Jean Goubault-Larrecq

Towards Producing Formally Checkable Security Proofs, Automatically

Research Report LSV-08-15

April 2008

Laboratoire Spécification et Vérification

ENS CACHAN

CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Towards Producing Formally Checkable Security Proofs, Automatically*

Jean Goubault-Larrecq goubault@lsv.ens-cachan.fr
LSV, ENS Cachan, CNRS, INRIA 61, avenue du président Wilson 94230 Cachan

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 evaluation level 7? We demonstrate that this is surprisingly hard: the problem is non-recursive in general. On the practical side, we show how we can extract finite models $\mathcal{M}$ from a set S of clauses representing π, automatically, in two ways. 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. This was implemented in the `h1` *tool suite. Experience on a number of protocols shows that this is practical.*

1 Introduction

So far, automated verification of cryptographic protocols in models in the style of Dolev and Yao [28] has been considered under a variety of angles: (un)decidability results [30, 38], practical decision procedures [49, 68, 5], extension to security properties other than secrecy and authentication (e.g., [15]), to protocols requiring equational theories, to soundness with respect to computational models (e.g., [43] 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 (secure/possibly insecure), but also formal proofs, which could be checked by one of the established tools, e.g., Isabelle [55] or Coq [8]. This is required in Common Criteria certification of computer products at the highest assurance level, EAL 7 [39], 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 platform. 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 [11].

It is certainly the case that hand-crafted formal proofs (e.g., [13, 56]) 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 a first answer to that question.

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 [62]. We observe that representing finite models explicitly 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.

Acknowledgments. We presented early findings at JFLA [34]: we thank the organizers and all the people who were there. David Lubicz, Bruno Blanchet, and Steve Kremer all suggested recently this was interesting. Thanks also to Koen Claessen, who suggested the use of Paradox to me. Finally, thanks to Ankit Gupta and to Stéphanie Delaune.

2 Related Work

Many frameworks and techniques have been proposed to verify security protocols in models inspired from Dolev and Yao [28]. It would be too long to cite them all. However, whether they are based on first-order proving [68, 22, 11],

*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.

tree automata [49], set constraints [5], 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 (a fact first observed by Bachmair, Ganzinger, and Waldmann [7]). 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 $\mathcal{H}_1$ [53].

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 $s, t, u, v, \ldots$, predicate symbols $P, Q, \ldots$, variables $X, Y, Z, \ldots$ 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 [6]. 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 [12]. Weak and strong secrecy are, in fact, close notions under reasonable assumptions [25].

We also concentrate on security proofs in *logical* models, derived from the Dolev-Yao model [28]. Proofs in *computational* models would probably be more relevant. E.g., naive Dolev-Yao models may be computationally unsound [48]. 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. [40], or [64]. 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 [62]. 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 [34] 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 [50, 46].

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* [41], or *inductionless induction* [45, 19]. 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 [21], in the case of first-order logic with equality and induction along the recursive path ordering.

We shall use 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 [26]. 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.

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 [66]. 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 [59]; 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 [10], to Selinger's [62], 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 assume $\mathtt{crypt}(M, K)$ denotes 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 [50].

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.

The protocol itself is modeled in a simple way, originally inspired from strand spaces [67], and similarly to Blanchet

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

Figure 1. Intruder capabilities

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

[10]. 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.

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 $\mathtt{a}$, $\mathtt{b}$, $\mathtt{s}$, and $\mathtt{i}$ for the attacker), is due to the fact that we wish to model unboundedly many sessions of the protocol in parallel. E.g., (11)

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

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

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

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

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

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

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

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

Figure 3. Protocol rules

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

Figure 4. Agents

$$\mathtt{att}_2(M) \Leftarrow \mathtt{att}_1(M) \quad (18)$$

$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_1(A, B, N_a))) \quad (19)$$

$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{na}_1(A, B))) \quad (20)$$

$$\mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{nb}_1(K_{ab}, A, B))) \quad (21)$$

Figure 5. Phases

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 ($\mathtt{a}$), Bob ($\mathtt{b}$), the trusted third-party ($\mathtt{s}$), and the Dolev-Yao attacker $\mathtt{i}$. Since we only deal with secrecy, considering this many agents is sufficient [22].

To model the fact that secrets may be corrupted over time, we distinguish two *phases* $i = 1, 2$. Intuitively, phase 1 represents sessions that are old enough that the old session keys $\mathtt{k}(\mathtt{sym}, \mathtt{session}_1(A, B, Na))$ 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 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 [56].

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.

$$\mathtt{att}_i(X) \Leftarrow \mathtt{agent}(X)$$

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

Figure 6. The attacker's initial knowledge

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.

$$\bot \Leftarrow \mathtt{att}_2(\mathtt{k}(\mathtt{sym}, \mathtt{session}_2(\mathtt{a}, \mathtt{b}, N_a))) \quad (22)$$

$$\bot \Leftarrow \mathtt{att}_2(K_{ab}), \mathtt{alice\_key}_2(\mathtt{a}, K_{ab}) \quad (23)$$

$$\bot \Leftarrow \mathtt{att}_2(K_{ab}), \mathtt{bob\_key}_2(\mathtt{b}, K_{ab}) \quad (24)$$

Figure 7. (Negated) security goals

Call, somewhat abusively, the *protocol* π the collection of the cryptographic protocol itself, the (Dolev-Yao) secu-

rity model, the security assumptions and the security goals. The clause set S_{NS} denoting the symmetric-key Needham-Schroeder protocol 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 [70] 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 [33], 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 [35, Proposition 3], producing an approximation S' in the decidable class $\mathcal{H}_1$ [53, 68]; then `h1` decides S' by the terminating resolution algorithm of [35]. 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 [6]).

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) produce 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. Furthermore, (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 [62], 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: 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 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 [27], 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 [47], 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" [6] 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 [6], 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 [6], as well as the kind of resolution used in SPASS [69] 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 [62] 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 K. Claessen's tool Paradox. As for our running example, our tool `h1` finds a 46 element model for S^{safe}_{NS} (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 *recur-*

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**

sive 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 [44]. Examples of such protocols are Bull and Otway's recursive authentication protocol [14], or the IKA protocols [65]. Note however that both are flawed [61, 58], 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 [16] 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.

Paradox represents such 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: !3 and !4 can be thought as values that will remain secret, but encrypting !4 with the compromised datum !2 will produce !2, which is known to the attacker in phase 2.

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 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 $\mathtt{true}(\mathtt{not}(F^*(\vec{A})))$. 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. We have conducted a small experiment on secrecy protocols found in the Spore library [63]—be warned that the proportion of secrecy protocols that are in fact secure is small—, see Figure 9. The NS row is our example S_{NS}^{safe}, while the amended NS row is a corrected version [51] 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 NSL row is the 7-message version of the Needham-Schroeder-Lowe protocol, checking that the secrecy of the exchanged messages is preserved, instead of

Protocol	Paradox			h1		
	#elts	#entries	#checks	#elts	#entries	#checks
NS	4	824	3 908	46	217 312	$430\ 10^6$
amended NS [51]	?(≥ 5)	?	?	57	188 724	$1.245\ 10^9$
NSL	4	2 729	2 208	over-approximated		
Yahalom [57]	6	5 480	38 864	> 56	–	

Figure 9. Model sizes

mutual authentication. 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 [57]. 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`. Paradox finds a 4 element model for NS in 1.6 s, a 4 element model for NSL in 4.85 s, a 6 element model for Yahalom in 53 min, and hasn't found one for amended NS in 8 hours 1/4; the only thing we know is that the least model contains at least 5 elements.

From an explicitly presented finite model $\mathcal{M}$, it is in principle easy to extract a formally checkable proof. In Coq, we declare an inductive type of values of $\mathcal{M}$, e.g., `Inductive` M : `Set` := `v1` : M | `v2` : M | `v3` : M | `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:

```
Definition crypt(m : M)(k : M) : M :=
  match m, k with
    v1, v1 ⇒ v1 | v1, v2 ⇒ v1 | v1, v3 ⇒ v4 | v1, v4 ⇒ v1
  | v2, v1 ⇒ v2 | v2, v2 ⇒ v1 | v2, v3 ⇒ v4 | v2, v4 ⇒ v4
  | v3, v1 ⇒ v3 | v3, v2 ⇒ v4 | v3, v3 ⇒ v4 | v3, v4 ⇒ v4
  | v4, v1 ⇒ v3 | v4, v2 ⇒ v2 | v4, v3 ⇒ v2 | v4, v4 ⇒ v2
  end.
```

and $\mathtt{att}_2$ would be described by `Definition` $\mathtt{att}_2(m : M)$: `Prop` := `match` m `with v1` ⇒ `True` | `v2` ⇒ `True` | _ ⇒ `False end`. 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 would then be very short: if C contains k free variables, we would write its proof in Coq's tactic language as `intro` x_1; `case` x_1; ... `intro` x_k; `case` x_k; `simpl`; `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 9), but also produces a proof term of size proportional to it.

We conclude that the explicitly presented models approach would only work for small models. While this approach is applicable for the 4 to 6 element models that Paradox found in Figure 9, 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. The Coq proofs corresponding to NS, NSL, and Yahalom are 748, resp. 1 038, resp. 3 581 lines long, and are checked by Coq in 3.29 s, 1.76 s, and 36.6 s respectively.

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 [57], 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 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 $\{[\mathtt{na}_i(A, B), B, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}$. 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 who 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 [13] for an early example). This is needed in verifying protocols where sessions must be sequential, e.g., for

the Otway-Rees protocol [54], which is secure if sessions are sequential, but insecure if sessions can be run in parallel [17]. 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 [9], or cryptographic APIs [24], in order of increased complexity.

Our model-checking technique will be able to check the larger models found by `h1` (see Figure 9). Some of it rests on intuitions on how we decide $\mathcal{H}_1$ by resolution [35], and the relationship to tree automata.

For each satisfiable set S of Horn clauses, and each predicate symbol 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 consists only of *tree automaton clauses* $P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n)$ (X_1, …, X_n pairwise distinct), this coincides with the usual definition of the set of terms recognized at P; such clauses are just tree automaton transitions from P_1, …, P_n to P. Accordingly, we shall call a set of tree automaton clauses a (tree) automaton. This connection between tree automata and Horn clauses was pioneered by Frühwirth *et al*. [31]; 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 [20], 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 [20, 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:

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

where $B_1(X_1), \ldots, B_k(X_k)$ are ϵ-blocks, and $X_1, \ldots, X_k$ are pairwise distinct. *Universal clauses* are of the form $P(X)$. An *alternating tree automaton* is any set S of alternating automaton clauses and universal clauses. (The standard definition [20] 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.) 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 [35, Proposition 3] to get a clause set S' in the decidable class $\mathcal{H}_1$ [53, 68], and such that S is satisfiable whenever S' is. Then `h1` saturates S' by ordered resolution with selection [35], 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 [35, 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 [33]. 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, …, 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 [42], 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 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 $Det(S_{prod})$, we add the *table entry* $f(q_1, \ldots, q_k) = q$ to the model. 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 9: 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. 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.*

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

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

Figure 10. Basic model-checking rules

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 [20, 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. □

To define our model-checking procedure (Figure 10, Figure 11), we need 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. The brace notation used there 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., $C_1, \ldots, C_n$ are all non-empty and share no free variable. The rules in Figure 11 apply only if no rule from Figure 10 applies. This implies that no universal predicate occurs on the right of $\vdash$.

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

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

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

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

Figure 11. Model-checking rules, end

To produce a Coq proof that $Det(S_{prod}) \models S$, we check that $\vdash C$ for each clause C in S. Our tool `h1mc`, also part of the `h1` tool suite [33], 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, ..., 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:*

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

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 $\forall X_1, \ldots, X_k \cdot \bigwedge_{\substack{1\le i\le k\\ 1\le j\le m}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow C(X_1, \ldots, X_k) \Rightarrow E(X_1) \vee \neg P(X_1)$ 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 $\forall X_2, \ldots, X_k, X_{k+1}, \ldots, X_{k+p} \cdot \bigwedge_{\substack{2\le i\le k+p\\ 1\le j\le m+1}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow E(f(\vec{X})) \vee D$. 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 $\forall X_1, \ldots, X_k \cdot \bigwedge_{\substack{1\le i\le k\\ 1\le j\le m}} (\forall X \preceq X_i \cdot E_j(X)) \Rightarrow C(X_1, \ldots, X_k) \Rightarrow E(X_1) \vee \neg P(X_1)$ 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. Typically, 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. See Appendix A for details.

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.

Finally, `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.

Finally, we describe additional optimizations in Appendix C. Experiments show that these are less important when it comes to model-checking alternating tree automata produced by `h1`, but are crucial in case one wants to model-check models produced by Paradox.

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 $\mathrm{lfp}_\Sigma\, T_{S_{prod}}$. Then:

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.*

We omit the proof, for lack of space. In any case, this is less central to our work. Also, neither soundness nor completeness has to be part of our trusted base: the onus of correctness rests entirely on Coq itself.

Since `h1`, as a resolution engine, produces proof that are independent on Σ, any set S_{prod} produced by `h1` from S will be such that not only $lfpT_{S_{prod}} \models S$, but the stronger assumption of Proposition 7.4 is satisfied. 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 $(-P\ Elim)$.)

	$Det(S_{prod})$			Coq proof		
Protocol	#elts	#entries	#checks	size	#lines	time
NS	46	217 312	$430\ 10^6$	1.3 Mb	27 318	1.86 s+41.7 s
am. NS	57	188 724	$1.245\ 10^9$	1.7 Mb	36 285	4.9 s+66.2 s
Yahalom	> 56		–	2.5 Mb	48 609	12.6 s+91.5 s

Figure 12. Coq proofs

On the other hand, the approach of Figure 10 and Figure 11, 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 12 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 9 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, `h1mc` manages to find a proof in a reasonable amount of time.

8 Equational Theories

More and more protocols in the literature can only be modeled using equational theories, to represent e.g. bitwise exclusive-or or modular exponentiation [23]. While `h1` really cannot deal with such equational theories, this is in principle easy to Paradox: just add the needed equations as unit clauses. For example, Figure 13 lists axioms for modular exponentiation as used in Diffie-Hellman key agreement, 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.)

We were happily surprised to see that this approach worked fine. (In particular, that secure protocols found in the literature again tend to have models with few elements.) We started 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. see Appendix D.

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

Figure 13. Diffie-Hellman equations

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 14. Gong's protocol, from SPORE

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.

For a more complicated example, we modeled Gong's protocol [32], or rather the variant from the SPORE repository [63]. This is shown in Figure 14, 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.

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 $S \cup \widetilde{E} \cup \mathcal{E}q$ against $\mathcal{M}$, 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 symbol 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 $\mathtt{equal}(X, Z) \Leftarrow \mathtt{equal}(X, Y), \mathtt{equal}(Y, Z)$.

This is easily done. 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 [35, Theorem 11]). But checking them against a finite model is no harder than in the non-equational case. Using the approach of Section 5, we have produced a 641 line Coq proof of the Diffie-Hellman protocol this way, which is checked in 0.74 s, and 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. The most frustrating aspect of things is that there seems to be no practically usable way of exploiting the fact that ProVerif, SPASS or `h1` concluded that there was no attack, to infer a proof of it.

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).

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 checkable proofs of computational soundness of the Dolev-Yao model, when it is indeed sound [40, 64], 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 [36] only required models with 3 elements. It remains to be examined whether scaling up is necessary, or is in fact a non-problem.

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] R. Amadio and W. Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *CONCUR'02*, pages 499–514. Springer-Verlag LNCS 2421, 2002.

[6] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [60], chapter 2, pages 19–99.

[7] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 75–83, 1993.

[8] 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.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and A. R. Pucella. Tulafale: A security tool for web services. In *Intl. Symp. Formal Methods for Components and Objects (FMCO'03)*, pages 197–222. Springer Verlag LNCS 3188, 2004.

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

[11] B. Blanchet. An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In *20th International Conference on Automated Deduction (CADE-20)*, Tallinn, Estonia, July 2005.

[12] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340. IEEE Computer Society, June 2005.

[13] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communication Security*, 1996.

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

[15] 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.

[16] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model building. In P. Baumgartner, editor, *Proc. CADE-19 Workshop W4*, Miami, FL, July 2003. `http://www.uni-koblenz.de/~peter/models03/final/soerensson/main.ps`.

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

[18] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, NY, 1978. Plenum Press.

[19] H. Comon. Inductionless induction. In R. David, editor, *2nd Int. Conf. in Logic For Computer Science: Automated Deduction. Lecture notes*, Chambéry, 1994. Univ. de Savoie.

[20] 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.

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

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

[23] 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.

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

[25] V. Cortier, M. Rusinowitch, and E. Zălinescu. Relating two standard notions of secrecy. *Logical Methods in Computer Science*, 3(2), 2007.

[26] A. Dawar. Model-checking first-order logic: Automata and locality. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, page 6. Springer, 2007.

[27] 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.

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

[29] P. K. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[30] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, 1999.

[31] T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th Symp. Logic in Computer Science*, pages 300–309. IEEE Computer Society Press, 1991.

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

[33] 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`.

[34] 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.

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

[36] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *Proceedings of the 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.

[37] 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.

[38] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. FLOC Workshop on Formal Methods in Security Protocols*, Trento, Italy, 1999.

[39] 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`.

[40] 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, *Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA'06)*, Seattle, Washington, USA, Aug. 2006.

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

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

[43] S. Kremer. Computational soundness of equational theories (tutorial). In G. Barthe and C. Fournet, editors, *Proceedings of the 3rd Symposium on Trustworthy Global Computing (TGC'07)*, Lecture Notes in Computer Science, Sophia-Antipolis, France, 2008. Springer. To appear.

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

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

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

[47] J. Marcinkowski and L. Pacholski. Undecidability of the Horn clause implication problem. In *Proc. 33rd Ann. Symp. Foundations of Computer Science (FOCS'92)*, pages 354–362, Pittsburgh, PA, 1992.

[48] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In M. Naor, editor, *Theory of Cryptography Conference - Proceedings of TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151, Cambridge, MA, USA, Feb. 2004. Springer.

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

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

[51] R. M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(12), Dec. 1987.

[52] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, avril 1980.

[53] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *9th Static Analysis Symposium (SAS)*. Springer Verlag LNCS 2477, 2002.

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

[55] 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.

[56] L. C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

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

[58] O. Pereira and J.-J. Quisquater. A security analysis of the cliques protocols suites. In *14th IEEE Computer Security Foundations Workshop*, pages 73–81, June 2001.

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

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

[61] 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.

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

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

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

[65] 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.

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

[67] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

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

[69] C. Weidenbach. *Combining Superposition, Sorts and Splitting*, chapter 27, pages 1965–2012. In Robinson and Voronkov [60], 2001.

[70] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. SPASS version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag LNAI 2392, 2002.

A Simulations in Alternating Tree Automata

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 ... and $B_k \subseteq NE(S)$, then $P \in NE(S)$. 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 [20, 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:

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

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

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

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 A.1 *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 $B_1 \subseteq NE(S_{prod}), \ldots, B_k \subseteq NE(S_{prod})$. 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 A.2 *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 A.1, 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`.

B Completeness

We wish to prove that, if C holds in $\mathrm{lfp}\, T_{S_{prod}}$, then we can derive $\vdash C$ using the model-checking rules. As stated, this is not quite true, and we need to make a technical point.

We decided early in this paper to assume the first-order signature Σ to be fixed. However, 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 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 $\vdash C$. The difficulty 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.

Proposition 7.4. *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 the familiar *Clark completion* from logic programming [18]. 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. □

C Other Optimizations in `h1mc`

Given a clause C, `h1mc` applies the following rules in turn. Some of these are rules from Figure 10 and Figure 11, some others are specific optimizations that have been found to be particularly effective.

The first thing that `h1mc` tries on a clause C in history Γ is to find whether:

- C has already been proved in some history $\Gamma' \subseteq \Gamma$. If so, $\Gamma \vdash C$ holds.
- Or checking $\Gamma \vdash C$ (with the same history Γ) has already been done, and failed. If so, $\Gamma \vdash C$ does not hold.

In all other cases, we cannot conclude yet. So `h1mc` refines these checks to look for an ϵ-clause $E(X)$ that would subsume C, and such that $\Gamma' \vdash \forall X \cdot E(X)$ was already shown to hold for some history $\Gamma' \subseteq \Gamma$. We could look for clauses that are not ϵ-clauses instead of $E(X)$ here, but ϵ-clauses are simpler to implement. This is naturally done by using and maintaining specific tables.

If these checks failed, `h1mc` turns to the so-called CheckSimple optimization.

C.1 The CheckSimple Optimization

A particularly effective way of checking a Horn clause $C = (P(t) \Leftarrow \mathcal{B})$ under history Γ is given by the following algorithm `CheckHornSimple`(C):

- If P is universal, then return true: C clearly holds in $Det(S_{prod})$. (This in particular subsumes the $(+Univ)$ rule.)
- If t is a variable X, and there is an atom $Q(X)$ in $\mathcal{B}$, with $Q\ \ R\ \ P$ for some simulation R on S_{prod}, then return true: the clause C holds in $Det(S_{prod})$. (See Section A for the notion of simulation.)
- If t is of the form $f(t_1, \ldots, t_n)$, then check whether there is an automaton clause $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$ in S_{prod} such that, for all i, $1 \leq i \leq n$, for every predicate $P' \in B_i$, `CheckHornSimple`$(P'(t_i) \Leftarrow \mathcal{B})$ returns true. If so, C again holds in $Det(S_{prod})$, so return true.
- In all other cases, return false.

This is a procedure that often manages to prove the input clause C valid in $Det(S_{prod})$ while avoiding the use of the costly rule $(+P, f\ \ Elim)$. We memoize the head to make this run in polynomial time.

In general, when C is a clause that is not necessarily Horn, say $P_1(t_1) \vee \ldots \vee P_k(t_k) \Leftarrow \mathcal{B}$, we let `CheckSimple`$(C)$ return true if and only if `CheckHornSimple`$(P_j(t_j) \Leftarrow \mathcal{B})$ returns true for some j, $1 \leq j \leq k$—in this case, too, C holds in $Det(S_{prod})$.

Otherwise, we turn to loop-checking.

C.2 Loop-Checking

Loop-checking amounts, basically, to checking whether rule $(Loop)$ applies, i.e., whether the current clause C occurs in the history Γ. We actually check whether there is a smaller clause $C' \subseteq C$ such that C' is in Γ. If so, $\Gamma \vdash C$ holds.

C.3 The $(-Univ)$ Rule

Otherwise, we try to apply $(-Univ)$, i.e., to find a universal predicate in the body of C.

If this fails, too, we use the NegEmptyInter optimization.

C.4 The NegEmptyInter Optimization

We may check that $\bot \Leftarrow P(X), Q(X)$, i.e., that the languages $L_P(S_{prod})$ and $L_Q(S_{prod})$ are disjoint, in polynomial time, see below. In this case, we say that P and Q are *disjoint*. We use this to conclude directly that $C = (P(t) \Leftarrow \mathcal{B})$ holds in $Det(S_{prod})$ as soon as there are two atoms $Q_1(u)$ and $Q_2(u)$ in $\mathcal{B}$, with the same u, and with Q_1 and Q_2 disjoint.

While this sometimes produces shorter Coq proofs from alternating tree automata S_{prod} as found by `h1`, this rule really shines when S_{prod} is a deterministic tree automaton, as produced by Paradox. In this case indeed, all states are disjoint.

To detect disjointness, we use two separate algorithms. The CheckDisjointSimple algorithm works in a similar way as the simulation finding algorithm. However, it is only used in case the second one, the CheckDisjointVerySimple algorithm, fails. The reason for using two algorithms is that CheckDisjointSimple will sometimes produce Coq proofs that are so big that Coq's guardedness checker (which verifies that recursive functions are indeed defined properly, by structural induction on the relevant arguments) just blows up.

We only describe the basic idea behind each algorithm.

CheckDisjointSimple is actually a function `CheckDisjointSimple`(B) taking a set B of predicates as arguments, and returns true only when B contains two disjoint predicates P and P' that it can prove disjoint. It enumerates all distinct pairs P, P' in B, and returns true in case P and P' are not universal, and for all alternating tree automaton clauses of the form $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$ and $P'(f(X_1, \ldots, X_n)) \Leftarrow B'_1(X_1), \ldots, B'_n(X_n)$ (with the same f), `CheckDisjointSimple`$(B_i \cup B'_i)$ returns true for some i. This again needs some form of loop-checking: we return true on encountering a pair P, P' on which we are looping.

CheckDisjointVerySimple may fail to conclude, in which case we turn to CheckDisjointSimple. Precisely, CheckDisjointVerySimple fails if and only if there is an alternating tree automaton clause $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$ in S_{prod} where P is not universal, and some B_i, $1 \leq i \leq n$, consists entirely of universal predicates. Call such clauses *unacceptable*. Otherwise, all clauses in S_{prod} are *acceptable*. Then, we use an auxiliary trick, based on congruence closure, a fast technique to decide ground equalities in ground equational theories [52, 29]. For each non-empty block $B(X) = P_1(X), \ldots, P_n(X)$, let $E(B)$ be the set of equations $P_1 \approx P_2, P_2 \approx P_3, \ldots, P_{n-1} \approx P_n$. For each acceptable alternating tree automaton clause C, say $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$, let $e(C)$ be the equation $f(P_1, \ldots, P_n) \approx P$, where P_1 is a predicate picked from B_1, $P_2 \in B_2$, ..., $P_n \in B_n$. Let $E(C)$ be $E(B_1) \cup \ldots \cup E(B_n) \cup \{e(C)\}$, and finally let $E(S_{prod})$ be the union of all $E(C), C \in S_{prod}$.

It is easy to see by induction on t that, for every ground term t, if t is recognized at P in S_{prod}, and P is not universal, then $t \approx P$ is provable in $E(S_{prod})$. In particular, if P and P' are two predicates such that $E(S_{prod}) \nvdash P \approx P'$, then P and P' are necessarily disjoint.

Observe that this trick in fact computes, in polynomial time, a *deterministic* tree automaton $\mathcal{A}$, whose states are equivalence classes $[P]$ of predicates P, modulo provable equality from $E(S_{prod})$. There is a transition $[P](f(X_1, \ldots, X_n)) \Leftarrow [P_1](X_1), \ldots, [P_n](X_n)$ in $\mathcal{A}$ whenever $E(S_{prod}) \vdash f(P_1, \ldots, P_n) \approx P$. $\mathcal{A}$ is a projection of S_{prod}, in the sense that any ground term recognized at P in S_{prod} is recognized at $[P]$ in $\mathcal{A}$ (provided P is not universal in S_{prod}). In general, $\mathcal{A}$ is different from $Det(S_{prod})$: for one, $\mathcal{A}$ is computed in polynomial time, while determinizing S_{prod} may take exponential time. On the other hand, if S_{prod} is already deterministic and contains no universal predicate, then $\mathcal{A}$ retrieves S_{prod} exactly.

If the current goal $\Gamma \vdash C$ could not be proved by realizing that the body of C contained two atoms $P(u)$ and $P'(u)$ with P and P' disjoint, `h1mc` proceeds to the next step: splitting.

C.5 Splitting

Splitting is just the $(Split)$ rule. While it is in principle only needed on disjunctions $C = (C_1 \vee \ldots \vee C_n)$ of ϵ-blocks, it is profitable in practice to spend some time trying

to block-decompose more complex clauses C eagerly. Then `h1mc` tries to prove $\Gamma \vdash C_1$, then $\Gamma \vdash C_2$ if the latter failed, ..., and finally $\Gamma \vdash C_n$ if all previous cases failed.

We use a simple heuristic to test first for those clauses C_i that will put less strain on `h1mc`. Let the weight of a clause C_i be 1 if it is definite (Horn, and head is not $\bot$) and has a non-empty body, 2 if C_i is non-Horn and has a non-empty body, 3 if C_i has an empty body and a head other than $\bot$, and 4 if C_i has $\bot$ as head. We prefer clauses with smaller weights, and in case of a tie, clauses with smaller sizes. This was obtained through trial and error.

C.6 The $(-P, f\ Elim)$ Rule

Only if all previous rules failed do we attempt to apply the $(-P, f\ Elim)$ rule. We may pick any non variable atom $P(f(t_1, \ldots, t_n))$ from the body of C. Currently, `h1mc` just picks the first one it encounters from C.

C.7 Checking for Explicit Counterexamples

If all previous rules failed, in particular C is of the form $P(t) \Leftarrow \mathcal{B}$, where the body $\mathcal{B}$ does not contain any function symbol, we can only apply one of the rules $(+P, f\ Elim)$ or $(-P\ Elim)$—or even $(+Elim)$ in case the signature is assumed fixed.

However, these rules are costly: $(+P, f\ Elim)$ takes exponential time by itself in the worst case, $(+Elim)$ produces premises that will most likely have to be dealt using $(+P, f\ Elim)$, and $(-P\ Elim)$, while not specifically costly, is the one rule that is responsible for increasing the history Γ, and Γ can increase up to an exponential number of elements.

We therefore try to do whatever is possible before we apply these rules, as a last resort.

The first thing we do is check for explicit counterexamples. For each predicate P such that P is non-empty in S_{prod}, i.e., such that there is a ground term t recognized at P in S_{prod}, we pick one. Write this term t_P. Then, letting C be $P(t) \Leftarrow P_{11}(X_1), \ldots, P_{1k_1}(X_1), \ldots, P_{n1}(X_n), \ldots, P_{nk_n}(X_n)$, we check whether $t_{P_{i1}}$ is recognized at $P_{i2}, \ldots, P_{ik_i}$, and $t[t_{P_{11}}/X_1, \ldots, t_{P_{n1}}/X_n]$ is *not* recognized at P. If so, we have found a ground instance $C\sigma$ of C that does not hold in $Det(S_{prod})$, so C cannot hold: `h1mc` returns false. (I.e., it does not go on and try the next rules: there is no hope of proving C.)

So much for the rough idea. We must make three things more precise.

First, the case where some P_{ij} is empty will be dealt with later, as part as the FindNegEpsilon optimization. In this case, we shall naturally conclude that $\Gamma \vdash C$ holds.

Second, while testing whether some given term is recognized at a given state can be done in polynomial time, finding the term t_P is as hard as testing whether P is non-empty, which is **EXPTIME**-complete. Instead, we use a heuristic, which may fail to return a term t_P even when P is non-empty: to find t_P, we look for some automaton clause $P(f(X_1, \ldots, X_n)) \Leftarrow B_1(X_1), \ldots, B_n(X_n)$, and for each i, $1 \leq i \leq n$, we look for a ground term t_{B_i} that would be recognized at all predicates of B_i. In case B_i is empty, we let t_{B_i} be some arbitrary constant. Otherwise, let $B_i = \{P_1, \ldots, P_m\}$, $m \geq 1$, then we let t_{B_i} be t_{P_1} provided t_{P_1} is defined and recognized at $P_2, \ldots, P_m$ as well. This again necessitates some loop-checking: we just fail and backtrack on encountering the same predicate P twice.

Third, we cheated a bit. In case C has a ground instance $C\sigma$ that does not hold in $Det(S_{prod})$, we can only in principle conclude that checking $\Gamma \vdash C$ fails provided Γ is *empty*. Indeed, if $\Gamma \vdash C$ then C holds under the assumptions in Γ. It might in principle be the case that $C\sigma$ does not hold, but this case would be ruled out by one of the induction hypotheses in Γ. Here is a hand-waving argument justifying why we can conclude, even when Γ is non empty. (While a formal proof would be better, remember that it does not really matter whether `h1mc` is proved correct: in the end, Coq is the sole judge for deciding whether the proof that `h1mc` produces is indeed correct or not.) Now, let $H_1, \ldots, H_m$ be the list of induction hypotheses in Γ, in the order that they have been introduced by the use of the $(-P\ Elim)$ rule. If $\Gamma \vdash C$ indeed held, with C false, necessarily some induction hypothesis H_i in Γ would be false, too. Now, by the form of $(-P\ Elim)$, H_i was introduced into the history when trying to establish $H_1, \ldots, H_{i-1} \vdash H_i$. Since H_i is false, either the goal $H_1, \ldots, H_{i-1} \vdash H_i$ will eventually fail anyway, or some H_j is false again, with $j < i$. Eventually, we shall reach a minimal j such that H_j is false. Repeating the argument above, the goal $H_1, \ldots, H_{j-1} \vdash H_j$ will eventually fail, so we can safely fail the descendant subgoal $H_1, \ldots, H_m \vdash C$.

C.8 The FindNegEpsilon Optimization

The FindNegEpsilon optimization is meant to exhume some further opportunities for the CheckSimple optimization to succeed. The point of FindNegEpsilon is, at the very least, to undo some nasty effects of converting Paradox-like models into deterministic tree automata. Consider for example that `crypt` is defined so that $\texttt{crypt}(!4, !2) = !2$ in Figure 8, and that $\texttt{att}_2(!2)$ is true. A direct translation of these into Horn clauses would yield:

$$\begin{array}{rcl} !2(\texttt{crypt}(X_1, X_2)) & \Leftarrow & !4(X_1), !2(X_2) \\ \texttt{att}_2(X) & \Leftarrow & !2(X) \end{array} \quad (28)$$

However, (28) is not a tree automaton clause. The actual translation we use would replace the latter by all clauses of the form $\mathtt{att}_2(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$, for all clauses of the form $!2(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$ deduced from the function tables. I.e., we would generate $\mathtt{att}_2(\mathtt{crypt}(X_1,X_2)) \Leftarrow !4(X_1), !2(X_2)$, among others.

This has the unfortunate consequence that checking clauses such as $\mathtt{att}_2(\mathtt{suc}(X)) \Leftarrow \mathtt{att}_2(X)$ is hard: we must use rule $(-P\ Elim)$ (with $P = !2$ here) first, then conduct an induction. The CheckSimple rule will not work here. However, the FindNegSimple optimization will (re)discover that $\mathtt{att}_2(X)$ can only be true provided $!2(X)$ or $!1(X)$ is true (in the example), then proceed to check whether we can prove $\mathtt{att}_2(\mathtt{suc}(X)) \Leftarrow !2(X)$ and $\mathtt{att}_2(\mathtt{suc}(X)) \Leftarrow !1(X)$. Experiments show that the latter can then be proved easily using the CheckSimple rule. Moreover, if this fails, then $\mathtt{att}_2(\mathtt{suc}(X)) \Leftarrow \mathtt{att}_2(X)$ is in fact not provable, because $\mathtt{att}_2(X)$ is in fact equivalent with $!2(X) \vee !1(X)$ in $Det(S_{prod})$.

The main task of FindNegEpsilon is to find such equivalences, of the form $P(X) \Leftrightarrow P_1(X) \vee \ldots \vee P_m(X)$, which are valid in $Det(S_{prod})$. It only does this for non-universal P, and looks for $P_1, \ldots, P_m$ that are *states*, i.e., predicates that occur in the body of at least one clause in S_{prod}. To do this, FindNegEpsilon enumerates the clauses of the form $P(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$ in S_{prod} (with P given), and collects in a state $\mathcal{Q}$ all states P_i, $1 \leq i \leq m$, such that S_{prod} also contains a clause of the exact form $P_i(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$. FindNegEpsilon then checks whether for every $P_i \in \mathcal{Q}$, for every clause of the form $P_i(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$, the clause $P(f(X_1,\ldots,X_n)) \Leftarrow \mathcal{B}$ is in S_{prod}. If so, then it is easy to see that $P(X) \Leftrightarrow P_1(X) \vee \ldots \vee P_m(X)$ is valid in $Det(S_{prod})$. Otherwise, FindNegEpsilon fails and we jump to the FindSomeCaseAnalysis optimization.

Given our current goal $\Gamma \vdash C$, `h1mc` looks for an atom $P(t)$ in the body of C such that FindNegEpsilon was able to discover an equivalence of the form $P(X) \Leftrightarrow P_1(X) \vee \ldots \vee P_m(X)$. (In case of ties, we choose one with the smallest m.) Write C as $C_0 \vee -P(t)$. Then `h1mc` replaces the task of checking $\Gamma \vdash C$ by the m tasks of checking $\Gamma \vdash C_0 \vee -P_i(t), 1 \leq i \leq m$, recursively.

Surprisingly, FindNegEpsilon also simplifies checking alternating tree automata produced by `h1` somewhat. In a number of cases, FindNegEpsilon actually discovers an equivalence of the form $P(X) \Leftrightarrow P_1(X) \vee \ldots \vee P_m(X)$ with $m = 0$, i.e., that P is empty. This has the effect of simplifying checking $\Gamma \vdash C_0 \vee -P(t)$ to the task of checking $\Gamma \vdash C_0$.

C.9 The FindSomeCaseAnalysis Optimization

In case the signature Σ is fixed to some signature Σ_0 (using the `-exact-sig` flag to `h1mc`), one may hope to show that the disjunction $q_1(X) \vee \ldots \vee q_m(X)$ holds in $Det(S_{prod})$, where $q_1, \ldots, q_m$ are the states of S_{prod}, as defined in Section C.8. If so, S_{prod} is a *complete* deterministic automaton. This is the case of automata produced from Paradox models.

We use a simple heuristic, named CheckExplicitUniv, to conclude that S_{prod} is complete is to check that for every function symbol f in Σ_0, of arity, say, k, for every indices j, i_1, ..., i_k in the interval $[1,m]$, the clause $q_j(f(X_1,\ldots,X_k)) \Leftarrow q_{i_1}(X_1),\ldots,q_{i_k}(X_k)$ is in S_{prod}, or subsumed by some clause in S_{prod}. (This is sound, but not complete. A complete procedure would be **EXPTIME**-complete again.)

If CheckExplicitUniv succeeds, then we know that $q_1(X) \vee \ldots \vee q_m(X)$ holds in $Det(S_{prod})$. The FindSomeCaseAnalysis then tries to find a variable X that would be free in some positive atom of C but would not be free in any negative atom of C, and replaces the task of checking $\Gamma \vdash C$ by those of checking $\Gamma \vdash C \vee -q_i(X)$ for each i, $1 \leq i \leq m$. In particular, checking a clause of the form $P(f(X,Y)) \Leftarrow Q(X)$ will turn into checking $P(f(X,Y)) \Leftarrow Q(X), q_i(Y)$ for each i. This gives new opportunities to apply CheckSimple.

Note that the CheckExplicitUniv heuristic only applies in the `-exact-sig` case, hence not to the alternating tree models found by `h1`. Its purpose, then, is to try to make the `h1mc` model-checker fare no worse than the simple approach of Section 5, up to some constant factor, on explicit, finite models, as found by Paradox. Unfortunately, on large examples such as Gong's protocol, this is still not enough to make it competitive. We have not been able to model-check the model found by Paradox on this example using 2GB of main memory, and the Coq proof that was generated was already 669MB long (7.36 million lines) when we interrupted `h1mc` after three hours. The model found by Paradox for Diffie-Hellman is checked by `h1mc` in 1.03 s, and generates a 25 660 line Coq proof that is checked in 60.4 s; `h1mc` takes 7.5 s on NS, generates a 89 308 line Coq proof that is checked in 746 s (a bit less than 13 minutes), and produces proofs for Yahalom and amended NS that are too big for Coq to verify.

C.10 The $(+P, f\ Elim)$ Rule

Iff all previous rules fail, in particular the goal to check is $\Gamma \vdash C$ with C an unsplittable clause whose body does not contain function symbols. If C contains some positive atom of the form $+P(f(t_1,\ldots,t_n))$, i.e., with some func-

tion symbol f in it, then `h1mc` applies the $(+P, f\ Elim)$ rule.

It may be that the rule applies to several positive atoms containing some function symbol. We use the following heuristic to pick one positive atom. Imagine we pick such a non variable atom, and C_0 is the rest of the clause C. Then $(+P, f\ Elim)$ will require us to examine all clauses in S_{prod} with a head of the form $P(f(\bar{X}))$, say $P(f(\bar{X})) \Leftarrow \mathcal{B}_i(\bar{X}), 1 \leq i \leq m$. Then, we shall generate the premises $C_0 \vee \bigwedge_{i=1}^{m} \bigvee \mathcal{B}_i(\bar{t}), 1 \leq i \leq m$. Let $\mathcal{B}_i$ contain k_i atoms. In the general case, $(+P, f\ Elim)$ will require us to generate $k_1 \times \ldots \times k_m$ new clauses as premises, as a cnf of the latter. The heuristic we use is then simply to find one positive atom that minimizes $k_1 \times \ldots \times k_m$.

C.11 The Universal Case

Finally, if all else failed, then C is an unsplittable clause not containing any function symbol, i.e., an ϵ-clause $E(X)$, then we distinguish two cases.

If $E(X)$contains some negative atom, i.e., is of the form $E'(X) \vee \neg P(X)$, then we apply $(-P\ Elim)$. This means checking, recursively, all premises $\Gamma, \forall X \cdot E(X) \vdash E'(f(\vec{X})) \vee D$ in turn, where $P(f(\vec{X})) \vee D$ ranges over the clauses in S_{prod}/P.

If $E(X)$ contains no negative atom, i.e., is of the form $P_1(X) \vee \ldots \vee P_n(X)$, then either the `-exact-sig` option was not given, meaning that we want to check $\mathrm{lfp}_{\Sigma}\, T_{S_{prod}} \models S$ for every Σ extending Σ_0 (see Proposition 7.4), and then no rule from Figure 10 or 11 applies: $\Gamma \vdash E(X)$ does not hold. Or the `-exact-sig` option was given, meaning we wish to check $\mathrm{lfp}_{\Sigma_0}\, T_{S_{prod}} \models S$, and we apply rule $(+Elim)$. However, we have already argued this was costly, because it entailed using the $(+P, f\ Elim)$ rule next. So, we first apply the CheckExplicitUniv procedure (see Section C.9); if CheckExplicitUniv finds that $P_1(X) \vee \ldots \vee P_n(X)$ holds in $Det(S_{prod})$, then $\Gamma \vdash E(X)$ holds. Otherwise, we apply $(+Elim)$.

D Modeling the Diffie-Hellman Protocol

We model the Diffie-Hellman protocol by the clauses in Figure 1 ($i = 1, 2$), Figure 16 ($i = 1, 2$), Figure 15 ($i = 1, 2$), Figure 13 and Figure 4.

The first three clauses of Figure 15 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 (`a`) and Bob (`b`) in current sessions to be secret.

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

Figure 15. Diffie-Hellman protocol rules, phases, and security goal

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

Figure 16. Diffie-Hellman extra intruder deduction rules

Figure 16 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 13, 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 [37].

E Modeling Gong's Protocol

Gong's protocol is based on the equational theory of bitwise exclusive or, shown in Figure 18.

We also need extra intruder deduction rules, shown in Figure 19.

The protocol rules are given in Figure 17. The first five clauses correspond to the five messages of Figure 14, the last two clauses define the keys K that Alice (A) and Bob

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

Figure 18. Axiomatizing xor

$$
\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))]),\\
&\mathtt{att}_i(H_a)
\end{aligned}
$$

Figure 17. Gong protocol rules

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

Figure 19. 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 20. Phases in Gong's protocol

(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 20. 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 21.

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

Figure 21. (Negated) security goals for Gong's protocol

Finally, Gong's protocol as a whole is defined by the rules in Figures 6, 4, 18, 1, 19, 17, and 21.