A Logic Programming Based Framework for Security Protocol Verification

Shujing Wang and Yan Zhang

Intelligent Systems Laboratory
School of Computing and Mathematics
University of Western Sydney
Penrith South DC, NSW 1797, Australia
E-mail: {shwang,yan}@scm.uws.edu.au

Abstract. Security protocol analysis has been a major research topic in information security and recognised to be a notoriously hard problem. In this paper, we take the advantage of answer set programming technology to develop an effective framework to verify security protocols carrying claimed security proof under adversary models on computational complexity theory. In our approach, a security protocol, adversary actions and attacks can be formally specified within a unified logic program. Then the verification is performed in an automatic manner by computing the stable models of the underlying logic program. We use Boyd-González Nieto conference key agreement protocol as our case study protocol to demonstrate the effectiveness and efficiency of our approach.

1 Introduction

In recent years, security protocols are increasingly being used in many diverse secure electronic communications and electronic commerce applications. However, despite an enormous amount of research effort expended in design and analysis of such protocols, it is still notoriously hard. When security protocols are designed by hand, errors may creep in by combining protocols actions in ways not foreseen by the designer [3]. Some protocols have been found errors after they were published many years, even since they have been proven secure [10].

The study of cryptographic protocols has led to the dichotomization of cryptographic protocol analysis techniques between the formal methods approach and the computational complexity approach [8]. The formal methods approach is to use logic based methods including model checking and theorem proving to automatically verify a protocol. The computational complexity approach, on the other hand, adopts a reductive process which allows a proven reduction from the problem of breaking the protocol to another problem believed to be hard. These two approaches have been developed in two mostly different communities. Recently, some research works have been done to bridge the gap between them, which achieve automatic provability under classical computational models, see [2, 4] for example.

In this paper, based on the answer set programming approach [5], we propose a framework to analyze security protocols that are found insecure against certain types of attacks. We use Boyd-González Nieto conference key agreement protocol as our case study protocol to demonstrate our approach.

2 Logic Programming Specification for Security Protocols

Modelling security protocols

Now we present how to model a security protocol through the case study protocol (refer to the appendix A for the complete specification program). For specification simplicity and efficiency, we simplify the case study protocol to a two-party protocol showed in Figure 2 as explained in [9]. Because in a protocol flow of Figure 1, message 1 and 2 can be sent concurrently, in the simplified protocol, we merge them into one message.

1. $U_1 \rightarrow U_2 : \mathcal{U} = \{U_1, U_2\}, S_{d_{U_1}}(\mathcal{U}, \{N_1\}_{e_{U_2}}), \{N_1\}_{e_{U_2}}$
2. $U_2 \rightarrow U_1 : U_2, N_2$

$$SK_{U_1} = H(N_1||N_2) = SK_{U_2}$$

Fig. 1. Simplified Boyd-González Nieto Conference Key Agreement Protocol

Let $\mathcal{U} = \{U_1, U_2\}$. The initiator, U_1 encrypts N_1 using the public key of U_2, signs $\mathcal{U}$ and the encrypted nonce $\{N_1\}_{e_{U_2}}$, and broadcasts $\mathcal{U}$, the signature value and the encrypted nonce in message flow 1. The principal, U_2, upon receiving the initiate message, will respond with his/her identity and a random nonce in message flow 2.

The first part of protocol specification is to set up principals and their keys through predicates, $player(A)$, $agent(A)$, $ag\_id(A, N)$, and $key(K)$, where K is one of key functions. For instance, in our case study protocol, we have

$player(u_1), player(u_2), adversary(a)$
$ag\_id(u_1, 0), ag\_id(u_2, 1), ag\_id(a, 2)$
$key(pKey(A)) \leftarrow agent(A).$
$key(sKey(A)) \leftarrow agent(A).$
$key(sig\_sKey(A)) \leftarrow agent(A).$
$key(sig\_vKey(A)) \leftarrow agent(A).$

The second part is to model relationships between keys of principals. In the case study protocol, there are encryption and signature keys which are specified as follows.

$asymKeyPair(pKey(A), sKey(A)) \leftarrow agent(A).$
$asymKeyPair(sig\_sKey(A), sig\_vKey(A)) \leftarrow agent(A).$
$asymKeyPair(K_1, K_2) \leftarrow asymKeyPair(K_2, K_1).$

The third part is about message flows in a protocol. During a protocol run, we assume that if a principal A sends a message to B and the adversary does not intercept it, B will receive it at the next time. We model the assumption using the rule:

$$gets(B, M, P, T+1) \leftarrow sends(A, B, M, P, T), \\ neq(A, B), not\ intercept(a, M, P, T+1).$$

A protocol consists of a sequence of messages. Except the first message which is sent by the initiator of the protocol run, principals will check preconditions before they send a response message. As explained in [3], a protocol was denoted like:

$$
\begin{array}{l}
A \rightarrow B_{i_1} : m_{i_1}, p_{i_1} \ \ \% \ first\ message\ A\ must\ send, \ldots \\
B_{j_1} \rightarrow A : m_{j_1}, p_{j_1} \ \ \% \ first\ message\ A\ must\ receive, \ldots \\
A \rightarrow B_{i_s} : m_{i_s}, p_{i_s} \ \ \% \ last\ message\ A\ must\ send\ before\ m, \ldots \\
B_{i_r} \rightarrow A : m_{i_r}, p_{i_r} \ \ \% \ last\ message\ A\ must\ receive\ before\ m, \ldots \\
A \rightarrow B : m, p
\end{array}
$$

As showed above, principal A will sends message (m, p) to B, if we check that a sequence of messages should have been received and sent before (m, p) in a correct run. We code the following rule:

$$
\begin{array}{l}
sends(A, B, m, p, T+1) \leftarrow \\
\quad sends(A, B_{i_1}, m_{i_1}, p_{i_1}, T_{i_1}), \ldots, sends(A, B_{i_s}, m_{i_s}, p_{i_s}, T_{i_s}), \\
\quad gets(A, m_{j_1}, p_{j_1}, T_{j_1}), \ldots, gets(A, m_{i_r}, p_{i_r}, T_{i_r}), T_{j_1} > T_{i_1}, \ldots, T_{i_r} > \\
T_{i_s}, \\
\quad protocol\text{-}dependant\ literals \\
contains(m, p, msg(.)).
\end{array}
$$

We consider preconditions for sending message m by principal A as actions that A has performed in previous steps according to the protocol run. Protocol dependant literals are usually to check the freshness of random nonces or timestamps and other conditions needed by particular protocols. Because we represent a message using a message id and type in predicate $sends$, we should add a fact rule, in which $contains$ is the head to denote what the message is indeed.

For instance, in our case study protocol, principal u_1 sends an initial message to start a protocol run. We model it as the following rule.

$$
\begin{array}{l}
sends(u_1, all, 0, 0, 0). \\
contains(0, 0, agset(u1, u2)). \\
contains(0, 0, sign(sig_s Key(u_1), agset(u1, u2) || enc(pKey(u_2), n(0)))). \\
contains(0, 0, enc(pKey(u_2), n(0))).
\end{array}
$$

Finally, we model the principal knowledge including the principal initial knowledge base and knowledge change during the protocol run. Each principal taking part in the protocol run has an initial knowledge base such as other principals' public keys. While sending and receiving messages, principals will hold them and derive more information by breaking or decrypting all messages for which they have a key. Their knowledge will change during the protocol run. We use predicate $holds$ to specify principals' knowledge.

For encryption and signature keys in the case study protocol, we code initial knowledge bases for principals using following rules.

$$
\begin{array}{l}
holds(A, pKey(B), 0) \leftarrow agent(A), agent(B). \\
holds(A, sig\_vKey(B), 0) \leftarrow agent(A), agent(B). \\
holds(A, sKey(A), 0) \leftarrow agent(A). \\
holds(A, sig\_sKey(A), 0) \leftarrow agent(A).
\end{array}
$$

Then we write following rules to model principals' knowledge change during the protocol run.

$$
\begin{aligned}
&holds(A, M, T) \leftarrow gets(A, M, P, T).\\
&holds(A, M, T) \leftarrow sends(A, B, M, P, T).\\
&holds(A, S, T) \leftarrow holds(A, M, T), contains(M, P, S).\\
&holds(A, S_1, T) \leftarrow holds(A, M, T), contains(M, P, S_1 || \ldots || S_n).\\
&\quad \ldots\\
&holds(A, S_n, T) \leftarrow holds(A, M, T), contains(M, P, S_1 || \ldots || S_n).\\
&holds(A, S_1, T) \leftarrow holds(A, enc(K_1, S_1 || \ldots || S_n), T),\\
&\qquad\qquad holds(A, K_2, T_1), asymKeyPair(K_1, K_2).\\
&\quad \ldots\\
&holds(A, S_n, T) \leftarrow holds(A, enc(K_1, S_1 || \ldots || S_n), T),\\
&\qquad\qquad holds(A, K_2, T_1), asymKeyPair(K_1, K_2).
\end{aligned}
$$

Modelling attacks

In our framework, the adversary model is closely based on Bellare-Rogaway model. If protocols with claimed security under Bellare-Rogaway model are found to be violating any of the conditions in the definition of *insecurity*, they will be insecure in Bellare-Rogaway model. Moreover, the proof of the protocol will also be invalid. Based on the definition of *insecurity*, we should model SIDs and session keys of principals. The SID of a principal is the concatenation of all messages he receives and sends. We use predicate *inSidList(U,M)* to record the messages that the principal U receives and sends.

$$
\begin{aligned}
&inSidList(U, M) \leftarrow sends(U, all, M, P, T).\\
&inSidList(U, M) \leftarrow gets(U, M, P, T).
\end{aligned}
$$

The following two rules specify that two principals have same SIDs, where the first one denotes that if a message is in the session id list of principal U_1, and not in the session id list of principal U_2, $sid\_neq\_pair(U_1, U_2)$ is true, and the second one specifies conditions which should be satisfied for two principals to have same SIDs.

$$
\begin{aligned}
&sid\_neq\_pair(U_1, U_2) \leftarrow\\
&\qquad inSidList(U_1, M),\ not\ inSidList(U_2, M),\ neq(U_1, U_2).\\
&same\_sid\_pair(U_1, U_2) \leftarrow\\
&\qquad not\ sid\_neq\_pair(U_1, U_2),\ not\ sid\_neq\_pair(U_2, U_1),\ neq(U_1, U_2).
\end{aligned}
$$

In our case study protocol, the session key of a principal is a one-way hush function of the concatenations of random nonces of all principals taking part in the conference protocol.

$$
\begin{aligned}
&sk(A, h(n(M_1), n(M_2))) \leftarrow holds(A, agset(B, C), T),\\
&\qquad holds(A, nonce(B, n(M_1)), T_1),\ holds(A, nonce(C, n(M_2)), T_2).
\end{aligned}
$$

The following rule models that principal U_1 and U_2 have same session keys.

$$
\begin{aligned}
&same\_sk\_pair(U_1, U_2) \leftarrow\\
&\qquad sk(U_1, h(n(M_1), n(M_2))),\ sk(U_2, h(n(M_1), n(M_2))),\ neq(U_1, U_2).
\end{aligned}
$$

Consider the condition 1 in *insecurity* definition as an instance, if two non-partner oracles have the same session keys, the protocol is insecure. Here two oracles are not partners if they have different SIDs. The attack is modelled as follows.

$$attack \leftarrow same\_sk\_pair(U_1, U2),\ not\ same\_sid\_pair(U_1, U2).$$

Note that $same\_sk\_pair(U_1, U_2)$ denotes that principal U_1 and U_2 have same session keys and $same\_sid\_pair(U_1, U_2)$ denotes that principal U_1 and U_2 have same SIDs.

3 Model Checking and Verification

After specifying security protocols, adversary actions, and attacks using lanuage $\mathcal{L}_{sp}$, we merge three parts into a logic program $\mathcal{P}$ in which we add a constraint rule,

$$\leftarrow not\ attack.$$

We use *Smodels* system [11] to verify security protocols as follows: (1) through *lparse*, we obtain a finite ground logic program $\mathcal{P}^g$ from program $\mathcal{P}$; (2) Using *smodels*, we compute stable models of ground program $\mathcal{P}^g$; (3) If no stable model exists, the attack does not exist for protocol runs up to time t_{max}[1]; (4) If there is a stable model, we collect atoms representing actions, $sends$, $gets$ and $intercept$ that are true in the model, from which we can find the sequence of actions that is an attack trace.

1. At time t_0, initiator u_1 broadcasts an initial message which has three parts: the set of principals in the protocol run, $agset(u_1, u_2)$; the signature of the principal set and encrypted random nonce $n(0)$ under the public key of principal u_2, $sign(sig\_sKey(u_1), agset(u_1, u_2)||enc(pKey(u2), n(0)))$; the encryption of the encrypted random nonce $n(0)$, $enc(pKey(u2), n(0))$.
2. At time t_1, $\mathcal{A}$ receives the message and intercepts it. After modifying the principal set to $agset(a, u_2)$ and make a new signature using his own signature key, $sign(sig\_sKey(a), agset(a, u_2)||enc(pKey(u_2), n(0)))$, $\mathcal{A}$ fabricates a new message, and sends it to principal u_2. Now $\mathcal{A}$ acts as an initiator and start a different session.
3. At time t_2, principal u_2 receives the message from $\mathcal{A}$ and believes that $\mathcal{A}$ initiates a protocol run.
4. At time t_3, principal u_2 broadcasts his identifier and random number.
5. At time t_4, principal u_1 and $\mathcal{A}$ receive the random nonce of principal u_2. u_1 believes he finishes his own session with u_2, however u_2 believes he is in a different session with $\mathcal{A}$.

An attack was found in 5.460 seconds. We observe that principal u_1's SID is $(0, 8)$ and principal u_2's SID is $(11, 8)$. Then u_1 and u_2 are not partners since they do not have matching SIDs. u_1 believes the session key $SK_{u_1} = h(n(0)||n(8))$ is being shared with u_2, but u_2 believes the session key $SK_{u_2} = h(n(0)||n(8)) = SK_{u_1}$ is being shared with $\mathcal{A}$. Although $\mathcal{A}$ does not know the session key as $\mathcal{A}$ does not know the value of $n(0)$, he is able to send query $Reveal$ to the session with u_2 and get $SK_{u_2} = h(n(0)||n(8))$ which is same as SK_{u_1}. Our case study protocol is not secure under Bellare-Rogaway model as being claimed.

[1] t_{max} is a max time limitation set up in the logic program.

4 Conclusions

In this paper, we developed a logic programming framework in which we not only use formal verification under adversary models in the computational complexity theory, but also integrate protocol analysis into the approach. As logic programming is a declarative executable approach for knowledge representation and reasoning, in our framework, we defined a security protocol specification language $\mathcal{L}_{sp}$ under logic programming with stable model semantics which is used to specify security protocols carrying claimed security proof under adversary models. Using $Smodels$ we are able to verify the program we have modelled. As a case study, Boyd-González Nieto conference key agreement protocol has been specified, verified using our framework.

References

1. M. Abdalla, O. Chevassut, P. Fouque, and D. Pointcheval. A Simple Threshold Authenticated key Exchange from Short Secrets. In *Advances in Cryptology - Asiacrypt 2005*, pages 566-584. Springer-Verlag.
2. M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computaiotnal Soundness of Formal Encryption). In *Journal of Cryptology*, 15(2), pages 103-127, 2002.
3. L. C. Aiello and F. Massacci. Verifying Security Protocols as Planning in Logic Programming. *ACM Transactions on Computational Logic* 2(4)(2001), pages 542-580.
4. M. Backes and C. Jacobi. Cryptographically Sound and Machine-Assisted Verification of Security Protocols. In *STACS 2003*, volume 2607/2003, LNCS, pages 310-329. Springer 2003.
5. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
6. C. Boyd and J. M. González Nieto. Round-optimal Contributory Conference Key Agreement. In *PKC 2003*, volume 2567/2003 of LNCS, pages 161-174. Spring,2003.
7. M. Bellare and D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology - Eurocrypt 2000*, pages 139-155. Springer-Verlag, 2000.
8. K. R. Choo. Refuting Security Proofs for Tipartite Key Exchange with Model Checker in Planning Problem Setting. In *the 19th IEEE Computer Security Foundations Workshop - CSFW 2006*, pages 297-308.
9. I. R. Jeong, J. Katz, and D. H. Lee. One-Round Protocols for Two-Party Authenticated Key Exchange. In *ACNS 2004*, volume 3089/2004 of LNCS, pages 220-232. Springer, 2004.
10. G. Lowe. Some New Attacks upon Security Protocols. In *Proceedings of the 9th IEEE Computer Security Foundatons Workshop(CSFW'96)*. pages 162-169. IEEE Computer Society Press.
11. I. Niemela, P. Simons and T. Syrjanen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-monotonic Reasoning 2000*.
12. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. In Journal of Computer Security 1998, 6, pages 85-128.
13. P. Ryan and S. Schneider. An Attack on a Recursive Authentication Protocol: A Cautionary Tale. In Information Processing Letters 1998. 65, 15, pages 7-16.