Verifying Mutual Authentication for the DLK Protocol using ProVerif tool

H. M. N. Al Hamadi, C. Y. Yeun, M. J. Zemerly, M. A. Al-Qutayri, A. Gawanmeh
Electrical and Computer Engineering Department, Khalifa University of Science, Technology and Research, P.O. Box 573, Sharjah, UAE

Abstract

This paper adopts the Distributed Lightweight Kerberos (DLK) protocol, which is a result of enhancing the well-known Kerberos protocol. One of the advantages of the DLK protocol is that it addresses mutual authentication and confidentiality challenges while reducing the required number of messages to securely communicate with multiple service providers. In this paper we formally analyze and verify the DLK protocol that has been designed for multi-agent based systems. We use the ProVerif formal model checker in order to model and verify the DLK protocol. Using ProVerif exposed essential security problems in the DLK protocol as non-formal analysis had shown. ProVerif enabled us to detect that mutual authentication is compromised between the DLK participants. We propose a nonce-based authentication technique in order to redesign the protocol and fix this flaw. We then verified the correctness of the proposed protocol using the ProVerif tool.

Keywords─ Formal Method; ProVerif; DLK protocol; Agent technology; Mutual authentication; Confidentiality; Integrity; Authorization

1. Introduction

Kerberors is a popular network authentication protocol. It provides a strong level of mutual authentication for a variety of client-server applications by using secret key cryptography, such as AES [1]. Recently, we have proposed a new methodology for an enhanced Kerberos protocol that provides secure distribution for the requests in multi-agent systems [2]. The Distributed Lightweight Kerberos (DLK) is a secure distributed protocol for multi-agent systems. It improves the security and efficiency aspects of the Kerberos protocol. Informal analysis and simulation based methods are usually used for testing and verification of protocols. Unfortunately, it is impossible to cover all possible protocol scenarios given the unbounded number of protocol runs, large number of participants, and the possible messages space. Simulation based techniques can be useful under the assumption that the protocol has a finite number of runs, and limited number of participants. In addition, analytical informal techniques are exhaustive, in particular, when considering modifications to the protocol. However, formal methods can be efficiently used to analyze and verify the correct design of these protocols. Complete verification will lead to trusted secure protocols that can be used with high degree of confidence.

ProVerif [3] is a model checking fully automatic tool for formally analyzing security protocols using Horn theory [4]. ProVerif models attacks that an attacker can perform based on multiple concurrent executions of the protocol. The tool imposes no limit on the number of concurrent protocol runs that an attacker may execute. Therefore, it is able to find attacks that require any number of concurrent protocol runs. The tool has been used successfully to model and analyze several security protocols, such as, the well-known mutual authentication protocol "Needham-Schroeder" which was tested by the ProVerif tool, and the tool provided a potential trace for the intruder to use [5]. ProVerif can verify the security of the protocol regardless of the participants' numbers, size of the generated messages and number of channels between the participants. In ProVerif, the intruder has control over the system by actively monitoring communication channels. In this case the intruder can capture, modify, compose, send, or resend messages. ProVerif provides a trace of the intruder's attack if the protocol has a security problem. It also provides security properties verification for mutual authentication and confidentiality. However, dealing with some security parameters such as timestamp is still a problematic issue and ProVerif cannot model the variable values.

Even though Kerberos protocol has received extensive analysis and it is believed to be correct, any extension or modification of the protocol can introduce errors that are hard to capture. In addition, it is necessary to ensure that the new modified

protocol can still provide the required services it is designed for. In this paper, we conducted security analysis of the DLK protocol. For security formal analysis, we use ProVerif tool in order to verify the DLK protocol and hence, ensure that the DLK protocol provides the mutual authentication and confidentiality security properties correctly. First we provide a formal model for the DLK protocol and for secrecy and mutual authentication security properties. Then, we used ProVerif to prove that the DLK protocol achieves mutual authentication and preserves data integrity, and hence, provides the proper security requirements.

The rest of the paper is organized as follows: Section 2 provides an overview of related work. Section 3 discusses the basic preliminary background of the paper. In Section 4, we give the formal model of DLK protocol and security requirements. We then provide formal analysis and highlight an authentication flaw in DLK. In Section 5, we provide an enhanced version of the DLK protocol to correct the mutual authentication flaw that was identified. Section 6 provides security analysis of the DLK protocol. Finally, Section 7 concludes the paper and highlights some future work.

2. Related Work

Hardware and software systems have grown in functionality and complexity. This growth decreases the probability of detecting vague errors that may lead to serious consequences, such as loss of money, information, time or even human life [6]. Therefore, constructing systems that operate reliably is one of the main objectives of using formal methods. Formal methods are mathematically-based techniques that help in designing, implementing and verifying hardware and software systems.

ProVerif is an automatic analysis tool that is similar to CryptoVerif [7] and FS2PV [8]. These tools use several formal methods in order to cover various security requirements. For example, confidentiality is covered with the CSP method [9], while authentication is addressed by Petri nets and a combination of the CSP and B methods [10], and authorization is dealt with using OrBAC method [11].

In a previous work [2], we have proposed a security protocol that is based on the Kerberos protocol and claimed that it provides the distribution nature for the user requests and covers several security requirements including authentication, confidentiality, integrity and authorization. The proposed protocol was designed to address the security vulnerabilities of multi-agent systems. However, the security analysis of the DLK protocol was theoretical and based on the original Kerberos protocol. In this paper we apply formal methods, particularly using ProVerif tool, to assess security robustness of DLK.

3. Preliminaries

3.1. DLK Protocol

Based on the distributed nature of the agent technology and the well-known Kerberos protocol [12] [13], we proposed the DLK security protocol that inherits the security strength of Kerberos and applies it in a distributed manner. The main block diagram of the DLK protocol is depicted in Figure 1. It shows the communication steps between the participants.

As in Kerberos, the AS Agent is the Authentication Server and the TGS Agent is the Ticket Granting Server. The SP Agent represents the Service Provider, while ***n*** is the number of service providers contacted by the TGS agent under specific conditions, such as when the SP agents are physically nearest to the client agent. In Kerberos, a client is involved in six messages, although it only wants to access one service provider. Therefore, one of the objectives of the DLK protocol is to reduce the amount of messages and provide the same security level of the Kerberos protocol [2]. In addition to that, the client has the option to interact with several SP Agents in the network based on its request. However, the DLK protocol is more complex than Kerberos due to an increase in the number of interactions to 4 instead of 3. As shown in Table 1, the TGS agent interacts with the SP agent which is not the case in the Kerberos protocol.

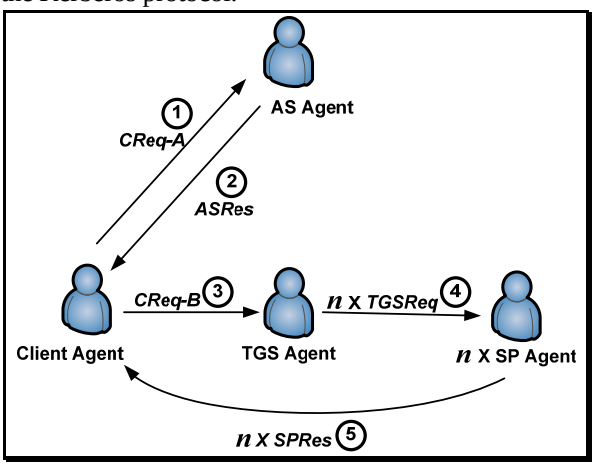

Figure 1. Block Diagram of the DLK Protocol

Table 1. DLK and Kerberos Comparison stages

Stages of Kerberos Protocol	Stages of DLK Protocol
Client ←→ AS Client ←→ TGS Client ←→ SP	Client ←→ AS Client → TGS TGS → SP SP ←→ Client

The numbers from (1) to (5) in Figure 1 refer to sequential events based on their execution. *CReq-A*, *CReq-B* and *TGSReq* denote two client requests and one TGS request, respectively. *ASRes* and *SPRes*

denote AS response and SP response respectively. In brief, if a client needs to invoke results from the service providers securely, only the username which identifies the client ID is sent to the AS agent in order to provide its identity to the system and prevent any flow of passwords on the network. Then, session keys are generated by the AS and TGS agents in order to guarantee the security of the channels between clients and SP agents. In the DLK protocol, the TGS agent distributes a number of encrypted requests to the SP agents on behalf of the client. Then, the received SP agents respond to the client agent directly without returning back to the TGS agent.

Table 2. DLK protocol "Table of Abbreviations"

AS	= Authentication Service Agent
TGS	= Ticket Granting Service Agent
SP	= Service Provider Agent
Cid	= Client's ID
SPid	= Service Provider's ID
SID	= Service ID
Req	= Client's Request
Res	= SP's results
K_X	= Private Key for X (Secret Key)
$K_{X,Y}$	= Session Key between X and Y
Add_C	= Client's Network Address
VP_T	= Ticket's Validation Period
t_X	= Timestamp labeled by X
$U(t_X)$	= Updated timestamp X
$\{M\}K_X$	= Message M encrypted by X's private key
$\{M\}K_{X,Y}$	= Message M encrypted by asession key between X and Y
$\|\|$	= Concatenation

Using the abbreviations in Table 2 and the stages in Figure 1, the details of the DLK protocol process are presented below:

(1) *CReq-A* : [Username]
(2) *ASRes* : $\{K_{C,TGS}\}K_C \| \{T1\}K_{TGS}$
(3) *CReq-B* : $\{T1\}K_{TGS} \| \{Cid, t_1, SID, Req\}K_{C,TGS}$
(4) *TGSReq* : $\{Cid, t_2, SID, K_{SP,C}, Req\}K_{SP} \| \{T2\}K_{C,TGS}$
(5) *SPRes* : $\{SPid, U(t_2), Res\}K_{SP,C} \| \{T2\}K_{C,TGS}$

Where $T1 = [Cid, Add_C, K_{C,TGS}, VP_T]$
$T2 = [SPid, Add_{SP}, K_{SP,C}, VP_T]$

3.2. Applied Pi-Calculus

Applied pi calculus is a language introduced in [14] for modeling distributed systems and their interactions. It is an extension of pure pi calculus which is useful for modeling cryptography protocols. Applied pi calculus shifts the concentration in pure pi calculus to encodings and treats all data as special types of processes, not as just names [14]. The syntax, semantics and operational semantics used in applied Pi-Calculus are described below.

1) Syntax and semantics: The main difference between applied pi calculus and pure pi calculus is that in applied pi calculus, processes are built from **terms**. These terms consist of an infinite set of names donated by *a,b,c,…*, an infinite set of variables donated by x,y,z,…, and a finite set of functional symbols or applications for processing terms (e.g. Encryption, Decryption, Hash) donated by *enc(), dec(), h()* which could be a constructor or destructor application. The set of terms in appied pi calculus are given below:

P, Q, R::=	Terms
a, b, c,…	name
x, y, z,…	variable
enc$(M_1,\ldots,M_n)$	function application

Where ***n*** is the number of terms inside a function, the grammar for the processes is defined as follows. Note that we define only processes related to our DLK protocol. M and N are terms, C is a condition, *a* is a name, *x* is a variable and *t* is a term type.

P, Q, R::=	Processes
0	null process
P|*Q*	parallel process
!*P*	replication of a process
new *a*:*t*; *P*	name restriction
let *x*=M in *P* else *Q*	term evaluation
if *C* then *P* else *Q*	conditional
event (N)	non-injective event
in(M, *x:t*); *P*	message input
out(M,N); *P*	message output

The meaning of *C* is to have a conditional expression in order to get a true or false output and helps in the operational semantics. The following describes the conditional statements using the terms *M*, *N* and *C*.

C ::=	Conditions
M = *N*	term equality
M <> *N*	term inquality
M	term (of type bool)
C && *C*	conjunction
C || *C*	disconjunction
not(*C*)	negation

2) Operational semantics: There are two relations between processes: structural equivalence and internal reductions. Structural equivalence (≡) can be simply seen in the following:

$$A \mid 0 \equiv A$$
$$A \mid (B \mid C) \equiv (A \mid B) \mid C$$
$$A \mid B \equiv B \mid A$$
$$!P \equiv P \mid !P$$

The internal reduction (→) is a relation closed under the structural equivalence, and can be described in the following example:

COMM: out(M,N); *P*| in(M, *x:t*); *Q* → P | Q
THEN: if *M=N* then *P* else *Q* → P
ELSE: if *M=N* then *P* else *Q* → *Q; where M≠N*

3.3. ProVerif

The automatic analyzer, ProVerif is used for verifying security properties for cryptographic protocols using a specification language that is based on an extension of pure Pi-calculus. ProVerif can prove security properties, correspondence and observational equivalence. This proofing capability helps the information security realm to analyze the secrecy and authentication of security protocols that are permitted from the provider. ProVerif is a tool that supports cryptographic primitives, including encryption and decryption (symmetric and asymmetric), digital signatures, and hash function. In addition, other primitives such as rewriting rules and equations can be modeled using terms [15].

In ProVerif, the protocol is converted into a set of Horn clauses [4], and the security properties are translated through queries on these clauses. The adversary is an agent in ProVerif that tries to reach the inquired data from the channels between processes. As an example, the syntax ***query*** *attacker* ***(****Data****)*** is used to let the adversary try to learn the value of *Data* by eavesdropping channels. Otherwise, ProVerif proves the secrecy properties of *Data*. It has the functionality to make the attacker ignore specific types of information by using the syntax, ***not*** *attack (Data).* Authentication is based on a sequence of specific events called correspondence assertions of the form "if some event (such as event #2) has been executed, then other events (such as event #1) have been executed". In other words, if it is not possible to execute event #2 before having event #1 executed, then authentication is achieved. Such a query is written as: ***query event*** *(2)* ==> ***event*** *(1)*. The truth table for this operation is shown in Table 3.

Table 3. Truth table of events

event (1)	event (2)	event (2) ==> event (1)
0	0	1
0	1	0
1	0	1
1	1	1

When ProVerif verifies a security property for a cryptographic protocol, the results guarantee the following as shown in Figure 2:

- When the proof is true, the proof is not related to the number of sessions of the protocol and the used message size. It also tests all the possibilities for an attacker to break the desired property.
- When the proof fails, ProVerif provides an attack trace which is reconstructed from a derivation of facts obtained from the clauses. However, ProVerif does not provide a solution to overcome the failure. Therefore, when such a situation appears, we study the derivation provided by ProVerif, and construct a solution to avoid this failure. The failure of proof always corresponds to an attack. A detailed discussion about ProVerif is available in [5].

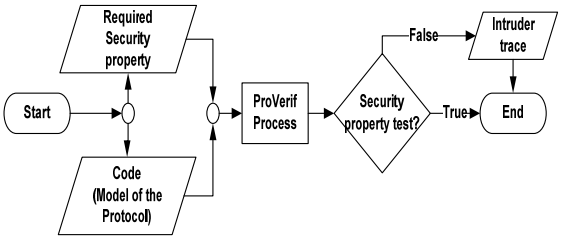

Figure 2. ProVerif block diagram

4. Verifying the DLK Protocol using ProVerif

This section describes the coding of the DLK protocol using ProVerif syntax, the output from running this code through the tool, and an analysis of the attack trace for the failure queries.

4.1. DLK Coding in the Applied Pi Calculus

Coding the DLK protocol starts by disposing of unnecessary parts, such as Addresses (Add_X), Validation period (VPt) and Service ID (SID), and keeping the rest of the protocol parts. Then, we determine the required cryptographic primitives with function symbols, and rewrite rules and equations over terms. For instance, the term *enc(M, K)* denotes the encryption function for encrypting the message *M* with the key *K* (which is a symmetric key in our protocol) and the result will be *Cipher*, which denotes the cipher text, as shown below.

$$enc(M, K) \rightarrow Cipher$$

On the other hand, we use another term, *dec(Cihper, K),* to represent the reverse function of the term *enc(M, K)*. By this property, we rewrite the relationship rule between these two functions that simplifies conditions for the processes. Such a relationship is written in the following form.

$$dec(enc(M, K), K) \rightarrow M$$

We define processes that model the task of client, AS, TGS and SP agents; these processes are specified to operate in parallel. The processes are communicated by a public channel ***c***, as the adversary has control over this channel. The adversary cannot fetch any information from a process unless this data has been transferred through the channel. The declaration of the channel is done by the syntax.

free *c*: channel

The DLK protocol as presented in subsection 3.1 is expected to satisfy (informally) the following:

- Authentication of C to SP: if the SP identifies C, it responds so that at the end of the protocol C

has approval to engage with SP in a session, only if SP permits it.
- Authentication of SP to C: similar to the above.
- Authentication of C to TGS: TGS makes sure that C has a valid ticket to use its service. This ticket guarantees the identity of C.
- Secrecy of session keys, requests and results: DLK protocol should provide the privacy of important information using appropriate functions such as *enc*() based on symmetric key algorithms.

In our model, we assume that secret keys K_C, K_T and K_S for the clients, TGS and SP respectively. The keys have private values and the adversary has no intention to recover them. Otherwise, the adversary would enter into an infinite loop of rules in order to find these values. This can be represented by the following syntax:

free Kc, Kt, Ks: key [private].

Our interest in this model is to verify the secrecy of the used session keys, requests and results. In order to challenge the adversary, we write the *query* syntax, as the following:

1. **query** attacker(*req*).
2. **query** attacker(*res*).
3. **query** attacker(*CTkey*).
4. **query** attacker(*CSkey*).

Where *req* is a request, and *res* denotes a result. *CTkey* is the session key between the TGS agent and the client agent, while *CSkey* denotes the SP - client session key.

The mutual authentication between the client agent and the SP agent is verified using the events technique as in Table 3. For example, if we have two participants, Alice and Bob, there are four events in order to have mutual authentication between them, as shown in Figure 3. Event 1 represents the request from Alice to create a trusted session with Bob. Event 2 shows the acceptance from Bob to complete the trusting process. Event 3 presents the response from Bob which is followed by Event 4 wherein Alice is allowed to start a trusted session.

In our model, we declare the events:

- BeginCS(), which is used by the client agent to record the status that the session key CTkey and timestamp t1 are available to proceed with the authentication.
- BeginSC(), which means that the SP agent denotes the intention to initiate authentication by successfully receiving the identity of the client agent.
- EndSC(), the SP agent believes it has completed its task and correctly updated the timestamp. This event is executed only when the SP agent has results for the request.
- EndCS(), which records the client agent's belief that it has the SP agent identity and the correct update of the timestamp.

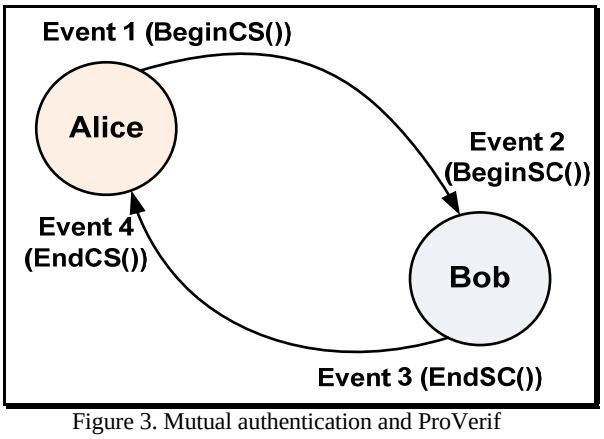

Figure 3. Mutual authentication and ProVerif

We test mutual authentication by making sure that *event(BeginSC())* cannot be executed before the execution of *event(BeginCS())*, and also that *event(EndCS())* cannot be executed before *event(EndSC())*. Moreover, *event(EndCS())* cannot be executed unless *event(BeginCS())* and *event(BeginSC())* are executed, as shown in Figure 3. This is written as follows:

1. **query** event(*BeginSC()*)==> event(*BeginCS()*).
2. **query** event(*EndCS()*) ==> event(*EndSC()*).
3. **query** event(*EndCS()*)==>event(*BeginCS()*)&& event(*BeginSC()*).

The coding and description of the 4 processes in the DLK protocol, Client, AS, TGS and SP agents, will be illustrated. These processes are executed multiple times in parallel using the following syntax:

1. **process**
2. **(**
   (!processC)|(!processAS)|(!processT)|(!processSP)
3. **)**

First, the client agent code has the owner username, or any identity that helps the AS agent to find the stored private key K_C for the client. We refer to the identity by C. The execution of the event in line #2 declares that it has the identity C. Line #4 explains the syntax of sending a message content C through channel ***c***. Then, the client agent receives a message of two parts, a new session key K_{CT} and a ticket *m2*, from the AS agent. The timestamp *t1* is taken in line #12 and sent with the request *req,* where they are encrypted using K_{CT}; in line #13.

```
1.  let processC=
2.      event BeginCA(C);
3.  (*  Message 1          *)
4.      out(c,(C));
5.  (*  Message 2          *)
6.      in(c,(m1:bitstring,m2:bitstring));
7.      let (CTkey:key)=dec(m1,Kc) in
8.      event EndCA(CTkey);
9.  (*  Message 3          *)
```

```
10.      new t1:timest;
11.      event BeginCT(T, CTkey);
12.      event BeginCS(CTkey, t1);
13.      out(c,(enc((C,t1,req),CTkey),m2));
14.  (*  Message 5          *)
15.      in (c,(m3:bitstring,m4:bitstring));
16.      let(SCkey:key,hostY:server,t11:timest)=
         decr(m4, CTkey) in
17.      event EndCT(hostY,C);
18.      let(=update(t1),res:bitstring,=hostY)=
         decr(m3, SCkey)in
19.      event EndCS(hostY,C,update(t1)).
```

Then, message 5 is received in line #15. In line 16, the client agent uses K_{CT} to decrypt the first part of the message and obtain a new session key K_{SC}, which is used to decrypt the second part of the message. On the same line of the decryption operation, line #18, there are two types of conditional logic; one to verify the matching of SP ID between both parts (*m3* and *m4*), and another to verify the updated timestamp. Line #19 executes the last event for the authentication between the client agent and the SP agent.

For the AS agent, we assume that it is dedicated only for serving the client ID, **C**, otherwise, it remains idle. This condition is specified in lines #23 and #31. Line #22 shows the received identity through ***c*** channel and line #25 declares a new session key K_{CT} between the client agent and the TGS agent. In addition, the first ticket in the DLK protocol is encrypted in line #26. In line #27, we created another function of encryption *encr()* for the ProVerif tool to encrypt the CTkey variable of type key, while the client agent can only decrypt it using the *dec()* function in line #7. The K_{CT} and ticket 1 are sent to the client agent in line #30. Line #32 represents the idle state of the AS agent in case it does not receive the dedicated identity.

```
20.  let processAS=
21.  (*  Message 1          *)
22.      in(c,(hostC:client));
23.      If hostC=C then
24.      event BeginAC(hostC);
25.      new CTkey:key;
26.      let m1=enc((hostC,CTkey),Kt) in
27.      let m2=encr((CTkey),Kc) in
28.      event EndAC(CTkey,hostC);
29.  (*  Message 2          *)
30.      out (c, (m1,m2))
31.      else
32.      0.
```

On the other hand, we do not have any assumption for the TGS agent process except that it does know which SP ID to contact. A message received from the client agent has ticket 1 (*m2)* which has the session key (K_{CT}). The message *m1* is decrypted using that session key. Then, the TGS agent creates the ticket 2 (*m3*), which has the new session key K_{CS}, and encrypted by K_{CT} (line #41). In addition, it constructs the request (line #42) on behalf of the client agent and sends both ticket 2 and the request through the channel ***c*** (line #45).

```
33.  let processT=
34.  (*  Message 3          *)
35.      in(c,(m1:bitstring,m2:bitstring));
36.      let(hostC:client,CTkey:key)=
         decr(m2,Kt) in
37.      let(=hostC,t3:timest,r:bitstring)=
         decr(m1,CTkey) in
38.      event BeginTC(hostC);
39.      new t12:timest;
40.      new CSkey:key;
41.      let m3=enc((CSkey,S,t12),CTkey) in
42.      let m4=enc((hostC,CSkey,t3,r),Ks) in
43.      event EndTC(S,hostC,CTkey,t3);
44.  (*  Message 4          *)
45.      out (c,(m3,m4)).
```

Finally, the last process is the SP agent. It waits for a message from the TGS agent to be received as in line #48. The process focuses on updating the timestamp of the client agent (*update(t34)*) and finding the result for its request (*result(r12)*). Both goals are encrypted using the received session key K_{CS} (line #49). Then, the SP agent sends the encrypted result along with the ticket 2 (*m1*) (line #52),

```
46.  let processSP=
47.  (*  Message 4          *)
48.      in( c, (m1:bitstring,m2:bitstring));
49.      let(host:client,CSkey:key,t34:timest,r12:
         bitstring) = decr(m2,Ks) in
50.      event BeginSC(host);
51.  (*  Message 5          *)
52.      out(c,(enc((update(t34),result(r12),S),
         CSkey),m1);
53.      event EndSC(S,host,result(r12),
         SCkey,update(t34)).
```

The results from running the complete previous code (lines #1 through #53) are illustrated in the next subsection.

4.2. ProVerif Analysis for Attack Traces

In order to automatically prove queries, Proverif translates the processes and adversary actions into a set of Horn clauses. Then, it runs the processes and searches for a valid security gap based on requested queries. In particular, the secrecy property, *i.e.* ProVerif, verifies if the adversary is able to reach an encrypted term from the messages exchanged through a public channel. In this section, we describe ProVerif output of DLK for the validation of authentication between the client agent, the SP agent, and the secrecy of the session keys, as well as request and result. ProVerif displays the output in the following sequence; first, prints out the internal representation, then, it handles each query sequentially.

Firstly, when the secrecy assumption is given about the DLK protocol, our interest is to have ProVerif tool checking it by stepping through all the possibilities to verify this assumption. ProVerif adversary impersonates an intruder *I* and engages with all protocol participants through the public channel **c**. When we ran our protocol in the ProVerif tool, it gave the output illustrated in Table 4.

As shown in Table 4, ProVerif verifies the secrecy of session keys K_{CS} and K_{TC}. In addition it verifies the secrecy of the request and its result.

Table 4. Results of confidentiality test on the DLK

Parameter	Initial status	ProVerif Output
K_C, K_T, K_S	Secure	Secure
Cid, SPid	Unsecure	No output
ASid, TGSid	Unsecure	No output
CTkey	Unknown	Secure
SCkey	Unknown	Secure
Req	Unknown	Secure
Res	Unknown	Secure
t1	Unknown	Secure
U(t1)	Unknown	Secure

However, the result of verifying the authentication between the client and SP agents comes with the possibility of a Man-In-The-Middle attack incidence. This incidence dropped the validation for a part of the mutual authentication rule. ProVerif proves that an Intruder with sequence of actions can execute the *event(EndCS())* before the *event(EndSC())* is executed.

query event(*EndCS()*) ==> event(*EndSC()*).

The following narration summarizes the sequence required for breaching the authentication property of DLK that the attack trace found by the ProVerif tool (The abbreviations in Table 2 are used):

(1) C → I : [C]
(2) I → AS : [C]
(3) AS → I : $\{K_{C,TGS}\}K_C \| \{T1\}K_{TGS}$
(4) I → C : $\mathbf{\{K_{C,TGS}\}K_C} \| X$
(5) C → I : $X \| \{Cid, t_1, SID, Req\}K_{C,TGS}$
(6) I → TGS : $\{T1\}K_{TGS} \| \{Cid, t_1, SID, Req\}K_{C,TGS}$
(7) TGS → I : $\{Cid, t_2, SID, K_{SP,C}, Req\}K_{SP} \| \mathbf{\{T2\}K_{C,TGS}}$
(8) I → SP : $\{Cid, t_2, SID, K_{SP,C}, Req\}K_{SP} \| Y$
(9) SP → I : $\mathbf{\{SPid, U(t_2), Res\}K_{SP,C}} \| Y$
(10) I → C : $\{K_{C,TGS}\}K_C \| X$
(11) I → C : $\{SPid, U(t_2), Res\}K_{SP,C} \| \{T2\}K_{C,TGS}$

Where T1 = [Cid, Add_C, $K_{C,TGS}$, VP_T]
T2 = [SPid, Add_{SP}, $K_{SP,C}$, VP_T]
X, Y = Random Values

The narration is explained as follows:

- In steps (1-3), the intruder interfered and received the client ID C. Then, he sent it to the AS agent. The AS agent responded, then, the intruder received the response.
- In step (4), the intruder used the inverse of function 2-tuple to separate the first part from the received message; $\mathbf{\{K_{C,TGS}\}K_C}$. Then, the intruder used the function 2-tuple to combine this part with random value X and send it back to the client agent.
- In steps (5-7), the client agent sent the request encrypted by the session key $K_{C,TGS}$. The intruder received the request and combined it with the second part from step (4); $\{T1\}K_{TGS}$. Then, he sent it to the TGS agent. The intruder received the response from the TGS agent in step (7). Then he used the inverse of function 2-tuple to separate the second part from the received message; $\mathbf{\{T2\}K_{C,TGS}}$.
- In steps (8-9), the intruder took the first part from the separation process in step (7) and combined it with random value Y. Then, he sent it to the SP agent. The intruder interfered with the response message from the SP agent and used the inverse of 2-tuple to obtain the first part; $\mathbf{\{SPid, U(t_2), Res\}K_{SP,C}}$.
- In steps (10-11), the intruder sent the same message in step (4) to execute the event *BeginCS()*. Then, the intruder sent a combined message from the first part in step (9) and the second part in step (7). Thus, the client agent successfully decrypted the received message and executed the event *BeginCS()* before event *EndSC()* is executed.

It must be noted that ProVerif verifies the validation of the other cases as shown in Table 5.

Table 5. Results of authentication test on the DLK

The Quiry	ProVerif output
event(*BeginTC()*)==> event(*BeginCT()*)	**True**
event(*EndCT()*) ==> event(*EndTC()*)	**True**
event(*EndCT()*)==>event(*BeginCT()*)&&event(*BeginTC()*)	**True**
event(*BeginSC()*)==> event(*BeginCS()*)	**True**
event(*EndCS()*) ==> event(*EndSC()*)	**False**
event(*EndCS()*)==>event(*BeginCS()*)&&event(*BeginSC()*)	**True**

We applied automatic verification on the DLK protocol, which helped capture an error in the protocol that would have been hard to discover using normal paper and pencil analysis. To achieve this, a formal model of the protocol was established first. Then, the authentication and secrecy properties were formally defined. Finally, ProVerif tool was used to verify that the protocol achieves mutual authentication. In fact, the tool helped to discover and correct a hidden error in our proposed protocol and provided the opportunity to correct it and verify the DLK protocol again. The next section discusses the proposed solution for the discovered error and proofs of the validation of this solution in order to provide mutual authentication between the client agent and the SP agent.

5. Enhanced Mutual Authentication in DLK Protocol

According to mutual authentication definition, the basic issue in the original DLK protocol is when the event *EndCS()* is executed before event *EndSC()*. In

order to provide mutual authentication between the client agent and the SP agent, we focus on one of the main notions and ideas in authentication protocols; challenge-response. Authentication protocols based on challenge-response require a question (challenge) from one party and a right answer (response) from another in order to be authenticated. The challenge-response technique could be based on some security parameters such as timestamp or nonce. In our case we opted for nonce.

5.1. Nonce-based Authentication

Nonce stands for a number used once. It is usually a random number used for authentication process in order that the message cannot be reused and its freshness is guaranteed, thus avoiding the replay attack. Nonce-based authentication can be adopted using both symmetric and asymmetric key cryptography [16].

Assuming Alice wants to authenticate with Bob using the nonce and the PKI cryptography. Here, the public key of Alice is denoted by $K_{Pub(A)}$, where the private key is denoted by the $K_{Priv(A)}$, and the same for Bob. Alice sends the nonce *n1* as a plaintext to Bob and receives *n1* along with her identity encrypted using the private key of Bob $K_{Priv(B)}$. Since the private key of Bob is secured and only known by Bob, thus proving the identity of the responder as shown below:

Alice → Bob: [*n1*]
Bob → Alice: {*A, n1*}$K_{Priv(B)}$

Alice decrypts the ciphertext using the public key of Bob $K_{Pub(B)}$ and verifies that the nonce *n1* is the same. The availability of the identity A in the ciphertext is to prove that Bob identifies Alice. The same impact can be achieved using a symmetric key cryptography. Instead of public and private keys for each entity, a shared key is between two entities (as Alice and Bob) denoted as K_{AB}. The scenario is slightly different than asymmetric key in order to protect the nonce *n1*. Here, Alice sends the nonce *n1* to Bob; however, she would send it encrypted using the shared key K_{AB} rather than a plaintext as the following:

Alice → Bob: {*A, n1*}K_{AB}
Bob → Alice: [*n1*]

This protocol authenticates Bob receiving nonce *n1* from Alice, since the shared key K_{AB} is only known by Alice and Bob. The previous two scenarios explain two categories of nonce-based protocol. These categories depend on when the nonce is encrypted and when it is sent as a plaintext; the first scenario is categorized as a plain-cipher, and the second is a cipher-plain method.

In the third category, the nonce is sent and received encrypted. This method can be applied in both symmetric and asymmetric key cryptography. In addition, it is useful in exchanging messages. In case Alice and Bob are using asymmetric key cryptography, both of them prove their identities using their public keys $K_{Pub(A)}$ and $K_{Pub(B)}$, respectively. As shown below:

Alice → Bob: {*A, n1*}$K_{Pub(A)}$
Bob → Alice: {*B, n1*}$K_{Pub(B)}$

On the other hand, Alice and Bob could use the symmetric key cryptography in order to exchange the nonce *n1* securely. As shown in the following:

Alice → Bob: {*A, n1*}K_{AB}
Bob → Alice: {*B, n1*}K_{AB}

Both ways in the third category prove the mutual authentication between Alice and Bob, without compromising the knowledge of the nonce *n1* by an intruder in the middle. Based on our proposed DLK protocol and narration summary of the attack trace found using ProVerif tool, the third category is used to solve our authentication issue between the client agent and the SP agent.

5.2. Redesign of the DLK protocol

The DLK protocol is based on symmetric key cryptography that implies the use of the third category of nonce-based authentication to solve the issue raised by ProVerif tool. The narration given by ProVerif proves the weakness of the generated ticket by the TGS agent in order to authenticate the SP agents with their result and message freshness. Therefore, our solution considers the nonce in the request message from the client agent to the TGS agent (*CReq-B*), in order that the TGS agent can integrate this nonce in the ticket T2 for the SP agent. Using the abbreviations in Table 2 and the stages in Figure 1, the details of the DLK protocol process are presented below:

(1) *CReq-A* : [Username]
(2) *ASRes* : $\{K_{C,TGS}\}K_C \| \{T1\}K_{TGS}$
(3) *CReq-B* : $\{T1\}K_{TGS} \| \{Cid, t_1, n1, SID, Req\}K_{C,TGS}$
(4) *TGSReq* : $\{Cid, t_2, SID, K_{SP,C}, Req\}K_{SP} \| \{T2\}K_{C,TGS}$
(5) *SPRes* : $\{SPid, U(t_2), Res\}K_{SP,C} \| \{T2\}K_{C,TGS}$

Where T1 = [Cid, Add_C, $K_{C,TGS,}$ VP_T]
T2 = [SPid, Add_{SP}, *n1*, $K_{SP,C}$, VP_T]

Based on the mutual authentication definition previously given, the nonce is exchanged in the protocol in order to execute the events in the correct sequence. More precisely, with that change, the intruder cannot apply a replay attack and performs a Man-In-The-Middle attack on the protocol.

After combining the nonce-based technique with the DLK protocol and illustrating the events sequence, verifying the protocol using the ProVerif tool or any other tool is a must in order to validate our solution.

5.3. Verifing the corrected DLK protocol using ProVerif tool

The overall ProVerif code for the DLK protocol remains the same except for declaring the variable nonce using the following syntax:

type nonce**.**

The nonce *n1* is initiated and sent by the client agent as a challenge and the client agent receives this nonce as a response. From the time of initiating till receiving the variable nonce passes through four stations. These stations are:

- The client agent, where the nonce *n1* is initiated and sent through the message between the client agent and the TGS agent (*CReq-B*). The following syntax shows the changes of message #3 in the process ProcessC:

```
(*      Message 3       *)
new t1:timest;
new n1:nonce;
event BeginCT(T, CTkey,n1);
event BeginCS(CTkey, t1);
out(c,(enc((C,t1,n1, req),CTkey),m2));
```

- The TGS agent, where the nonce *n1* is directed from the TGS agent to the SP agent. After the TGS agent receives the nonce *n1*, it sends the nonce to the SP agent. However, the SP agent is not able to decrypt the nonce, as the nonce is encrypted in the ticket using the session key between the TGS agent and the client agent (CTkey). This can be modified in the code using the following syntax in the process ProcessT:

```
(*      Message 3       *)
in(c,(m1:bitstring,m2:bitstring));
let(hostC:client,CTkey:key)=
    decr(m2,Kt) in
let(=hostC,t3:timest,n5:nonce,r:bitstring
    )=decr( m1, CTkey) in
event BeginTC(hostC,n5);
new t12:timest;
new CSkey:key;
let m3=enc((CSkey,S,n5,t12),CTkey) in
let m4=enc((hostC,CSkey,t3,r),Ks) in
event EndTC(S,hostC,CTkey,t3,n5);
(*      Message 4       *)
out (c,(m3,m4)).
```

- The SP agent forwards the ticket along with the result to the client agent request. So that, the SP agent code remains the same.
- The client agent, which receives the message *SPRes* from the SP agent. Firstly, it verifies the value of nonce in order to execute event *EndCT()*. Then, it verifies the updated timestamp to execute event *EndCS()*.

```
(*      Message 5       *)
in (c,(m3:bitstring,m4:bitstring));
let(SCkey:key,hostY:server,,=n1,
    t11:timest)=decr(m4, CTkey) in
event EndCT(hostY,C,n1);
let(=update(t1),res:bitstring,=hostY)=
    decr(m3, SCkey)in
event EndCS(hostY,C,update(t1)).
```

As in Section 4.2, the verification is applied on the secrecy and mutual authentication between the client agent and TGS agent. However, the privacy of *n1* must be verified first of all, otherwise, its usage is not effective. Table 6 shows all the parameters in the protocol are categorized as secure or not, as obtained from the ProVerif tool.

Table 6. Results of confidentiality test on DLK protocol

Parameter	Initial status	ProVerif output
K_C, K_T, K_S	Secure	Secure
Cid, SPid	Unsecure	No output
ASid, TGSid	Unsecure	No output
CTkey	Unknown	Secure
SCkey	Unknown	Secure
Req	Unknown	Secure
Res	Unknown	Secure
t1	Unknown	Secure
U(t1)	Unknown	Secure
n1	Unknown	Secure

As for the mutual authentication in the DLK protocol, the output of the ProVerif tool confirms the availability of mutual trust between client, TGS and SP agents. Table 7 illustrates the results of all the test queries regarding authentication after the addition of the nonce in the DLK protocol.

Table 7. Results of authentication test on DLK protocol

The Quiry	ProVerif output
event(*BeginSC()*)==> event(*BeginCS()*)	**True**
event(*EndCS()*) ==> event(*EndSC()*)	**True**
event(*EndCS()*)==>event(*BeginCS()*)&&event(*BeginSC()*)	**True**
event(*BeginTC()*)==> event(*BeginCT()*)	**True**
event(*EndCT()*) ==> event(*EndSC()*)	**True**
event(*EndCT()*)==>event(*BeginCT()*)&&event(*BeginTC()*)	**True**

6. Security Analysis of the DLK Protocol

There are several advantages offered by the DLK protocol over the Kerberos protocol. One of them is the Kerberos protocol includes the network address in the tickets. However, we assume that there is no security benefit from including the network address, as the attacker could have control on the network. Thus, the attacker can easily fake his network address in order to communicate with other parties, so there is no extra security gained by relying on the network address. While the DLK protocol relies instead on the ticket in the encryption process.

The original Kerberos versions are based on the timestamp. Therefore, they should be based on a trusted clock synchronizing technique. However, even if this addresses the detection of the replay attack, the attacker could perform a real time replay attack with the typical five minutes lifetime. In our protocol, we proposed the timestamp and the updated

timestamp between the client and the SP agents to keep the signature of Kerberos in our protocol; however we can replace this technique by a challenge/ response mechanism (nonce). In addition, we could remove the timestamp from any ticket and keep the nonce only.

Password-guessing assault is a common security issue between our protocol and Kerberos. The attacker may try to capture the encrypted messages between the clients and the AS in order to get enough information about each client. Then, these encrypted messages with the password of the client will be run by the attacker through various techniques in order to guess the client password. It is enough for the attacker to guess one password in order to penetrate the security of the protocol. We can reduce the impact of such attack by forcing the client to choose a complex password or by increasing the size of the generated key from the password. Even with this counter measure we cannot prevent all password-guessing attacks or social engineering attacks.

7. Conclusions and Future Work

In this paper, we have formally analyzed the security properties of the DLK protocol that secures the messages exchanged in a distributed agent-based system. The automatic verifier, ProVerif, has been used for this analysis. This study improves the understanding of the protocol by formally verifying the mutual authentication between client and SP agents and proving the secrecy of the information inside the encrypted messages.

We were able to determine some attacks that might occur due to the security protocol design deficiencies. Testing the DLK protocol using ProVerif detected the possibility of a Man-In-The-Middle attack. The mutual authentication problem between the client and SP agents is fixed in this work using nonce-based authentication. The fixed protocol was then verified in ProVerif to prove that it achieved mutual authentication between the client agent and SP agents correctly.

The enhanced DLK protocol is a general security methodology scheme that provides mutual authentication, confidentiality, integrity and authorization, while it is not linked to the multi-agent system. Therefore, we plan to expand our research and find out other application areas that can adopt the DLK protocol without the agent technology. This way, the DLK protocol can be used to provide a secure distribution method for any system. In addition, the ProVerif model we did for the DLK protocol can be used in order to verify any necessary changes in security concepts of the DLK or any other security protocol that we will propose based on the DLK.

8. References

[1] RFC 3394, http://www.ietf.org/rfc/rfc3394.txt, Feb. 2012

[2] H.M.N. Al-Hamadi, C.Y. Yeun, M.J. Zemerly, M. Al-Qutayri; "Distributed lightweight Kerberos protocol for Mobile Agent Systems", IEEE GCC Conference and Exhibition, Dubai, 2011, pp. 233-236.

[3] M. Peters, P. Rogaar; "A review of ProVerif as an automatic security protocol verifier", http://agoraproject.eu/papers/A review of ProVerif as an automatic security protocol verifier.pdf, 13th of Sept. 2011.

[4] R. Kusters, T. Truderung, “Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation”, 22nd IEEE Computer Security Foundations Symposium, New York, 2009, pp.157-171.

[5] B. Blanchet, B. Smyth; "ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial", http://www.proverif.ens.fr/manual.pdf, 11th of July 2011.

[6] E. M. Clarke, J. M. Wing; "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, 1996, Vol.28, pp. 626-643.

[7] B. Blanchet; "CryptoVerif: Cryptographic protocol verifier in the computational model", http://www.cryptoverif.ens.fr, 11th of July 2011.

[8] K. Bhargavan, C. Fournet, A.D. Gordon, S. Tse; "Verified interoperable implementations of security protocols", 19th IEEE Computer Security Foundations Workshop,Venice, 5-7 July 2006, pp. 139 -152.

[9] Q. Li, F. Yang, H. Zhu, L. Zhu; "Formal Modeling and Analyzing Kerberos Protocol", 2009 WRI World Congress on Computer Science and Information Engineering, 2009, Vol.7, pp. 813-819.

[10] M. Butler; "On the Use of Data Refinement in the Development of Secure Communications Systems", Formal Aspects of Computing, 2002, Vol. 14, No.1, pp. 2–34.

[11] Z. Zhang, X. Zhang, R. Sandhu; "ROBAC: Scalable Role and Organization Based Access Control Models", CollaborateCom 2006. International Conference on Collaborative Computing: Networking, Applications and Worksharing, Georgia, 2006, pp. 1-9.

[12] RFC 4120, http://tools.ietf.org/html/rfc4120#ref-NT94, 11th of July 2011.

[13] M.A. Sirbu, J.C.-I. Chuang, "Distributed authentication in Kerberos using public key cryptography", 1997 Symposium on Proceedings Network and Distributed System Security, California, 1997, pp. 134-141.

[14] M. Abadi, C. Fournet; "Mobile Values, New Names, and Secure Communication", Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL’01), 2001, pp. 104–115.

[15] B. Blanchet, A. Chaudhuri;, "Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage" , IEEE Symposium on Security and Privacy, California, 2008, pp. 417-431.

[16] M. Bugliesi, R. Focardi, M. Maffei, “Analysis of typed analyses of authentication protocols”, 18th IEEE Workshop in Computer Security Foundations, France, 2005, pp. 112- 125.

