \mathtt{att}_i(X \oplus Y) \Leftarrow \mathtt{att}_i(X), \mathtt{att}_i(Y)
$$

Figure 28: Gong extra intruder deduction rules

$$\begin{aligned}
\mathtt{att}_2(M) &\Leftarrow \mathtt{att}_1(M)\\
\mathtt{att}_2(\mathtt{f}_1(\mathtt{ns}_1(A,B,N_a,N_b),N_a,B,\mathtt{p}(A))) &\\
\mathtt{att}_2(\mathtt{na}_1(A,B)) &\\
\mathtt{att}_2(\mathtt{nb}_1(A,B,N_a)) &\\
\mathtt{att}_2(\mathtt{ns}_1(A,B,N_a,N_b)) &
\end{aligned}$$

Figure 29: Phases in Gong's protocol

five messages of Figure 25, the last two clauses define the keys K that Alice (A) and Bob (B) get, respectively. In Bob's case, note that we obtain K from message 3, and we check the value of H_a using message 5. The latter just means checking whether $\mathtt{att}_i(H_a)$ holds in our model.

Handling phases is done by slight variants of the rules of Figure 5, shown in Figure 29. We now assume the old keys $\mathtt{f}_1(\mathtt{ns}_1(A,B,N_a,N_b),N_a,B,\mathtt{p}(A))$ are known in phase 2, as well as all old nonces.

Our security goals are again, that all session keys, as generated by the server, and as received by Alice and Bob, are unknown to the intruder, see Figure 30.

$$\begin{aligned}
\bot &\Leftarrow \mathtt{att}_2(\mathtt{f}_1(\mathtt{ns}_2(\mathtt{a},\mathtt{b},N_a,N_b)))\\
\bot &\Leftarrow \mathtt{att}_2(K_{ab}),\mathtt{alice_key}_2(\mathtt{a},K_{ab})\\
\bot &\Leftarrow \mathtt{att}_2(K_{ab}),\mathtt{bob_key}_2(\mathtt{b},K_{ab})
\end{aligned}$$

Figure 30: (Negated) security goals for Gong's protocol

Finally, Gong's protocol as a whole is defined by the rules in Figures 6, 4, 27, 1, 28, 26, and 30.

Using the approach of Section 5, we have produced a 2 555 line Coq proof of Gong's protocol, which is checked in 1 204 s (20 minutes).

9 Conclusion

We hope to have demonstrated, first, that producing formally checkable proofs from first-order formulations S of security goals π was difficult, and sometimes more difficult than verification itself.

On the other hand, we hope to have shown that formal Coq proofs of security could be extracted and checked efficiently from a model (in the explicit model approach of Section 5), or from a model-checking process (in the automata-theoretic approach of Section 7). A summary of our results can be found in Figure 31.

This endeavor is a first step towards formally verifying full security protocols, and many things remain to be done. For one, complementing this work with formally

Protocol		Finding the model Time	#elts	#entries	#checks	Coq proofs #lines	time
Without equality:							
NS [66]	(Paradox)	1.62s	4	824	3 908	1 038	0+3.29s
	(h1)	0.70s	46	217 312	430 10^6	15 560	0.53+10.73s
amended	(Paradox)	–	–	–	–	–	–
NS [67]	(h1)	1.71s	57	188 724	1.245 10^9	31 640	1.90+25.67s
NSL7	(Paradox)	4.85s	4	2 729	2 208	4 415	0+1.76s
[67, 59]	(h1)	8.03s		over-approximated		–	–
Yahalom	(Paradox)	3 190s	6	5 480	38 864	14 646	0+36.6s
[72]	(h1)	4.82s	≥ 57		$\geq$ 2.46 10^9	60 938	7.34+53.77s
Kerberos	(Paradox)	17.87s	5	1 767	5 518	2 584	0+2.57s
[19]	(h1)	0.94s	57	7 952	84.5 10^6	30 326	2.02+23.97s
X.509 [78]	(Paradox)	3 395s	4	142 487	12 670	35 472	0+11.01s
	(h1)	0.44s	≥ 29		$\geq$ 228.5 10^6	20 471	0.95+23.33s
EAP-AKA	(Paradox)	54.3s	3	2 447	1 457	3 763	0+4.42s
[7]	(h1)	1.93s	72	22 550	7.74 10^9	32 229	0.88+43.30s
EKE [11]	(Paradox)	0.44s	4	3 697	4 632	5 023	0+1.99s
	(h1)	1.88s	48	16 016	64.5 10^6	73 683	3.18+89.94s
Requiring an equational theory (using Paradox):							
Diffie-Hellman [35]		0.34s	3	229	1 191	641	0+0.74s
EKE [11]		0.40s	4	1 055	9 939	5 465	0+2.90s
SKEME [55]		2 218s	6	1 968	125 753	7 352	0+76s
JFKr [5]		524s	3	577	13 028	6 335	0+47.6s
Gong [78]		7 161s	4	4 066	471 145	2 555	0+1 204s

Figure 31: Summary of practical results

checkable proofs of computational soundness of the Dolev-Yao model, when it is indeed sound [51, 79], would be desirable. There is a growing interest from industrial firms and defense agencies towards formally checked proofs of security models, and we believe our work solves an important part of it.

Another necessary step is to find techniques that would scale up better. While Paradox and the explicit model approach of Section 5 work fine when there is a model of at most, say, 6 elements, the automata-theoretic approach of Section 7 handles much larger models, but cannot cope with equational theories yet. However, note that the number of elements of a model is a very bad measure of its size: function and predicate tables are much larger than what the number of elements suggests. We have also observed that the size of the model is independent of the size of the protocol to be proved secure. Rather, the size of the model seems to be correlated to its logical complexity. In particular, we have observed, reproducing an experiment by Koen Claessen, that some safe C implementations of roles in the Needham-Schroeder asymmetric key protocol [44] only required models with 3 elements.

It remains to be examined whether scaling up is necessary, or is in fact a non-problem. The experiments we conducted show, for example, that Paradox, although generally slower than `h1` on non-equational problems (or equational problems that can be converted to non-equational problems), tends to find models with very few elements almost all the time. Further research might help in finding models with possibly more elements, but faster, and which would be easier to check, using `h1mc` for example.

9.1 Acknowledgments

We presented early findings at JFLA [41]: we thank the organizers and all the people who were there. David Lubicz, Bruno Blanchet, and Steve Kremer provided support by showing interest in this research. Thanks also to Koen Claessen, who suggested the use of Paradox to me. Finally, thanks to Ankit Gupta, to Stéphanie Delaune, to Cédric Fournet, and to the anonymous reviewers.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[3] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. *SIGPLAN Notices*, 36(3):104–115, 2001.

[4] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols. *Information and Computation*, 148(1):1–70, Jan. 1999.

[5] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold. Just fast keying: Key agreement in a hostile Internet. *ACM Transactions on Information and System Security*, 7(2):1–30, May 2004.

[6] R. Amadio and W. Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *Proc. 13th International Conference on Concurrency Theory (CONCUR'02)*, pages 499–514. Springer-Verlag LNCS 2421, 2002.

[7] AVISPA—automated validation of Internet security protocols and applications. Web site, 2006. `http://avispa-project.org/`.

[8] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [75], chapter 2, pages 19–99.

[9] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 75–83. IEEE Computer Society Press, 1993.

[10] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58–74, Mar. 2009.

[11] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proc. 13th IEEE Symp. Research in Security and Privacy (S&P'93)*, pages 72–84, Oakland, CA, May 1992. IEEE Computer Society Press.

[12] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer Verlag, 2004. 469 pages.

[13] K. Bhargavan, C. Fournet, A. D. Gordon, and A. R. Pucella. Tulafale: A security tool for Web services. In *Proc. 2nd International Symposium on Formal Methods for Components and Objects (FMCO'03)*, pages 197–222. Springer Verlag LNCS 3188, 2004.

[14] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE Computer Society Press, 2001.

[15] B. Blanchet. An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In R. Nieuwenhuis, editor, *Proc. 20th International Conference on Automated Deduction (CADE-20)*, Tallinn, Estonia, July 2005. Springer Verlag LNAI 3632.

[16] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[17] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Proc. 3rd ACM Conference on Computer and Communications Security (CCS'96)*, New Delhi, India, Mar. 1996. ACM Press.

[18] J. Bull and D. J. Otway. The authentication protocol. Technical Report DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03, Defence Research Agency, Malvern, UK, 1997.

[19] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, 426(1871):233–271, Dec. 1989.

[20] R. Chadha, S. Kremer, and A. Scedrov. Formal analysis of multi-party contract signing. *Journal of Automated Reasoning*, 36(1-2):39–83, Jan. 2006.

[21] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model building. In P. Baumgartner, editor, *Proc. CADE-19 Workshop W4*, Miami, Florida, July 2003.

[22] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature, v1.0. `http://citeseer.ist.psu.edu/clark97survey.html`, 1997.

[23] K. L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Non-monotonic Reasoning*, pages 311–325, San Francisco, California, 1987. Morgan Kaufmann Publishers.

[24] H. Comon. Inductionless induction. In Robinson and Voronkov [75], chapter 14, pages 913–962.

[25] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `www.grappa.univ-lille3.fr/tata`, 1997. Version of Sep. 6, 2005.

[26] H. Comon and R. Nieuwenhuis. Induction=i-axiomatization+first-order consistency. *Information and Computation*, 159(1–2):151–186, 2000.

[27] H. Comon-Lundh and V. Cortier. Security properties: Two agents are sufficient. *Science of Computer Programming*, 50(1–3):51–71, 2004.

[28] R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess what? Here is a new tool that finds some new guessing attacks. In R. Gorrieri and R. Lucchi, editors, *Proc. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues in the Theory of Security (WITS'03)*, pages 62–71, Warsaw, Poland, Apr. 2003.

[29] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[30] V. Cortier, S. Delaune, and G. Steel. A formal theory of key conjuring. In *Proc. 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 79–93, Venice, Italy, July 2007. IEEE Computer Society Press.

[31] V. Cortier, M. Rusinowitch, and E. Zălinescu. Relating two standard notions of secrecy. *Logical Methods in Computer Science*, 3(3:2):1–29, 2007. `http://arxiv.org/pdf/0706.0502`.

[32] A. Dawar. Model-checking first-order logic: Automata and locality. In J. Duparc and T. A. Henzinger, editors, *Proc. 21st International Workshop on Computer Science Logic, 16th Annual Conference of the EACSL (CSL'07)*, page 6, Lausanne, Switzerland, Sept. 2007. Springer Verlag LNCS 4646.

[33] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, Aug. 1981.

[34] P. Devienne, P. Lebègue, A. Parrain, J.-C. Routier, and J. Würtz. Smallest Horn clause programs. *Journal of Logic Programming*, 27(3):227–267, 1994.

[35] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[36] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[37] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proc. Workshop on Formal Methods and Security Protocols (FMSP'99)*, Trento, Italy, July 1999.

[38] T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th Annual IEEE Symposium Logic in Computer Science (LICS'91)*, pages 300–309. IEEE Computer Society Press, 1991.

[39] L. Gong. Using one-way functions for authentication. *Computer Communication Review*, 19(5):8–11, Oct. 1989.

[40] J. Goubault-Larrecq. *The* `h1` *Tool Suite*. LSV, ENS Cachan, CNRS, INRIA projet SECSI, 2003. `http://www.lsv.ens-cachan.fr/~goubault/H1.dist/dh1index.html`.

[41] J. Goubault-Larrecq. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In V. Ménissier-Morain, editor, *Actes des 15èmes Journées Francophones sur les Langages Applicatifs (JFLA'04)*, pages 1–40, Sainte-Marie-de-Ré, France, Jan. 2004. INRIA. Invited paper.

[42] J. Goubault-Larrecq. Deciding $\mathcal{H}_1$ by resolution. *Information Processing Letters*, 95(3):401–408, Aug. 2005.

[43] J. Goubault-Larrecq. Towards producing formally checkable security proofs, automatically. In *Proc. 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 224–238. IEEE Computer Society Press, June 2008.

[44] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *Proc. ¡6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379, Paris, France, Jan. 2005. Springer.

[45] J. Goubault-Larrecq, M. Roger, and K. N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, Aug. 2005.

[46] M. Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26:401–406, 1980.

[47] M. Hofmann. A simple model for quotient types. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Proc. 2nd Intl. Conf. Typed Lambda Calculi and Applications (TLCA '95)*, pages 216–234, Edinburgh, UK, Apr. 1995. Springer Verlag LNCS 902.

[48] P. V. Homeier. Quotient types. In R. J. Boulton and P. B. Jackson, editors, *Supplemental Proceedings, 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, pages 191–206, Sept. 2001. Number EDI-INF-RR-0046 in Informatics Report Series, Division of Informatics, University of Edinburgh, `http://www.inf.ed.ac.uk/publications/report/0046.html`.

[49] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Workshop on Formal Methods and Security Protocols (FMSP'99)*, Trento, Italy, July 1999.

[50] Information technology – security techniques – evaluation criteria for IT security. ISO/IEC 15408 Standard, 2005. Parts 1-3, `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`.

[51] R. Janvier, Y. Lakhnech, and L. Mazaré. Relating the symbolic and computational models of security protocols using hashes. In P. Degano, R. Küsters, L. Viganò, and S. Zdancewic, editors, *Proc. Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA'06)*, pages 67–89, Seattle, Washington, USA, Aug. 2006. Informal proceedings at `http://www.easychair.org/FLoC-06/fcs-arspa06.pdf`.

[52] I. L. K. Kao and R. Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29(3):14–21, 1995.

[53] D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31:125–157, 1987.

[54] D. C. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer, 1997. 400 pages.

[55] H. Krawczyk. SKEME: A versatile secure key exchange mechanism for Internet. In *Proc. 4th Internet Society Symposium on Network and Distributed Systems Security (SNDSS'96)*, pages 114–127. IEEE Computer Society Press, Feb. 1996.

[56] S. Kremer. Computational soundness of equational theories (tutorial). In G. Barthe and C. Fournet, editors, *Proc. 3rd Symposium on Trustworthy Global Computing (TGC'07)*, pages 363–382, Sophia-Antipolis, France, 2008. Springer Verlag LNCS 4912.

[57] R. Küsters and T. Trudering. On the automatic analysis of recursive security protocols with XOR. In W. Thomas and P. Weil, editors, *Proc. 24th Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, pages 646–657. Springer Verlag LNCS 4393, 2007.

[58] D. S. Lankford. A simple explanation of inductionless induction. Technical Report MTP-14, Math. Dept., Louisiana State University, 1981.

[59] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1996.

[60] G. Lowe. A family of attacks upon authentication protocols. Technical Report 1997/5, Dept. Mathematics and Computer Science, U. Leicester, 1997.

[61] J. Marcinkowski and L. Pacholski. Undecidability of the Horn clause implication problem. In *Proc. 33rd Annual Symposium on Foundations of Computer Science (FOCS'92)*, pages 354–362, Pittsburgh, Pennsylvania, 1992. IEEE Computer Society Press.

[62] R. Matzinger. Computational representations of Herbrand models using grammars. Technical Report TR-WB-Mat-96-2, Technische Universität Wien, Feb. 1997. version 4.0.

[63] W. McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003.

[64] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In M. Naor, editor, *Proc. 1st IACR Theory of Cryptography Conference (TCC'04)*, pages 133–151, Cambridge, Massachussetts, Feb. 2004. Springer Verlag LNCS 2951.

[65] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Prox. 6th International Static Analysis Symposium (SAS'99)*, pages 149–163. Springer Verlag LNCS 1694, Sept. 1999.

[66] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, Dec. 1978.

[67] R. M. Needham and M. D. Schroeder. Authentication revisited. *ACM SIGOPS Operating Systems Review*, 21(1):7, Jan. 1987.

[68] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *Proc. 9th International Static Analysis Symposium (SAS'02)*, pages 20–35. Springer Verlag LNCS 2477, Sept. 2002.

[69] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM SIGOPS Operating Systems Review*, 21(1):8–10, Jan. 1987.

[70] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC Series*, pages 361–386. Academic Press, 1990.

[71] L. C. Paulson. Proving properties of security protocols by induction. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 70–83, Rockport, Massachussetts, 1997. IEEE Computer Society Press.

[72] L. C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*, 9(3):197–216, Jan. 2001.

[73] O. Pereira and J.-J. Quisquater. A security analysis of the cliques protocols suites. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 73–81. IEEE Computer Society Press, June 2001.

[74] X. Rival and J. Goubault-Larrecq. Experiments with finite tree automata in Coq. In R. J. Boulton and P. B. Jackson, editors, *Proc. 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, pages 362–377, Edinburgh, Scotland, UK, Sept. 2001. Springer Verlag LNCS 2152.

[75] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.

[76] P. Y. A. Ryan and S. A. Schneider. An attack on a recursive authentication protocol: A cautionary tale. *Information Processing Letters*, 65(1):7–10, 1998.

[77] P. Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1):73–87, July 2001. Proc. 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV'01).

[78] Spore—security protocols open repository. `http://www.lsv.ens-cachan.fr/spore/`, 2005.

[79] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *Proc. 19th IEEE Computer Security Foundations Symposium Workshop (CSFW'06)*, pages 153–166. IEEE Computer Society Press, Washington, DC, USA, 2006.

[80] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8):769–780, 2000.

[81] T. Tammet. *Resolution Methods for Decision Problems and Finite-Model Building*. PhD thesis, Göteborg University, 1992.

[82] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2-3):191–230, Jan. 1999.

[83] Writing for the TPHOLs community. Part of the Guide for Authors of TPHOL conferences (Theorem Proving in Higher-Order Logics) since 1999, see `http://www-sop.inria.fr/croap/TPHOLs99/authors.html`, 1999.

[84] C. Weidenbach. Towards an automatic analysis of security protocols. In H. Ganzinger, editor, *Proc. 16th International Conference on Automated Deduction (CADE-16)*, pages 378–382, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[85] C. Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [75], chapter 27, pages 1965–2013.

[86] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. SPASS version 2.0. In A. Voronkov, editor, *Proc. 18th International Conference on Automated Deduction (CADE'02)*. Springer-Verlag LNAI 2392, July 2002.