Modeling and Verifying TESAC Using CSP

Dongzhen Sun[1], Huibiao Zhu*[1], Yuan Fei*[2], Lili Xiao[1], Gang Lu[1], Jiaqi Yin[1]
[1]Shanghai Key Laboratory of Trustworthy Computing,
School of Computer Science and Software Engineering
East China Normal University, Shanghai, China
[2]School of Information, Mechanical and Electrical Engineering,
Shanghai Normal University, Shanghai, China

***Abstract*—Cloud computing is an emerging computing paradigm in IT industries. The wide adoption of cloud computing is raising concerns about management of data in the cloud. Access control and security are two critical issues of cloud computing. Time efficient secure access control (TESAC) model is a new data access control scheme which can minimise many significant problems. This scheme has better performance than other existing models in a cloud computing environment. TESAC is attracting more and more attentions from industries. Hence, the reliability of TESAC becomes extremely important. In this paper, we apply Communication Sequential Processes (CSP) to model TESAC, as well as their security properties. We mainly focus on its data access mechanism part and formalize it in detail. Moreover, using the model checker Process Analysis Toolkit (PAT), we have verified that the TESAC model cannot assure the security of data with malicious users. For the purpose of solving this problem we introduce a new method similar to digital signature. Our study can improve the security and robustness of the TESAC model.**

***Index Terms*—TESAC; Cloud computing; CSP; Access control;Modeling; Verification;**

I. Introduction

Cloud computing is considered to be an important driver of world-wide IT industries [1]. With the development of cloud computing technology, many data access models (ACMs) have been proposed by researchers [2]–[4]. Time efficient secure access control (TESAC) [5] model is a new data access control scheme which uses asymmetric encryption to guarantee data security. This scheme is more efficient than other existing solutions after an evaluation in terms of both theoretical and experimental results.

The main types of access control models can be roughly divided into three categories [6]: Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role Based Access Control (RBAC). In order to achieve the goal of the network as a high-performance computer, cloud service providers specify suitable access control policies for users to access data and other resources. For limiting user access rights in different situations, Ferraiolo et al. [2] put forward the role-based access control (RBAC), where cloud service provider determines the user access to the system by means of the job role. Gao et al. [3] came up with novel data access control (NDAC) to ensure secured and confidential data communication between users and cloud servers. For the express purpose of providing a superior decision-making ability, Danwei et al. [4] presented usage control-based access (UCON) model which can easily implement the security strategy of DAC, MAC and RBAC. It is a scheme that combines all the merits of the traditional access control models. These models attach great importance to security issues, but they adopt only informal methods to analyze and there is few work on the formalization of these models. Due to the superb characteristics of TESAC, the industry will definitely draw a great amount of interests in this scheme. So it is extremely important to verify and analyze it by formal methods. we use classical process algebra language Communicating Sequential Processes (CSP) [7], [8] to model TESAC, and in terms of a model checking tool Process Analysis Toolkit (PAT) [9] to verify some vital properties. The verification results demonstrate and show that the security problem really exists in this scheme. Through the formalization, we want to provide a deeper understanding of TESAC as well as its security properties, and we hope that other researchers can realize the security problems that may presence in their system from our verification results.

This paper is organized as follows: Section II presents an overview of TESAC, as well as introduction to CSP. Section III is devoted to the modelling of TESAC. The results that we use PAT to verify the original model are presented in Section IV, and give the improvement for better performance. Finally, conclusions and future directions are given in Section V.

Corresponding authors: hbzhu@sei.ecnu.edu.cn (H. Zhu),
yuanfei@shnu.edu.cn (Y. Fei).

II. Background

In this section, we will give a briefly introduce of TESAC, especially its data access control scheme in cloud computing. After that, we also give a brief introduce of process algebra CSP.

A. TESAC

TESAC is a new data access control scheme based on the users' profiles. It is different from traditional access control models implementing a cloud environment. It can resolve many problems, such as high overhead of the system, high searching time for providing the public key of the data owner, high data accessing time, etc. The scheme consists of three entities:

- **Cloud service provider:** It possesses a number of servers having sufficient storage space and power to provide infrastructure and cloud services for both users and data owners.
- **Data owner:** It sends its encrypted data or file to the cloud database and managed by servers. Data owner can be any user.
- **User:** It must be registered at the cloud server for accessing a data or file.

The specific meaning of some notations of keys are listed in Table I. The process of encryption and decryption is given in Steps 1-8. When the user wants to get data from server, the following sequence of actions occurs:

1) First, the user sends a data access request to the server.
2) When the server receivs the request, it encrypts PUOWN by using its own PRSP and PUUSR, and then, provides it to the users.
3) The user obtains PUOWN through two layers of decryption. Then the user sends a request to the data owner for obtaining the secret key and the certificate. The user encrypts the request message with PRUSR and PUOWN, and then, sends it to the data owner.

DOI reference number: 10.18293/SEKE2019-122

4) The data owner uses PROWN and PUUSR to decrypt the package to get the users message. The data owner needs to verify the legality of the user to the server.
5) The server sends a feedback message to the data owner.
6) If it is a positive feedback, the data owner uses PROWN and PUUSR to encrypt its secret and certificate. Finally, the data owner sends this package to the user.
7) The user uses PRUSR and PUOWN to decrypt the message. After the user acquires the secret and the certificate, it uses PRUSR and PUSP to encrypt the certificate. The user then sends it to the server.
8) The server decrypts the message. If the user's presented certificate is matched with the corresponding certificate of the requested data. The server provides the data to the user. Finally, the user uses PRUSR, PUSP and secret to decrypt the message to get the desired data.

TABLE I
NOTATION AND DESCRIPTION

PUOWN	public key of the data owner
PROWN	private key of the data owner
PUUSR	public key of the user
PRUSR	private key of the user
PUSP	public key of the server
PRSP	private key of the server

B. A brief overview of CSP

In this subsection we give a brief overview of CSP (Communication Sequential Processes). It is a process algebra proposed by Hoare in 1978. As one of the most mature formal methods, it is tailored for describing the interaction between concurrency systems by mathematical theories. Because of its well-known expressive ability, CSP has been widely used in many fields [10]–[12].

CSP processes are constituted by primitive processes and actions. We use the following syntax to define the processes in this paper, whereby P and Q represent processes, the alphabets α(P) and α(Q) mean the set of actions that the processes P and Q can take respectively, and a and b denote the atomic actions and c stands for the name of a channel. The syntax of CSP is given as below.

$$P, Q ::= \mathbf{Skip} \mid \mathbf{Stop} \mid a \to P \mid c!v \to P \mid c?x \to P \mid P; Q \mid P \parallel Q \mid P \Box Q \mid P \lhd b \rhd Q \mid b * Q \mid P[[a \leftarrow b]]$$

1) **Skip** indicates a basic process that terminates successfully.
2) **Stop** represents that a process can't do anything any more.
3) $a \to P$ denotes a process first engages in action a, then acts due to the specification of process P.
4) $c!v \to P$ describes a process sends a value v through channel c, then behaves according to the specification of P.
5) $c?x \to P$ represents a process receives a value and assigns it to the variable x, then the behavior is like process P.
6) $P; Q$ indicates only when process P has terminated can process Q start to perform.
7) $P \parallel Q$ means process P and process Q perform in parallel. And they must synchronize facing the same events.
8) $P \Box Q$ stands for external choice. A process behaves following the specification of process P or Q. However, the choice depends on the environment.
9) $b * Q$ expresses circulation. If the value of variable b is true, a process behaves like process Q circularly. Otherwise, it ends the circulation.
10) $P \lhd b \rhd Q$ shows conditional choice. If the condition b is true, a process acts likes P, otherwise, like Q.
11) $P[[a \leftarrow d]]$ indicates event a is replaced by d in process P.

III. MODELING TESAC

A. Sets, Messages and Channels

In order to formalize the protocol more conveniently, we give the fundamental information about sets, messages and channels. We define seven sets in our model. **Entity** set represents entities including servers, users and data owners. **Req** set denotes request and confirming messages. **ACK** set means feedback messages. **Content** set contains the content to be encrypted. **PUKey** set contains all the public key of entities, **PRKey** set contains all the private key of entities, **Sec** set contains the key that the data owner uses to encrypt the data.

Two core elements of modeling are internal processing procedures of entities and message packets transmitted between entities. Based on the sets defined above, we abstract them into different messages. We use the form *E(k,d)* to indicate that *k* is utilized to encrypt the message *d*. Each message includes a tag from the set $\{msg_{req}, msg_{key1}, msg_{data1}, msg_{ack}, msg_{key2}, msg_{data2}, msg_{ackin}\}$.

The messages that are transmitted among entities as follows:

$$\begin{aligned}
MSG_{req} &= \{msg_{req}.a.b.req, msg_{req}.a.b.cof | \\
&\quad a, b \in Entity, req, cof \in Req\} \\
MSG_{key1} &= \{msg_{key1}.a.b.E(K_1, E(K_2^{-1}, d)) | a, b \in Entity, \\
&\quad K_1 \in PUKey, K_2^{-1} \in PRKey, d \in Content\} \\
MSG_{data1} &= \{msg_{data1}.a.b.E(K_1, E(K_2^{-1}, E(K, d))) | \\
&\quad a, b \in Entity, K_1 \in PUKey, K_2^{-1} \in PRKey, \\
&\quad K \in Sec, d \in Content\} \\
MSG_{ack} &= \{msg_{ack}.a.b.x | a, b \in Entity, x \in Ack\} \\
MSG_{key2} &= \{msg_{key2}.E(K_1, (K_2^{-1}, d)).K_1^{-1}.K_2 | d \in Content, \\
&\quad K_1, K_2 \in PUKey, K_1^{-1}, K_2^{-1} \in PRKey\} \\
MSG_{data2} &= \{msgdata2.E(K_1, E(K_2^{-1}, E(K, d))).K_1^{-1}.K_2.K | \\
&\quad K_1, K_2 \in PUKey, K_1, K_2 \in PUKey, \\
&\quad K_1^{-1}, K_2^{-1} \in PRKey, K \in Sec, d \in Content\} \\
MSG_{ackin} &= \{msg_{cont}.y | y \in Content\} \\
MSG_{out} &= MSG_{req} \cup MSG_{key1} \cup MSG_{data1} \cup MSG_{ack} \\
MSG_{in} &= MSG_{key2} \cup MSG_{data2} \cup MSG_{ackin} \\
MSG &= MSG_{out} \cup MSG_{in}
\end{aligned}$$

MSG_{req} represents the set of request messages. MSG_{key1} stands for the set of two-layer encryption messages. MSG_{data1} indicates the set of messages whose real data encrypted by public and private keys. MSG_{ack} denotes feedback messages. MSG_{key2} stands for the set of three-layer encryption messages. MSG_{data2} indicates messages sent to the process specially for internal processing. MSG_{ackin} represents messages that return data to entities by internal process. MSG_{out} represents the set of messages transmitted between entities, MSG_{in} denotes internal processing messages of entities.

Then, we give the definitions of channels to model the communications between processes:

- channels between users, data owners and servers, denoted by COM_PATH: ComUS, ComUD, ComDS.
- channels of intruders who intercept users, data owners and servers, represented by $INTRUDER\_PATH$: FakeU, FakeS, FakeD.
- channels of processing messages, depicted by $PROCESS\_PATH$: GetU, GetD, GetS.

The declarations of the channels are as follows:

Channel *COM_PATH, INTRUDER_PATH* : MSG_{out}
Channel *PROCESS_PATH* : MSG_{in}

B. Overall Modeling

As mentioned above, the whole scheme contains three important entities, including $User$, $DataOwner$ and $Server$. We formalize the whole system as below.

$$System =_{df} System_0[|INTRUDERPATH|]Intruder$$
$$System_0 =_{df} User||DataOwner||Server$$
$$User =_{df} User_1[|PROCESSPATH|]ProcessU$$
$$DataOwner =_{df} DataOwner_1[|PROCESSPATH|]ProcessD$$
$$Server =_{df} Server_1[|PROCESSPATH|]ProcessU$$

User, *DataOwner* and *Server*, as their names demonstrate, represent the user, the data owner and the server. *ProcessU*, *ProcessD* and *ProcessS* denote the internal processing procedure of the user, the data owner and the server. Considering the existence of intruders, we also build process $Intruder$ to simulate the behavior of intruders who eavesdrop and modifies messages. Interprocess communication between processes are illustrated in Fig.2.

C. User Modeling

We first formalize process $User_0$ to describe the behavior of a user process without intruders.

$$\begin{aligned}
User_0 =_{df}\ & ComUS!msg_{req}.U.S.req_data \rightarrow \\
& ComUS?msg_{key1}.S.U.E(PUUSR, E(PRSP, PUOWN)) \rightarrow \\
& GetU!msgkey2.E(PUUSR, E(PRSP, PUOWN)). \\
& PRUSR.PUSP \rightarrow GetU?msg_{ackin}.PUOWN \rightarrow \\
& ComUD!msg_{key1}.U.D.E(PUOWN, E(PRUSR, req_sc)) \rightarrow \\
& ComUD?msg_{key1}.D.U.E(PUUSR, E(PROWN, (s, c))) \rightarrow \\
& GetU!msg_{key2}.E(PUUSR, E(PROWN, (s, c))). \\
& PRUSR.PUOWN \rightarrow GetU?msg_{ackin}.s.c \rightarrow \\
& ComUS!msg_{key1}.U.S.E(PUSP, E(PRUSR, c)) \rightarrow \\
& ComUS?msg_{data1}.S.U.E(PUUSR, E(PRSP, E(s, data))) \rightarrow \\
& GetU!msg_{data2}. \\
& E(PUUSR, E(PRSP, E(s, data))).PRUSR.PUSP.s \rightarrow \\
& GetU?msg_{ackin}.data \rightarrow User_0;
\end{aligned}$$

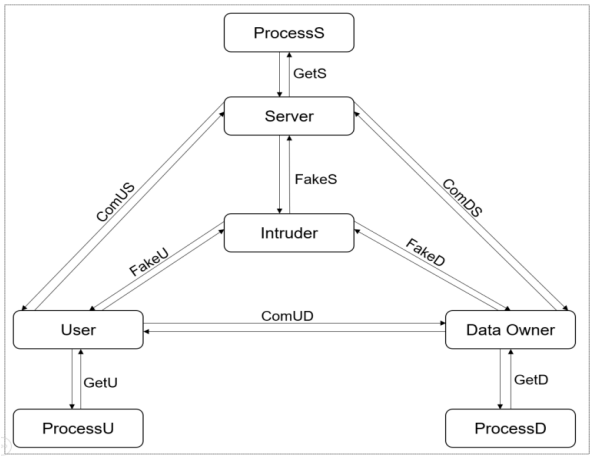

Fig. 1. Interprocess communication between processes in model

where, req_data represents the data request sent from the user to the server. req_sc represents the request of the secret and the certificate that the user sends to the data owner. s represents the secret and c represents the certificate. The six actions on channel $ComUS$ and $ComUD$ correspond to Steps 1-8 of User in Fig.1 in order. After receiving the encrypted messages, entity $User_0$ sends them to internal processing part $ProcessU$ by way of channel $GetU$, as well as accepts the decrypted messages.

Then the existence of intruder actions needs to take into consideration. For example, we must allow the instances of data request to be faked, the instances of responses of $PUOWN$ to be intercepted, etc. We do this via renaming.

$$\begin{aligned}
User_1 =_{df}\ User_0[[& ComUS?\{|ComUS|\} \leftarrow ComUS?\{|ComUS|\}, \\
& ComUS?|ComUS| \leftarrow FakeU?|ComUS|, \\
& ComUS!|ComUS| \leftarrow ComUS!|ComUS|, \\
& ComUS!|ComUS| \leftarrow FakeS!|ComUS|, \\
& ComUD?|ComUD| \leftarrow ComUD?|ComUD|, \\
& ComUD?|ComUD| \leftarrow FakeU?|ComUD|, \\
& ComUD!|ComUD| \leftarrow ComUD!|ComUD|, \\
& ComUD!|ComUD| \leftarrow FakeD!|ComUD|]]
\end{aligned}$$

$\{|c|\}$ denotes the set of all communications over channel c. Whenever $User_0$ does an action on channel $ComUS$ or $ComUD$, $User_1$ will does a corresponding action on channels with prefix Com or channels with prefix $Fake$. Here, channels with prefix Com only include $ComUS$ and $ComUD$ and channels with prefix $Fake$ only include $FakeU$ and $FakeS$.

We can define CSP processes representing the data owner and server similarly.

D. ProcessU Modeling

In order to simulate the internal process of the user, we use $ProcessU$ to deal with decrypting message. We must consider the possibility of intruder actions.

$$\begin{aligned}
ProcessU =_{df}\ & GetU?msg_{key2}.E(PUUSR, E(PRSP, PUOWN)). \\
& PRUSR.PUSP \rightarrow \\
& \left(\begin{array}{l} GetU!msg_{ackin}.PUOWN \rightarrow ProcessU \\ \lhd(((PUSP == PRSP)||(PUI == PRI)) \\ \wedge(PUUSR == PRUSR))\rhd \\ (GetU!msg_{ackin}.NO \rightarrow ProcessU) \end{array} \right) \\
& \Box GetU msgkey2.E(PUUSR, E(PROWN, (s, c))). \\
& PRUSR.PUOWN \rightarrow \\
& \left(\begin{array}{l} GetU!msg_{ackin}.PUOWN \rightarrow ProcessU \\ \lhd(((PUOWN == PROWN)||(PUI == PRI)) \\ \wedge(PUUSR == PRUSR))\rhd \\ (GetU!msg_{ackin}.NO \rightarrow ProcessU) \end{array} \right) \\
& \Box GetU?msg_{data2}.E(PUUSR, E(PRSP, E(sec, data))). \\
& PRUSR.PUSP.s \rightarrow \\
& \left(\begin{array}{l} GetU!msgackin.PUOWN \rightarrow ProcessU \\ \lhd(((PUSP == PRSP)||(PUI == PRI)) \\ \wedge(PUUSRmatchPRUSR) \wedge (sec == s))\rhd \\ (GetU!msgackin.NO \rightarrow ProcessU) \end{array} \right)
\end{aligned}$$

We use PUI to represent the public key of the intruder, PRI represents the private key of the intruder. $ProcessU$ receives the encrypted message with decryption keys by channel $GetU$. Then it judges whether the decryption key can decrypt the message successfully. If the key does not match, $ProcessU$ returns a negative message to $User_0$. Else, it sends the decrypted content to $User_0$. The internal process $ProcessD$ of the data owner and $ProcessS$ of the server can be defined similarly.

E. Intruder Modeling

Finally, we give the formalization of the intruder. We also regard the intruder as a process that can perform any attack as a real world intruder can be. It can intercept or fake messages in the

communication on channel $ComUS, ComUD$ and $ComDS$. We define the set of facts that an intruder might learn as follows:

$$
\begin{aligned}
&Fact =_{df} \\
&\quad \{U, S, D\} \cup \{PUOWN, PUUSR, PUSP\} \\
&\quad \cup \{E(k, content) | k \in \{Sec, PUKey, PRKey\}, \\
&\quad content \in Content, (s, c)\} \cup MSG_{out} \cup \{PUI, PRI\}
\end{aligned}
$$

Intruder can derive new facts from the ser of $Facts$ it has learned. We use the symbol $F \mapsto f$ to indicate that the fact f can be derived from the set F of facts. The definition is given as follows:

$$
\begin{aligned}
&\{K_1^{-1}, E(K_1, E(K_2^{-1}, d)\} \mapsto E(K_2^{-1}, d), \{K_2, E(K_2, d)\} \mapsto d \\
&\{K_2, d\} \mapsto E(K_2^{-1}, d), \{K_1, E(K_2^{-1}, d)\} \mapsto E(K_1, E(K_2^{-1}, d) \\
&F \mapsto f \wedge F \subseteq F' \Rightarrow F' \mapsto f
\end{aligned}
$$

The first two rules represent encryption and the third and the fourth represent decryption. The final rule means if the intruder can derive the fact f from a set of facts F, then f can also be derived from a larger set F'.

We give a definition of **Info** function in which the intruders can learn by seeing the intercepted messages, shown as follows:

$$
\begin{aligned}
&Info(msg_{req}.a.b.req) =_{df} \{a, b, req\} \\
&Info(msg_{req}.a.b.cof) =_{df} \{a, b, cof\} \\
&Info(msg_{key1}.a.b.E(K_1, (K_2^{-1}, d))) =_{df} \{a, b, E(K_1, (K_2^{-1}, d))\} \\
&Info(msg_{data1}.a.b.E(K_1, E(K_2^{-1}, E(k, d)))) =_{df} \\
&\qquad \{a, b, E(K_1, E(K_2^{-1}, E(k, d)))\} \\
&Info(msgack1.a.b.x) =_{df} \{a, b, x\}
\end{aligned}
$$

where $a, b \in Entity$, $req, cof \in Req$, $K_1 \in PUKey$, $K_2^{-1} \in PRKey, d \in content, k \in Sec, x \in Ack$.

We define a channel $deduce$ to be used for deducing new facts. The definition is given as follows:

Channel deduce: Fact.P(Fact)

All the messages transmitted between entities can be overheard by the intruder. It can deduce a new fact from ones it has already known.

It can also fake some messages if he knows all the sub-messages. The formalization of $Intruder_0$ is defined as below:

$$
\begin{aligned}
&Intruder_0(F =_{df} \\
&\Box\Box m \in MSG_{out} FakeU?m \rightarrow \\
&FakeS!m\{user\_fake\_success = true\} \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box m \in MSG_{out} FakeS?m \rightarrow \\
&FakeU!m\{server\_fake\_success = true\} \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box m \in MSG_{out} FakeU?m \rightarrow \\
&FakeD!muser\_fake\_success = true \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box m \in MSG_{out} FakeD?m \rightarrow FakeU!m \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box m \in MSG_{out} FakeD?m \rightarrow FakeS!m \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box m \in MSG_{out} FakeS?m \rightarrow \\
&FakeD!mserver\_fake\_success = true \rightarrow \\
&Intruder_0(F \cup Info(m)) \\
&\Box\Box f \in Fact, f \notin F, F \mapsto fdeduce.f.F \rightarrow \\
&Intruder_0(F \cup f)
\end{aligned}
$$

When $Intruder\_0$ receives some messages, if it is not encrypted by another entity with its public key, $Intruder\_0$ can replace some content in the message and send it to the original receiving entity; If the message is encrypted with the entitys public key, $Intruder\_0$ can not know or replace the content of the message. Because it does not know the private key that matches the public key. However, $Intruder\_0$ has its own public key and private key, allowing it to fake messages to send to the entities. We give the definition of IK to represent the initial knowledge of the intruder:

$$
\begin{aligned}
&Intruder =_{df} Intruder_0(IK) \\
&IK =_{df} \{U, S, D, PUI, PRI\}
\end{aligned}
$$

IV. VERIFICATION AND IMPROVEMENT

In this section, we will verify the four properties (deadlock freedom, user faking, server faking and protocol completeness) by virtue of the model checker PAT. According to the verification results, we improve the original model for a better safety performance.

A. Security Specification

We allow intruders to perform a series of intrusive actions and give some facts that intruders can learn. Under these conditions, whether the intruders can attack the system successfully by intercepting and faking. If the intruder can get $PUOWN$ or data from the server, then it can be concluded that the system was successfully attacked by the intruder. This is in the ideal situation, but the protocol is running in an open environment, then $PUOWN$ is public as a public key. Once an intruder knows the data owner's public key, it is critical that the system can detect and prevent the disclosure of the message. Thus we will verify our systems against the following specification:

$$
\begin{aligned}
&SPEC_u =_{df} CHAOS - \left(\sum -\{FakeU\}\right) \\
&SPEC_d =_{df} CHAOS - \left(\sum -\{FakeD\}\right) \\
&SPEC_s =_{df} CHAOS - \left(\sum -\{FakeS\}\right)
\end{aligned}
$$

If the system with intruders refines these specifications, then it would indeed be secure.

B. Properties Verification

We describe some security properties as well as their assertion descriptions in PAT code and give the verification result.We do not give a description of the intruder disguised as the data owner, because the disguise as a data owner has no realistic meaning. Because the data owner can be any user and intruders will not use this protocol to get its own data. We use $System$ to represent the original model.

Property 1: Deadlock Freedom

The model should not run into a deadlock state. In PAT, there is a primitive to describe this situation:

$$
\#assert\ System\ deadlockfree;
$$

Property 2: User Faking

This property represents that the intruder has successfully pretended to be a legal user without being realized by the system. We define a boolean variable $user\_fake\_success$ for verification in PAT.

$$
\begin{aligned}
&\#define\ User\_Fake\_Success \\
&\qquad user\_fake\_success == true; \\
&\#assert\ System\ reaches\ User\_Fake\_Success;
\end{aligned}
$$

Property 3: Server Faking

Similarly, this property represents that the intruder has successfully

pretended to be the server without being realized by other entities. We define a boolean variable $server\_fake\_success$ for verification in PAT.

$$\#define\ Server\ Fake\ Success$$
$$server\_fake\_success == true;$$
$$\#assert\ System\ reaches\ Server\_Fake\_Success;$$

Property 4: Protocol Completeness

Because $PUOWN$ is a public key, if the intruder already acquires the $PUOWN$, then the protocol cannot be fully executed. The server receives three different requests when it executes a complete protocol. We use a constant n to record whether the server has accepted three requests. We define a boolean variable $protocol\_completeness$ for verification in PAT.

$$\#define\ Protocol\ Completeness$$
$$protocol\_completeness == true;$$
$$\#assert\ System\ reaches\ Protocol\_Completeness;$$

The verification results are shown in Fig.2. Deadlock freedom is valid which means that the $System$ model does not run into a deadlock state. The $User\_Fake\_Success$ property is valid and PAT provides a trace which leads to a state where this property is satisfied. Intruder can implement an event req_sc on channel $FakeD$ to be a legal user without being realized. Similarly, the $Server\_Fake\_Success$ property is valid shows that intruder can successfully pretend to be a legal server without being realized. The third property $Protocol\_Completeness$ is not valid. It represents that intruder can obtain data without executing the complete protocol. Property 2, 3 and 4 verified the insecurity of TESAC.

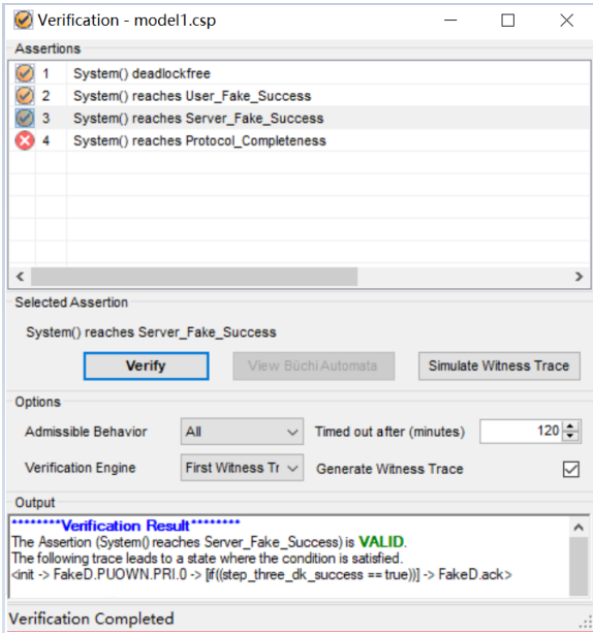

Fig. 2. Verification Result of the model

C. Attack and Improvement

As the verification result shows above, although this scheme adopts asymmetric encryption and the server uses the certificate to authenticate the user, the system is still not reliable. The server only requests users to obtain the data decryption key and certificate from the data owner for decrypting the data at the time of first data accessing. When users initiate a data access request for the subsequent time, the server directly grants the data. So once the intruder successfully obtains data for the first time, there will be no security in the system. An example trace of the intruder acquires data successfully, which is presented as below:

$$ComUS!msg_{req}.U.S.req\_data \rightarrow$$
$$FakeU?msg_{key1}.S.I.E(PUI, E(PRSP, PUOWN)) \rightarrow$$
$$FakeD!msg_{key1}.IU.D.E(PUOWN, E(PRI, req\_sc)) \rightarrow$$
$$ComDS?msg_{req}.D.S.cof \rightarrow FakeD!msgreq.IS.D.ack \rightarrow$$
$$FakeU?msg_{key1}.D.U.E(PUI, E(PROWN, (s, c)))$$

First of all, the user sends a data request to the server by channel $ComUS$. The intruder intercepts the message sent by the server to the user through channel $FakeU$. Then the intruder pretends to be a user to request the secret and the certificate from the data owner. Data owner validates the users authenticity from the server, this message is intercepted by the intruder and returns a positive feedback message to the data owner. Finally, the data owner provides the certificate and secret key to the intruder.

That is to say, once an intruder obtains a legitimate identity using the above path, the intruder can obtain data before deleting it from the servers user list, which will undoubtedly cause disaster. Next we will make changes to the protocol and not to reduce its time efficiency.

The request information sent by the user for $PUOWN$ cannot confirm the origins, as well as the confirm feedback information sent by the server to data owner. In order to change this situation, we will improve the protocol. When the user requests to obtain $PUOWN$, it needs to sign with his own private key, in this way the server can use the digital signature to authenticate the user. Similarly, the confirm message sent by data owner to server and the feedback message of server returnd to data owner are all signed with their own private key. The specific updating of the model are as follows:

$$MSG_{req} = \{msg_{req}.a.b.E(K_1, E(K_2^{-1}, req\_data)),$$
$$msg_{req}.a.b.E(K_1, E(K_2^{-1}, cof))|a, b \in Entity,$$
$$K_1 \in PUKey, K_2^{-1} \in PRKey, req, cof \in Content\}$$
$$MSG_{ack} = \{msg_{ack1}.a.b.E(K_1, E(K_2^{-1}, x))|$$
$$a, b \in Entity, x \in Ack, K_1 \in PUKey, K_2^{-1} \in PRKey\}$$

The message changed in the new model corresponds to actions 1, 4 and 5 in Fig. 1. These messages are all related to the server. So we give the updated server process which can be formalized as follow:

$$SERVER_0 =_{df} Initialization\{n = 0\} \rightarrow$$
$$ComUS?msg_{req}.U.S.E(PUSP, E(PRUSR, req\_data)) \rightarrow$$
$$GetS!msg_{key2}.E(PUSP, E(PRUSR, req\_data))$$
$$.PRSP.PUUSR \rightarrow Gets?msg_{ackin}.req\_data \rightarrow$$
$$ComUS!msg_{key1}.S.U.E(PUUSR, E(PRSP, PUOWN))\{n = 1\} \rightarrow$$
$$ComDS?msg_{req}.D.S.E(PUSP, E(PROWN, cof)) \rightarrow$$
$$GetS!msg_{key2}.E(PUSP, E(PROWN, cof))$$
$$.PRSP.PWOWN \rightarrow Gets?msg_{cont}.cof \rightarrow$$
$$ComDS!msg_{ack}.S.D.E(PUOWN, E(PRSP, ack))\{n = 2\} \rightarrow$$
$$ComUS?msg_{key1}.U.S.E(PUSP, E(PRUSR, c)) \rightarrow$$
$$GetS!msg_{key2}.E(PUSP, E(PRUSR, c)).PRSP.PUUSR \rightarrow$$
$$GetS?msg_{ackin}.c \rightarrow$$
$$ComUS!msg_{data1}.S.U.E(PUUSR, E(PRSP, E(s, data)))\{n = 3\} \rightarrow$$
$$SERVER_0;$$

We define a variable n to record the number of messages sent by the server. First, $SERVER\_0$ initializes the variable n. It will be assigned a value of 1 when the server sends $PUOWN$ to the

user by channel $ComUS$. $SERVER\_0$ communicates with its internal process $PROCESSS$ through channel $GetS$. The server will return a feedback message to the confirmation message sent by the data owner. If this action completes successfully, the value of n becomes 2. The server provides the data to the user if the user's presented certificate is matched with the corresponding certificate of the requested data. Meanwhile, the value of n is modified to 3.

Then we update $ProcessS$ to be $PROCSSS$ correspondingly.

$$
\begin{aligned}
&PROCESSS =_{df} \\
&\quad GetS?msg_{key2}.E(PUSP, E(PRUSR, req\_data)). \\
&\quad PRSP.PUUSR \rightarrow \\
&\quad \left(\begin{array}{l} GetS!msg_{ackin}.req\_data \rightarrow PROCESSS \\ \lhd((PUSP == PRSP) \wedge (PUUSR == PRUSR)) \rhd \\ (GetU!msgackin.NO \rightarrow PROCESSS) \end{array} \right) \\
&\quad \Box GetS!msg_{key2}.E(PUSP, E(PROWN, cof)).PRSP.PUOWN \rightarrow \\
&\quad \left(\begin{array}{l} GetS!msg_{ackin}.cof.YES \rightarrow PROCESSS \\ \lhd((PUSP == PRSP) \wedge (PUOWN == PROWN) \wedge \\ (n == 1)) \rhd (GetS!msgackin.cof.NO \rightarrow \\ PROCESSS)) \end{array} \right) \\
&\quad \Box GetS?msg_{key2} E(PUSP, E(PRUSR, c)).PRSP.PUUSR \rightarrow \\
&\quad \left(\begin{array}{l} GetS!msg_{ackin}.c \rightarrow PROCESSS \\ \lhd((PUSP == PRSP) \wedge (PUUSR == PRUSR)) \rhd \\ (GetS!msgackin.cof.NO \rightarrow \\ PROCESSS)) \end{array} \right)
\end{aligned}
$$

$PROCSSS$ receives the message sent by $SERVER\_0$ through the channel $GetS$. It should be noted that if the message is a confirmation message, $PROCSSS$ will judge whether the value of n is 1. This is to confirm that the user is getting PUOWN from the server instead of others.

D. Overall Modeling

We formalize the new system as below.

$$
\begin{aligned}
&SYSTEM = SYSTEM_1[|INTRUDERPATH|]INTRUDER, \\
&SYSTEM_1 =_{df} USER||DATAOWNER||SERVER, \\
&USER =_{df} USER_1[|PROCESSPATH|]PROCESSU, \\
&DATAOWNER =_{df} \\
&\qquad DATAOWNER_1[|PROCESSPATH|]PROCESSD, \\
&SERVER =_{df} \\
&\qquad SERVER_1[|PROCESSPATH|]PROCESSS.
\end{aligned}
$$

The verification results are given as below:

The verification results show that **User_Fake_Success** and **Server_Fake_Success** properties are invalid, which means that system realized that the intruder were performing actions of invading and immediately stopped the process. The **Protocol_Completeness** property is valid, which means that after we improve the protocol. It can be guaranteed that the protocol is executed completely once when the user requests data.

E. CONCLUSION AND FUTURE WORK

In this paper, we have formalized the TESAC using classical process algebra language CSP. Then, we have fed the model into the model checker PAT and verified the security of the model by asserting four properties (deadlock freedom, user faking, server faking, protocol completeness). The verification results show that user faking property and server faking property are valid. It means intruder can successfully pretend to be a legal user or server without being realized. Protocol completeness property is not valid. It demonstrates that TESAC cannot guarantee users to follow all the steps for data accessing when the intruder attacks. This means that TESAC is not secure in a cloud computing environment. In order to solve these problems, we have made improvements to TESAC by introducing a method similar to digital signature. We have verified the improved

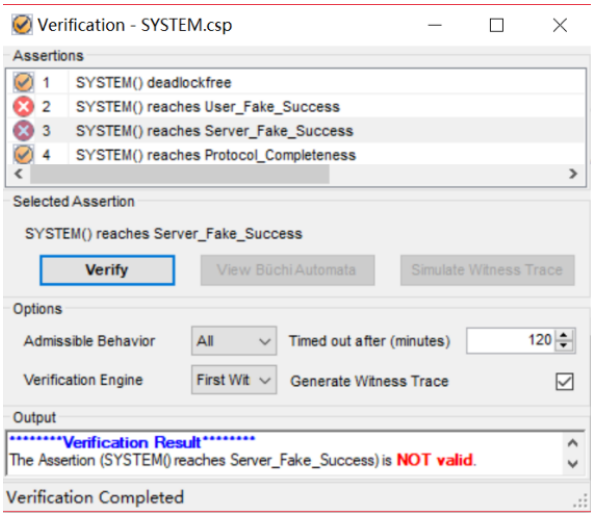

Fig. 3. Verification Result of the improved model

model with respect to the four properties, and the new verification results indicate that the improved model can prevent intruders from invading the system. The security and robustness of TESAC would be improved through our efforts.

ACKNOWLEDGMENT

This work was partly supported by National Natural Science Foundation of China (Grant No. 61872145) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No.ZF1213).

REFERENCES

[1] M. ARMBRUST, "Above the clouds : A berkeley view of cloud computing," *Science*, vol. 53, pp. 07–013, 2009.
[2] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," *CoRR*, vol. abs/0903.2171, 2009.
[3] X. W. Gao, Z. M. Jiang, and R. Jiang, "A novel data access scheme in cloud computing," vol. 756. Trans Tech Publications, 10 2013, pp. 2649–2654.
[4] D. Chen, X. Huang, and X. Ren, "Access control of cloud service based on UCON," in *Cloud Computing, First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*, 2009, pp. 559–564.
[5] S. Namasudra and P. Roy, "Time saving protocol for data accessing in cloud computing," *IET Communications*, vol. 11, no. 10, pp. 1558–1565, 2017.
[6] Z. Mahmood, "Continued rise of the cloud," 2014.
[7] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.
[8] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984.
[9] PAT, "Pat: Process analysis toolkit." [Online]. Available: https://doi.org/10.1145/828.833
[10] A. W. Roscoe, *Understanding Concurrent Systems*, ser. Texts in Computer Science. Springer, 2010.
[11] G. Lowe and A. W. Roscoe, "Using CSP to detect errors in the TMN protocol," *IEEE Trans. Software Eng.*, vol. 23, no. 10, pp. 659–669, 1997.
[12] Y. Fei and H. Zhu, "Modeling and verifying NDN access control using CSP," 2018, pp. 143–159.