Hindawi
Security and Communication Networks
Volume 2017, Article ID 7042835, 15 pages
https://doi.org/10.1155/2017/7042835

Research Article

A New Method to Analyze the Security of Protocol Implementations Based on Ideal Trace

Fusheng Wu,[1] Huanguo Zhang,[1] Wengqing Wang,[2] Jianwei Jia,[2] and Shi Yuan[2]

[1]*Computer School of Wuhan University, Wuhan 430072, China*
[2]*Key Laboratory of Aerospace Information Security and Trusted Computing of Ministry of Education, Wuhan University, Wuhan 430072, China*

Correspondence should be addressed to Huanguo Zhang; liss@whu.edu.cn

Received 6 May 2017; Revised 12 August 2017; Accepted 20 August 2017; Published 18 October 2017

Academic Editor: Jiankun Hu

The security analysis of protocols on theory level cannot guarantee the security of protocol implementations. To solve this problem, researchers have done a lot, and many achievements have been reached in this field, such as model extraction and code generation. However, the existing methods do not take the security of protocol implementations into account. In this paper, we have proposed to exploit the traces of function return values to analyze the security of protocol implementations at the source code level. Taking classic protocols into consideration, for example (like the Needham-Schroeder protocol and the Diffie-Hellman protocol, which cannot resist man-in-the-middle attacks), we have analyzed man-in-the-middle attacks during the protocol implementations and have carried out experiments. It has been shown in the experiments that our new method works well. Different from other methods of analyzing the security of protocol implementations in the literatures, our new method can avoid some flaws of program languages (like C language memory access, pointer analysis, etc.) and dynamically analyze the security of protocol implementations.

1. Introduction

With the fast development of the network communication, information security is becoming more and more important [1, 2]. To protect the network information from attacks, protocols are usually applied. However, general methods (e.g., formal method, computational model, and computational soundness formal) cannot guarantee the security of protocols during the process of their implementations. That is, even if protocols have been theoretically proved to be secure, some insecure factors (like the language characteristics of protocols' source codes, the operating environments of the protocol implementations) arise when implementing them at the source code level. Therefore, researchers focus on the security analysis of protocol implementations at the source code level [3].

During implementing at the source code level, it is difficult to guarantee the security of protocol specifications due to language characteristics (such as C language memory access, pointer analysis, etc.). Hence, it is more complex to analyze the security of protocol implementations at the source code level compared to that of protocols on the theoretical level. To avoid these insecure factors that languages bring, some methods have been proposed to analyze the security of protocol implementations at the source code level. Among them are two representatives: model extraction and code generation. Model extraction is applied to avoid the problem of concrete state space explosion. During the extracting process, an abstract mapping is set up to map a concrete protocol model onto a corresponding abstract model and its properties onto corresponding abstract properties. If the security properties of the protocols on the abstract model have been proved to be sound and the abstract mapping has been proved to be reliable, then the security properties on the concrete model are proved. It guarantees the security of the protocol implementation and provides its reliability's demonstration. Related research achievements include [4–7]. Sometimes leaks arise in the process of protocol implementations due to the design imperfection, which leads protocol implementations to be insecure (such as the SSL protocol, the TLS protocol). To avoid these cases, code generation is applied. That is, protocol specifications are analyzed before

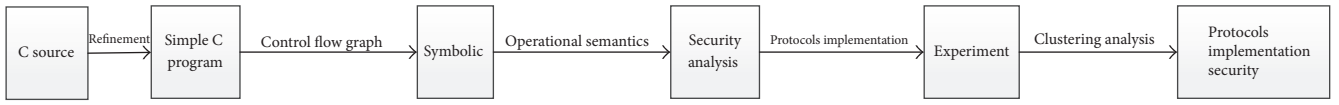

Figure 1: The security analysis process of protocol implementations.

implementations. After that, refined mapping and some related running choices (like concrete program languages or running environments) are applied to map a program abstract model onto a corresponding concrete model and its abstract properties onto corresponding concrete properties. If the security properties of the protocol on the abstract model have been proved to be sound and the mapping has been proved to be reliable, the properties on the concrete model are proved. Related research achievements include [8–10]. Now model extraction and code generation have been widely applied to protocol security analysis. However, neither model extraction nor code generation can guarantee the security of protocol implementations due to program language flaws, which causes the gap between the protocol security analysis on theoretical level and the security analysis of the protocol implementations at the source code level [11, 12].

The thought of our new method to analyze the security of protocol implementations at the source code level derives from a daily life phenomenon. When an object moves from A to B in an environment without any barriers (called the ideal environment), a trace will be produced by the object, which is called the ideal trace. In nonideal environments, the object will be attacked by a third party, and its trace (called nonideal trace) will deviate from the ideal trace (the detailed definitions of ideal trace and nonideal trace are given in Section 5). If the moving behavior is rectified after the object is attacked, the degree of deviation of its nonideal trace from the ideal trace will become smaller. Based on the thought above, we propose a new method to analyze the security of protocol implementations by means of the traces. The traces consist of the sets of function return values when implementing a protocol. Our new method is carried out like this: we exploit the ideal trace and the method of cluster analysis (including the degree of deviation and the similarity of the trace sequences) as the evaluated reference to analyze protocol security. To prove our new method, with strong simulation of π-calculus we refine the source codes of the Needham-Schroder protocol as an example and set up the security analysis model of protocol implementations with labelled transition systems [13]. Besides, taking the Needham-Schroder protocol and the Diffie-Hellman protocol, for example, we verify man-in-the-middle attacks on OpenSSL with our new method. To describe the specific steps of the security analysis in detail, a flow graph is drawn in Figure 1.

Our original contributions are as follows:

(1) We bring in the ideal trace as the evaluated reference of the security analysis of protocol implementations at the source code level and setting up a bijection of the mapping of events onto traces.

(2) We propose a method for analyzing the security of protocol implementations by comparing the similarity and the deviation between nonideal traces and the ideal trace during implementing protocols at the source code level.

The remaining work of our article is as follows: a summary of related works in Section 2; the preliminaries in Section 3; building a new model in Section 4; the security analysis of protocols in Section 5; taking classical protocol implementations, for example, and doing experiments in Section 6; conclusion and future work in Section 7.

2. Related Work

It is practical and valuable to analyze the security of protocol implementations at the source code level. In recent years, researchers focus on this field and here we can find great achievements [14].

The security analysis of protocol implementations at the source code level is very complex and different from common protocol security analysis [15], for it must take program language structure and running environment into account, which adds difficulty to the security analysis of protocol implementations. To solve this problem, literature [4] has done some researches, analyzing the security of implementing the Needham-Schroeder protocol written in C language. In literature [4], C language annotations act as trust assertions, and a trust assertion model is established for protocol security analysis by means of Horn Logic. Some researchers have used some function libraries and intermediate languages, like C language, to automatically analyze the security of protocol implementations. This method has avoided the problems that C language structure brings about (like pointer operation, buffer overflow, etc.). The related literatures are [5–7, 16, 17]. Similarly, some researchers have applied the latest techniques or tools to the security analysis of protocol implementations at the source code level, which is a new direction in this field. For example, based on VCC, C language application program interface (API) has been applied to the analysis security of protocol implementations at the source code level. Literature [18] has proposed a general method for protocol security analysis, which has distinguished two different items mapped onto the same array. In literature [19], a general verifying method has been applied to practical TPM and HSMs platforms. Literature [20] has exploited interface constraint and program logic reasoning to analyze the security of protocol implementations at the source code level. To reduce the difficulty brought about by C language structure when analyzing the security of protocol implementations, researchers have exploited C language complier to analyze the security of protocol implementation. The examples are [21, 22].

Transport Layer Security (TLS) has been widely put into practice so it is a focus to analyze the security of

protocol implementations on the base of TLS. For example, literature [23] has proposed that the OpenSSL crypto library can be applied to the security analysis of implementing TLS in the C language environment. Another example is literature [24], which has proposed to analyze the security of protocols on base of the control-flow integrity of the message authentication codes (MACs). This method has taken it into account that an adversary attack protocols by means of C/C++ pointer and memory leaks during the process of protocol implementations.

The methods mentioned above have promoted the research of the security analysis of protocol implementations. However, they cannot settle the security problems that are caused by the inherent flaws of program language structures. Compared with them, our new method can avoid the flaws of program language structure and dynamically analyze the security of protocol implementations at the source code level. Hence, it is helpful and valuable for protocol design, security verification, and security evaluation [25].

3. Preliminaries

For better understanding of our new method, it is necessary to have a basic knowledge of labelled transition system, strong simulation, program refinement, and so on.

3.1. Labelled Transition System and Strong Simulation. According to the automata theory, a labelled transition system is a system transferring model which consists of start point, input label, and terminal point. But a labelled transition system has no fixed start states and accepting states, which is different from automata. Therefore, any state in a labelled transition system can act as a start state. Therefore, a labelled transition system has an advantage over other systems when dynamically analyzing the security of protocol implementations at the source code level.

Definition 1 (labelled transition system [13]). A labelled transition system (LTS) over Act is a pair (Q, T) consisting of

(1) a set Q of states;

(2) a ternary relation $T \subseteq (Q \times \text{Act} \times Q)$, known as a transition relation.

If $(q, \alpha, q') \in T$, we write $q \xrightarrow{\alpha} q'$, and we call q the source and q' the target of the transition. If $q \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} q_n$, then we call q_n a derivate of q under $\alpha_1\alpha_2 \cdots \alpha_n$.

Definition 2 (strong simulation [13]). Let (Q, T) be an LTS, and let S be a binary relation over Q. Then S is called a strong simulation over (Q, T); if, whenever pSq, $p \xrightarrow{\alpha} p'$ then there exists $q' \in Q$ such that $q \xrightarrow{\alpha} q'$ and $p'Sq'$.

We say that q strongly simulates p if there exists a strong simulation S such that pSq.

Definition 3 (extended strong simulation). Let (Q, T) be an LTS and (Q', T') be an LTS and S be a binary relation over $Q \times Q'$, and let $p, q \in Q$. If pSq satisfies

(1) $Q \subseteq Q'$, $T \subseteq T'$,

(2) $p', q' \in Q'$ and $\alpha = \alpha'$, $\alpha' \in Q'$, and if $p \xrightarrow{\alpha} p'$, then there exists q' such that $q \xrightarrow{\alpha} q'$ and $p'Sq'$. Then S is called an extended strong simulation over a 4-tuple (Q, Q', T, T').

3.2. Program Refinement. When implementing a protocol, it is very difficult to verify its properties in concrete program state space (called the state-explosion problem [26]). Program refinement is exploited to solve this problem. Program refinement, which simplifies two programs with refined relation, is an important technique of verifying programs and is useful to reduce program state space.

Definition 4 (program refinement). Let p be a concrete program, and then there exists p', a refined program of p. That is, the process from p to p' is recursive. The behaviors produced by program p' belong to the subset of the behaviors produced by program p. Therefore, p' will never produce the behaviors that p cannot produce, and wherever a program p is applied, p' can be used to replace it. Hence, p' is called a refined program of p.

According to Definition 4, many program verification problems can be reduced to refined verification. For example, the verification of a program or an algorithm is always reduced to the refined relation between programs and protocol specifications. Similarly, there exists refined relation between a protocol implementation and its specifications. To reduce refined relation and refine source codes, we design a refining algorithm of protocol implementations, shown in Algorithm 1.

The complexity analysis of the algorithm consists of two circulations. One circulation achieves the goal of choosing functions. The other labels the functions which satisfy the condition, and refines source codes. Assuming there exists N functions in the source codes, among these functions, M functions can satisfy protocol specifications ($M \leq N$). The worst case is $M = N$. Hence, the complexity of the algorithm is $O(2N) \approx O(N)$, which is polynomial time. The algorithm can be achieved.

3.3. The Program Refinement of Protocol Source Codes. In most cases, the program source codes consist of functions, such as C language source codes, Java language source codes, and F# language source codes. According to the rules of C language program execution, when implementing a protocol, called function return values, the function behaviors are decided, and the sets of called function return values decide the behavior traces of protocol interactive communications. Therefore, if the functions of source codes are refined, the difficulty of the state-explosion problem will be reduced and the security properties of the protocol will not change after refinement. Hence, we can refine a protocol to analyze its security if the refinement relation of source codes exists.

Proposition 5. *If the nonfunction part is removed from program source codes, the newly produced program source codes are the refinement of the previous program source codes.*

```
Step 1. IniStack(S); // initiate a stack. Here, S represents stack.
Step 2. Input file.c; // input a file (protocol source codes)
Step 3. While (judge whether the file of protocol source codes is finished)
            {if (judge whether functions are true) Push(S, f); /*functions push into the
             stack. Here, f represents file. c.*/
              Continue to seek the functions of protocol source codes.
             } //while
Step 4. While (! StackEmpty(S)) // judge whether the stack is empty.
            { GetTop(S, f); // pop the functions which are on the top of the stack.
              if (the top functions of the stack satisfy protocol specifications) /*the functions
                can reflect the code execution of protocol interactive communication, such as
                send ( ) and recv ( ) of socket API. */
                {identify and reserve the functions;}
              Else {delete the functions;}
              } //while
```

Algorithm 1: The protocol source code refinement algorithm (written in C pseudocode).

Proof. Assume that the program running is a LTS. That is, let the LTS of the previous program source codes be (Q, T), and let the LTS of the program source codes obtained after removing the nonfunction part be (Q', T'). Let S be a binary relation over $Q \times Q'$. And there exist $Q' \subseteq Q$, $q, q' \in Q$, and $p, p' \in Q'$. Here α and α', respectively, are the input labels of two program source codes, and $\alpha = \alpha'$ (called α in the following). According to Definition 1, there exist $p \xrightarrow{\alpha} p' \in T$ and $q \xrightarrow{\alpha} q' \in T'$, as well as qSp and $q'Sp'$. According to Definition 3, the behaviors generated by LTS (Q', T') are the subset of LTS (Q, T), and (Q', T') will never produce any behaviors that (Q, T) cannot produce. That is, wherever (Q, T) is applied, it can be replaced by (Q', T'). Hence, the proposition is proved. □

4. Establishing the Model

There are always some faults in a protocol due to its imperfect design. These faults make the protocol vulnerable to malicious third-party attacks. For example, the Needham-Schroeder protocol and the Diffie-Hellman protocol cannot resist man-in-the-middle attacks. To avoid these malicious attacks, it is necessary to analyze the security of a protocol on theory level before its implementations (such as formal, computational model, the computationally sound formal). However, a protocol is not secure when implementing it at the source code level, although it has been proved to be secure on theory level. That is, it is essential to analyze the security of protocol implementations at the source code level.

In this paper, a method is proposed by us to analyze the security of protocol implementations at the source code level written in C. Our method establishes models in the following steps: ① describe a protocol symbolically; ② acquire the program source codes of the protocol; ③ refine these source codes; ④ draw the control-flow graph and the state diagram of the protocol; ⑤ establish the model of the traces of the protocol implementations. We take the classic Needham-Schroeder protocol, for example, to show how to establish a model with our new method.

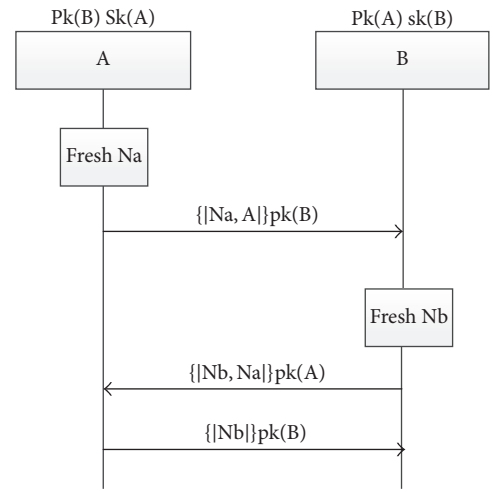

Figure 2: The Needham-Schroeder protocol.

4.1. The Protocol Symbolic Description. It is a basic requirement of protocol communications that at least two participants participate in an interactive communication in accordance with certain specifications. Generally, Message Sequence Charts (MSC) of protocol interactive communications are expressed by an ITU-standardized protocol specification language (ITU: International Telecommunication Union) [27] (in this paper, protocol specifications are expressed by the approaches from ITU and literature [28]). The symbolic description of the Needham-Schroeder protocol specifications is shown in Figure 2.

In Figure 2 A and B, respectively, denote two participants of a protocol; pk(A) and pk(B), respectively, denote the public keys of the participants; sk(A) and sk(B), respectively, denote the private keys of the participants; Fresh Na and Fresh Nb, respectively, denote temporary values of A and B; and $\{|\text{Na, A}|\}_{\text{pk(B)}}$ denotes the encryption of information {|Na, A|} with the public keys of B.

4.2. The Refinement of Source Codes. Taking the Needham-Schroeder source codes written in C, for example, we illustrate the refinement of source codes with Algorithm 1 and then analyze the security of protocol implementations at the source code level. When writing protocol codes, we exploit the mechanism of RAS public key cryptography for encryption and decryption, and the functions of the OpenSSL crypto library are used while encrypting and decrypting. The protocol runs over OpenSSL. Due to the limited length of this paper, only main source codes are shown in Algorithm 2.

As is shown in Algorithm 2, the source codes of the Needham-Schroeder protocol include many redundant ones. According to Proposition 5 and Algorithm 1, the security analysis of protocol implementations will not be influenced if those redundant codes are deleted. After deletion, the left source codes of the Needham-Schroeder protocol are mainly function codes (shown in Algorithm 3). According to Definition 4, the source codes in Algorithm 3 are the refinement of the source codes in Algorithm 2. In the example of the Needham-Schroeder protocol implementations, the behaviors produced by the source codes in Algorithm 3 are the subset in those produced by the source codes in Algorithm 2. Therefore, we can exploit the traces of function return values produced by the source codes in Algorithm 3 to dynamically analyze the security of the Needham-Schroeder protocol implementations at the source code level.

According to Definition 4 and Proposition 5, the security analysis of protocol implementations after refinement is consistent with that before refinement.

4.3. Program Control-Flow Graph. To clearly see the traces of the function return values generated during the protocol implementations, taking the source codes gained after the refinement (in Algorithm 3), for example, we draw a program control-flow graph to show the process of calling functions after the refinement, shown in Figure 3.

In Figure 3, ➜ denotes the control-flow direction of program main functions; "→" denotes the control-flow direction of calling functions; and "⇠" denotes the control-flow direction of function returning.

As is shown in the program control-flow graph, there are only two types of function return values: ① deterministic return values (like numerical values, alphabets, symbols, etc.); ② nondeterministic return values (such as calling functions directly or indirectly). When function return values are not deterministic, functions will continue to call other functions until the function return values become deterministic.

4.4. From the Control-Flow Graph to the State Graph. From Figure 3, we can clearly see the traces of the protocol implementations at the source code level and the state of how functions are called. According to C language grammar and program executive rules, in the normal process of program implementations every called function has a return value (deterministic or nondeterministic) and the implementation of every function is related to its return value. When a function receives its return value, the program will execute next step in order. Every function can be regarded as a state node, and function return values can be regarded as input labels. In this case, the program control-flow graph is just like an automata state graph. As mentioned above, function return values are clarified into deterministic values (like numbers, symbols, character strings, etc.) and nondeterministic values (such as functions). Here, we use {identify} to denote a deterministic value set and {unidentify} to denote a nondeterministic value set. Then after refinement, the control-flow graph of source codes is transferred into the state graph of a LTS, whose input label is $\{\text{identify, unidentify}\}^*$. A LTS has no initial state and receiving state. It only has starting point and terminal point, and any state can be regarded as its starting state. Hence, let function Gene_Rand( ) be the starting point of a LTS and let Send( ) function be its terminal point. The state graph of the Needham-Schroeder protocol is obtained after refinement, shown in Figure 4.

In Figure 4 "S" denotes a state, and its subscript denotes the state of its corresponding function point. For example, S_{My} denotes the state of memcpy( ) function. If a called function does not call other functions any more, its return value is deterministic, and {identify} is written to denote its input label. If a called function continues to call other functions, then its return value is nondeterministic, and {unidentify} is written to denote its input label. Here, "unidentify = identify|unidentify" can be used to denote the constitution of nondeterministic input labels. Its final return value is deterministic {identify}. For example, the return value of the state S_{GR} is not deterministic at first, for it calls other functions, like BN_new( ) function. Then, the input label of the state S_{GR} is {unidentify}.

4.5. The Traces of Protocol Implementations. As is shown in Section 4.4, a LTS can be established on the base of the function return values in the program control-flow graph. A LTS is denoted by a 4-tuple $(S, L, \rightarrow, S_0)$. We define the LTS of protocol implementations as follows:

(1) S denotes a function node state set; S_0 denotes the starting point state of the LTS; and S_e denotes the terminal point state of the LTS.

(2) S_i denotes other functions, and there exists $S_i \in S$. $S_i \xrightarrow{\alpha} S_{i+1}$ denotes the transition relation between any two adjacent states S_i, S_{i+1}. α denotes an input label. Let $\alpha = \{\text{identify|undentify}\}^*$. If α is unidentify, there exists a substitute function σ, and $\sigma(\alpha) = \{\text{identify}\}$. Then final function return value is deterministic.

According to the LTS of protocol implementations, after protocol implementations there exist the traces of function return values: traces $= \{\text{identify}_1, \text{identify}_2, \ldots, \text{identify}_n\}$ $n \in N$. Then the security of protocol implementations at the source code level can be dynamically analyzed on the base of these traces.

5. The Security Analysis of Protocol Implementations

Functions are an important part of the source codes of a program design language, especially C language. If the

```
1int main(int,argc,char argv[ ])
2{
3WSADATA wsaData;
4SCOKET client;
5RSA ∗R;
6serv.sin_family=AF_INET;
7serv.sin_port=htons(port);
8serv.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
9client=socket(AF_INET,SOCK_STREAM,0);
10connect(client,(struct sockaddr∗)&serv,sizeof
(serv));
11Gene_Rand( );
12Comp_Str( );
13memcpy(plaitxt_A_E,Str_Id,1024);
14public_Encrypt(plaitxt_A_E);
15strcpy(Cipher_buf,(const char∗)cipher);
16iSend=send(client,Cipher,sizeof(Cipher),0);
17iLen=recv(client,(char ∗)cipher,sizeof(cipher),0);
18memcpy(Tcipher_Na_Nb,cipher_Na_Nb,1024);
19Private_Dencrypt_Na_Nb(R,Tciper_Na_Nb);
20memcpy(plaitxt_Nb,StrNb,1024);
21Public_Encrypt_Nb(plaitxt_Nb);
22strcpy(Cipher,(const char∗)cipher);
23iSend=send(client,Cipher,sizeof(Cipher),0);
 24}
25int Gene_Rand( )
26{
27BIGNUM ∗n;
28int ret,bits=128;
29char ∗sn;
30n=BN_new( );
31ret=BN_pseudo_rand(n,bits,1,1);
32sn=BN_bn2dec(n);
33strcpy(RandNum,sn);
34return 0;
 35}
36int Com_Str( )
37{
38strcat(Str_Id,RandNum);
39strcat(Str_Id,"|A");
40return 0;
 41}
42int Public_Encrypt(unsigned char∗s)
43{
44RSA ∗r;
45int ret,flent,len;
46BIGNUM ∗bnn,∗bne;
47bnn=BN_new( );
48bne=BN_new( );
49ret=BN_dec2bn(&bnn,strn);
50ret=BN_dec2bn(&bnn,stre);
51r=RSA_new( );
52r->n=bnn;
53r->e=bne;
54flen=RSA_size(r);
55len=RSA_public_encrypt(flen,s,cipher_A_E,r,3);
56return 0;
 57}
58int Private_Dencrypt_Na_Nb(RSA ∗r,unsigned char∗s)
59{
60int len,flen;
```

Algorithm 2: Continued.

```
61flen=RSA_size(r);
62len=RSA_private_decrypt(flen,s,plaitxt_Na_Nb, r,3);
63return 0;
64}
65int StrDivNb(char str[ ])
66{
67int k=0,i=0,j=0;
68int strlen=0;
69int Publi_Encrypt_Nb(usigned char*s)
70{
71RSA *r;
72int ret,flen,len;
73BIGNUM *bnn,*bne;
74bnn=BN_new( );
75bne=BN_new( );
76ret=BN_dec2bn(&bnn,strn);
77ret=BN_dec2bn(&bne,strn);
78r=RSA_new( );
79r->n=bnn;
80r->e=bne;
81flen=RSA_size(r);
82len=RSA_public_encrypt(flen,s,cipher_Nb,r,3);
83return 0;
84}
```

Algorithm 2: The part source codes of the Needham-Schroeder protocol.

behaviors of called program functions are regarded as the events of a protocol implementation, function return values can act as the conditions for running an event during the process of protocol implementations. In this paper, operational semantics are exploited to analyze the behaviors' security of protocol implementations at the source code level.

5.1. The Operational Semantics of Function Return Values. Operational semantics clearly display the traces of function return values of a protocol implementation. That is, they display concrete behaviors of protocol implementations. Therefore, operational semantics have an advantage over other methods and are competent to analyze the security of protocol implementations.

(1) The function return values of protocol implementations is as follows.

$F = P(\text{RunFunc}) \times P(\text{Run}) \times \text{ReturnValue}$. Here P(RunFunc) denotes the functions of a protocol, P(Run) denotes running it, and ReturnValue denotes its function return values.

(2) The BNF forms of function return values (ReturnValue) during protocol implementations are as follows:

ReturnValue ::= Identity|Unidentify

Unidentify ::= Function|Identify

Function := Self_Func|Libr_Func

Self_Func := Function|Identify

Libr_Func := Function|Identify

Identify := number|alphabet|string

Here, Self_Func denotes self-defined functions and Libr_Func denotes library functions. Every function has a return value (deterministic or nondeterministic). Every function return value is an element of a trace (ReturnValue $\in$ Traces). After a protocol is implemented, all the function return values constitute its traces. Here is the definition of the BNF form of the traces:

Traces ::= number|alphabet|symbol

According to the relation between called function events of C language and return values, every called function has a return value, including void values. Then there exists the bijection M between called function events and return values: CallEvent|$\rightarrow$ReturnValue. According to the traces obtained after the protocol implementations, $\forall$ReturnValue $\in$ Trace is true. Therefore, there exists the bijection: $\{e|\rightarrow t, | \forall e \in \text{CallEvent}, \forall t \in \text{Trace}\}$. That is, every event corresponds to an element of the traces of protocol implementations.

Definition 6 (event trace). Let every event ε of protocol implementations correspond to a function return value ν. There exists a set t, which consists of all the function return values. The set t denotes an event trace.

Once the bijection between called function events of protocol implementations and function return values is set up, there exists the bijection between the event set and the elements of the traces: $\text{Trace} = \{t \mid \forall e \mid\rightarrow \forall \nu \in t\}$. If the event is not secure, its trace is not secure either, which will inevitably lead to the insecurity of the protocol implementations.

```
1int main(int argc,char argc[ ])
2{
3connect(client,(struct sockaddr *)&serv,sizeof(serv));
4Gene_Rand( );
5Comp_Str( );
6memcpy(plaitxt_A_E,Str_Id,1024);
7Public_Encrypt(plaitxt_A_E);
8strcpy(Cipher_buf,(const char *)cipher);
9iSend=send(client,Cipher,sizeof(cipher),0);
10iLen=recv(client,(char *)cipher,sizeof(cipher),0);
11memcpy(Tcipher_Na_Nb,cipher_Na_Nb,1024);
12Private_Dencrypt_Na_Nb(R,Tcipher_Na_Nb);
13memcpy(plaitxt_Nb,StrNb,1024);
14Public_Encrypt_Nb(plaitxt_Nb);
15strcpy(Cipher,(const char *)cipher);
16iSend=send(client,Cipher,sizeof(Cipher),0);
 17}
18int Gene_Rand( )
19{
20n=BN_new( );
21ret=BN_pseudo_rand(n,bits,1,1);
22sn=BN_bn2dec(n);
23strcpy(RandNum,sn);
 24}
25int Comp_Str( );
26{
27strcat(Str_Id,RandNum);
28strcat(Str_Id,"|A");
 29}
30int Public_Encrypt(unsigned char *s)
31{
32bnn=BN_new( );
33bne=BN_new( );
34ret=BN_dec2bn(&bnn,strn);
35ret=BN_dec2bn(&bne,stre);
36r=RSA_new( );
37flen=RSA_size(r);
38len=RSA_public_encrypt(flen,s,cipher_A_E,r,3);
 39}
40int Private_Dencrypt_Na_Nb(RSA *r,unsigned char *s);
41{
42flen=RSA_size(r);
43len=RSA_private_decrypt(flen,s,plaitext_Na_Nb, r,3);
 44}
45int Public_Encrypt_Nb(unsigned char *s)
46{
47bnn=BN_new( );
48bne=BN_new( );
49ret=BN_dec2bn(&bnn,strn);
50ret=BN_dec2bn(&bne,stre);
51r=RSA_new( );
52flen=RSA_size(r);
53len=RSA_public_encrypt(flen,s,cipher_Nb,r,3);
 54}
```

Algorithm 3: The refined part source codes of the Needham-Schroeder protocol.

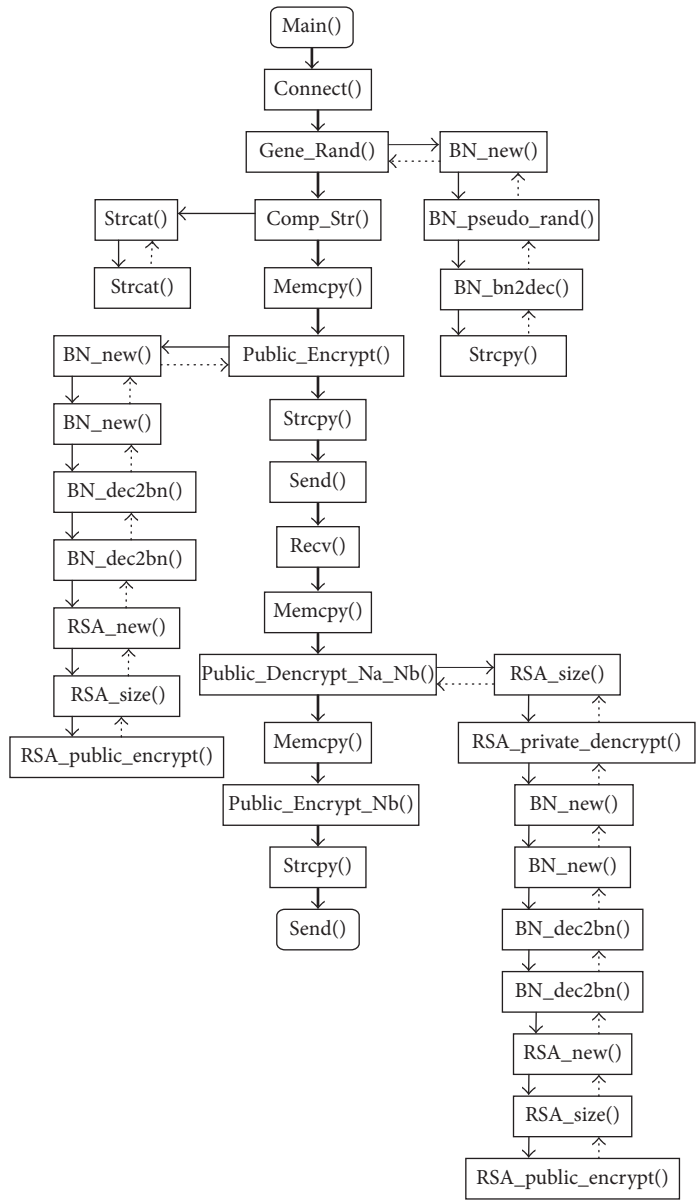

Figure 3: The program control-flow graph.

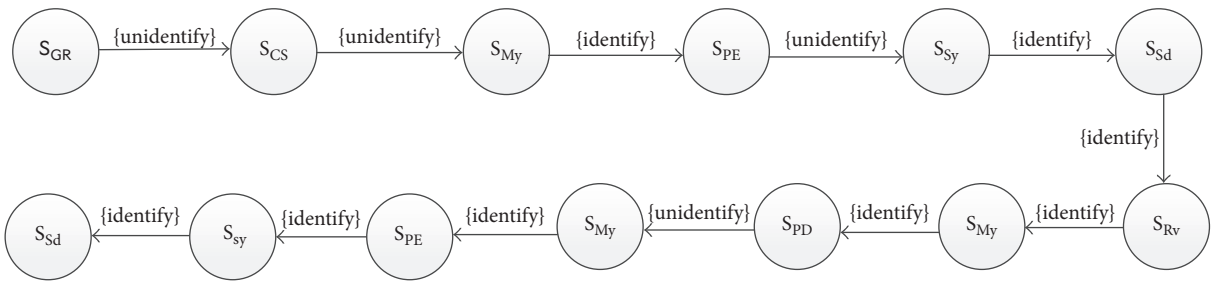

Figure 4: The LTS of part source codes.

5.2. Ideal Trace and Nonideal Trace. Generally, it depends on the Dolev-Yao model assumption to decide whether a protocol is secure or not. Equally, the trace of function return values is obtained on the base of the Dolev-Yao model assumption. We are the first to propose "the ideal trace" as the reference of protocol security evaluation for the precise security analysis of protocol implementations at the source code level.

In the following definitions, the symbol $\prec$ denotes the ordinal relation of called functions during the protocol implementations. As Definition 6 shows, $\prec$ denotes the ordinal relation of the events during the protocol implementations. $\prec$ is a binary relation over $e \times e$. $\prec$ satisfies partial ordering relation on event sequences $\{e_0, e_1, \ldots, e_n\}$: ① reflexivity (there exists $e \prec e$ for any event e); ② antisymmetry (if there exists $e_1 \prec e_2$ and $e_2 \prec e_1$ for any events e_1 and e_2, then there exists $e_1 = e_2$); ③ transmissibility (if there exists $e_1 \prec e_2$ and $e_2 \prec e_3$ for e_1, e_2, e_3, then there exists $e_1 \prec e_3$).

Definition 7 (nonideal trace). The protocols are implemented in the following environments:

(1) The environments are based on the Dolev-Yao model (adversaries have the capacity to attack actively or passively).

(2) The environments of implementing protocols are insecure.

① The environments are insecure due to the flaws of implementing program language structures, such as memory overflow and pointer operation.

② There are malicious code attacks in the environments.

The trace obtained in such environments is called a nonideal trace.

Definition 8 (ideal trace). The trace of general protocol implementations satisfies the following conditions.

(1) In an ideal communication environment (there are no adversary attacks in an ideal communication environment, passive attacks, or active attacks; i.e., there is no Dolev-Yao model assumption in the environment), all the participants of a protocol are honest. The information sent or received by these participants is protected and read by encrypting and decrypting techniques.

(2) Partial ordering relation: $e_0 \prec e_1 \prec e_2, \ldots, e_i \ \ 0 \le i \le n$. Here $\prec$ denotes that the events of protocol implementations satisfy partial order relation.

Generally, if a trace of protocol implementations satisfies both condition (1) and condition (2), it is called the ideal trace.

Definition 9 (the similarity between a nonideal trace and the ideal trace). Let α, β be ordered sequences, respectively, obtained in an ideal environment or a nonideal environment. According to the clustering method, the similarity between the ideal trace and a nonideal trace is defined as follows:

(1) Let the Euclidean distance between ordered sequences α and β be $d_{\alpha\beta}$.

(2) Let the similarity between ordered sequences α and β be $s_{\alpha\beta}$, and $s_{\alpha\beta} = \delta d_{\alpha\beta}^{-1}$. δ denotes the similarity coefficient.

The bigger $s_{\alpha\beta}$ is, the smaller the degree of deviation between α and β is. This shows that the similarity between the ideal trace and the nonideal trace is bigger, and vice versa.

According to Definitions 7, 8, and 9, the ideal trace consists of function return values obtained in an ideal environment. In practice, most communications are carried out in nonideal environments. If the trace produced in a nonideal communication environment deviates from the ideal trace, it indicates that the protocol is attacked by a third party. If the attacked protocol is improved, its trace will deviate less from the ideal trace. That is, after its improvement, the similarity between its trace and the ideal trace is bigger. It means that the improved protocol is more secure than the original protocol.

6. The Method and the Experiment

According to Algorithm 1, in the process of protocol implementations, source codes are refined. After that, labelled function return values are obtained and constitute the sequences. When analyzing the security of protocol implementations, we exploit the ideal trace as the reference of security evaluation. According to the clustering method, the sequences can be used as samples to analyze the security of protocol implementations at the source code level by means of the deviation of Euclidean distance and the similarity.

6.1. The Steps of Our New Method

(1) Supposing that source codes are executed in an ideal communication environment: after an execution, function return values are obtained and form a sequence. It is called the sequence of ideal trace data.

(2) Supposing that source codes are executed in a nonideal communication environment: after an execution, function return values are obtained and form a sequence. It is called the sequence of nonideal trace data.

(3) With the clustering method [29], the Euclidean distance [30] d_{ij} between the ideal trace and the nonideal trace is calculated. The following is the formula of the Euclidean distance:

$$d\left(x_i - x_j\right) = \left[\sum_{k=1}^{p}\left(x_{ik} - x_{jk}\right)^2\right]^{1/2}. \tag{1}$$

Let $d_{ij} = d(x_i, x_j)$, and let $D = (d_{ij})_{p\times p}$ be a distance matrix:

$$\begin{bmatrix} 0, d_{12}, \ldots, d_{1n-1}, d_{1n} \\ d_{21}, 0, \ldots, d_{2n-1}, d_{2n} \\ \vdots \\ d_{n1}, d_{n2}, \ldots, d_{nn-1}, 0 \end{bmatrix}. \tag{2}$$

Here $d_{ij} = d_{ji}$. If $d_{ij} = 0$, the nonideal trace does not deviate from the ideal trace.

Table 1: The trace of function return values of protocol implementations.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
INS (1)	80	1	136	216	144	176	53	5	192	128	128	128	128	208	1	168	216	16	48	53	1	192
MANS (2)	112	1	216	216	176	208	53	5	80	128	128	128	128	240	1	187	216	48	80	53	1	47
INSL (3)	80	1	136	216	144	176	53	5	192	128	128	128	128	208	1	175	216	16	48	53	1	200
MANSL (4)	112	1	240	216	176	208	53	5	80	128	128	128	128	240	1	178	216	48	80	53	1	147
	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
INS (1)	128	128	128	128	176	208	53	5	208	128	128	173	241	0	0	128	128	244	191	0	0	1
MANS (2)	128	128	128	208	208	240	53	5	128	128	128	128	128	0	0	0	1	216	0	0	216	0
INSL (3)	128	128	128	128	176	208	53	5	208	128	210	210	244	16	0	128	128	244	149	0	0	1
MANSL (4)	128	128	128	128	208	240	53	5	128	128	128	128	128	0	0	0	1	216	0	0	216	0

Table 2: Euclidean distance between protocols.

	INS	MANS	INSL	MANSL
INS	0	439	102	412
MANS	439	0	439	152
INSL	102	439	0	410
MANSL	412	152	410	0

(4) Supposing that, after implementing the protocols α and β, their sequences of trace data are obtained: let $s_{\alpha\beta}$ be the similarity between the ideal trance and a nonideal trance. The similarity between the sequence of α and the sequence of β is

$$s_{\alpha\beta} = \frac{1}{1 + d_{\alpha\beta}}. \quad (3)$$

Formula (3) satisfies the following: ① If $d_{\alpha\beta} \to \infty$, $s_{\alpha\beta} \to 0$; ② if $d_{ij} \to 0$, $s_{\alpha\beta} \to 1$.

Therefore, their similarity is inversely proportional to their deviation. That is, the smaller the similarity is, the easier the protocol implementation is attacked.

With our new method, when analyzing the security of protocol implementations, there are two cases: ① If $d_{ik} > d_{ij}$, the nonideal trace $\{t_k\}$ deviates from the ideal trace $\{t_i\}$ (d_{ik} denotes the deviation between the ideal trace $\{t_i\}$ and a nonideal trace $\{t_k\}$). It means that the protocol is attacked during implementations. ② The bigger $s_{\alpha\beta}$ is, the closer the trace of the protocol α gets to the trace of the protocol β. It means that the protocol implementations are more secure.

6.2. The Experiments. We carry out experiments with classical protocols and their improvements (written in C). In the experiments, we analyze the cases in which these protocols are attacked by man-in-the-middle attacks during implementations. The running environment is Win7, Visual studio 2010, Intel(R) CPU G3240, memory 4 GB, openssl-1.0.1s. The protocols run by using the functions and the big number in OpenSSL function library. The data of the protocols are encrypted and decrypted with the mechanism of the RSA public key. Participants communicate by linking TCP of Socket API. Simulate experiments are carried out with the pattern of client/server. The function return values are transformed into numerical values and then used as experiment data, which will not influence the result of the security analysis of protocol implementations.

(1) We analyze man-in-the-middle attacks of the Needham-Schroeder protocol implementations and the Needham-Schroeder-Lowe protocol implementations.

Experiments are carried out in two types of environments: ① the ideal environment and ② nonideal environments. In the ideal environment, the Needham-Schroeder protocol is called INS for short and the Needham-Schroeder-Lowe protocol is called INSL for short. In nonideal environments, the Needham-Schroeder protocol is called MANS for short and the Needham-Schroeder-Lowe protocol is called MANSL for short. NSL is the improvement of the Needham-Schroeder protocol. Theoretically, NSL can resist man-in-the-middle attacks. After running these protocols, their traces are obtained, shown in Table 1.

The illustration of the data in Table 1: ① the numbers in the first line are the serial numbers of the functions of the protocol implementations; ② the first row corresponds to the names of the protocols and their serial numbers; ③ the values at the intersections are the function return values of each protocol implementation. For example, INS (1) (5) = 144 means that, during the implementation of INS, the fifth function return value is 144.

As is shown in Table 1, the traces of the Needham-Schroeder protocol implementations and of the Needham-Schroeder-Lowe protocol implementations deviate from the traces of their implementations in the ideal environments. According to formula (1), we, respectively, calculate each Euclidean distance d_{ij} between INS and MANSL, INSL and MANSL, and MANS and MANSL. $d_{ij} = d_{ji}$, and $i, j \in N, 1 \leq N \leq 44$. The results of calculation are shown in Table 2.

According to formula (2), we, respectively, calculate each similarity between INS and MANSL, INSL and MANSL, and MANS and MANSL. The results are shown in Table 3.

Table 3: Similarity between protocols.

	INS	MANS	INSL	MANSL
INS	1	2.27×10^{-3}	9.71×10^{-3}	2.41×10^{-3}
MANS	2.27×10^{-3}	1	2.27×10^{-3}	6.54×10^{-3}
INSL	9.71×10^{-3}	2.27×10^{-3}	1	2.43×10^{-3}
MANSL	2.41×10^{-3}	6.54×10^{-3}	2.43×10^{-3}	1

Table 4: The traces of function return values of protocol implementations.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
IDH (1)	1	1	32	1	1	0	0	77	32	128	136	0	1	1	32	1	1	0	77	32	69	8
MADH (2)	0	1	1	32	1	1	0	77	32	40	8	1	1	32	1	1	0	0	77	32	112	40
IDHS (3)	1	1	32	1	1	0	0	77	32	112	104	0	1	1	32	1	1	0	77	32	112	40
MADHS (4)	0	1	1	32	1	1	0	77	32	114	120	1	1	32	1	1	0	0	77	32	144	142

Table 5: Euclidean distance between protocols.

	IDH	MADH	IDHS	MADHS
IDH	0	176	64	167
MADH	176	0	135	187
IDHS	64	135	0	129
MADHS	167	187	129	0

Table 6: The similarity between protocols.

	IDH	MADH	IDHS	MADHS
IDH	1	5.67×10^{-3}	1.529×10^{-2}	5.59×10^{-3}
MADH	5.67×10^{-3}	1	7.40×10^{-3}	5.33×10^{-3}
IDHS	1.529×10^{-2}	7.40×10^{-3}	1	7.71×10^{-3}
MADHS	5.59×10^{-3}	5.33×10^{-3}	7.71×10^{-3}	1

As is shown in Table 2, $d_{12} = 439$ means that the trace of INS deviates from the trace of the MANS protocol implementations; $d_{34} = 410$ means that the trace of INSL deviated from the trace of MANSL. As is shown in Table 3, $2.43 \times 10^{-3} > 2.27 \times 10^{-3}$ means that the similarity between INSL and MANSL is bigger than the similarity between INS and MANS.

(2) We analyze man-in-the-middle attacks of the Diffie-Hellman protocol implementations and the Diffie-Hellman-Signature protocol implementations.

The protocols are carried out in two types of environments: ① the ideal environments and ② nonideal environments. In the ideal environment, Diffie-Hellman is called IDH for short and Diffie-Hellman-Signature is called IDHS for short; in nonideal environments, Diffie-Hellman is called MADH for short and Diffie-Hellman-Signature is called MADHS for short. DHS is the improvement of the Diffie-Hellman protocol and, theoretically, it can resist man-in-the-middle attacks. After running these protocols, their traces are obtained, shown in Table 4.

According to formula (1), we, respectively, calculate each Euclidean distance d_{ij} between IDH and MADHS, IDHS and MADHS, and MADH and MADHS. $d_{ij} = d_{ji}$, and $i, j \in N, 1 \leq N \leq 22$. The results of calculation are shown in Table 5.

According to formula (2), we, respectively, calculate the similarity between IDH and MADHS, IDHS and MADHS, and MADH and MADHS. The results are shown in Table 6.

As is shown in Table 5, $d_{12} = 176$ means that the trace of IDH deviates from the trace of MADH; $d_{34} = 129$ means that the trace of IDHS deviates from the trace of MADHS. As is shown in Table 6, $7.71 \times 10^{-3} > 5.67 \times 10^{-3}$ means that the similarity between IDHS and MADHS is bigger than the similarity between IDH and MADH.

(3) We analyze the performance of our new method.

To illustrate the performance of our new method, we only list the time overhead of each protocol implementation in the ideal environment and nonideal environments (shown in Table 7) and discuss the relation between the time overhead and the similarity (shown in Figures 5 and 6). Due to the limited length of this paper, we only list and discuss the time overhead and the similarity of each protocol implementation in the simulated experiments mentioned above.

As is shown in Table 7, the time overhead of implementing protocols in the ideal environment is less than the time overhead of implementing protocols in the nonideal environments.

We calculate the absolute values $|\Delta T|$ of the difference of the time between implementing protocols in the ideal environment and implementing protocols in the nonideal

Table 7: The time overhead of implementing protocols.

Name	IDH	IDHS	MADH	MADHS	INS	INSL	MANS	MANSL
Time	0.916	0.789	1.567	0.132	0.037	0.030	0.100	0.096

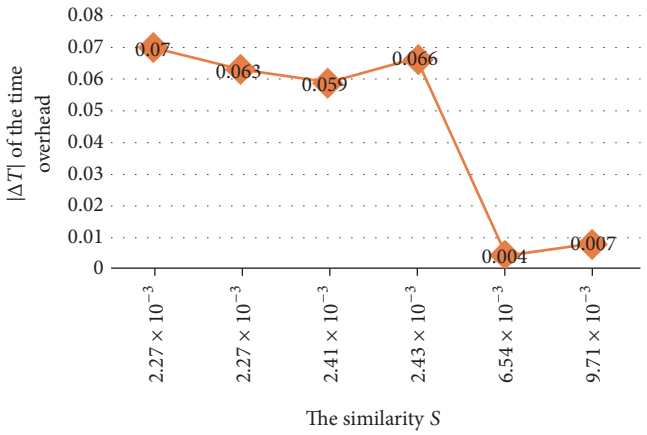

Figure 5: The change of similarity and $|\Delta T|$ of the time overhead.

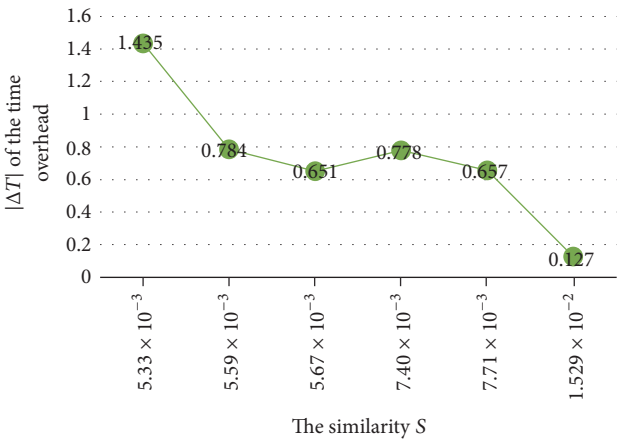

Figure 6: The change of similarity and $|\Delta T|$ of the time overhead.

environments. We list $|\Delta T|$ of the time overhead and $S_{\alpha\beta}$ of the Needham-Schroeder protocol and its improvement (shown in Table 3), as well as $|\Delta T|$ of the time overhead and $S_{\alpha\beta}$ of Diffie-Hellman protocol and its improvement (shown in Table 6). By comparing their changes, we can see their relation clearly, shown in Figures 5 and 6.

This shows that the relation between $S_{\alpha\beta}$ and $|\Delta T|$ tends to be in inverse proportion. It means that $S_{\alpha\beta}$ is related to the performance of protocol implementations in different environments.

As is shown in experiments (1), (2), and (3), the security of protocol implementations can be analyzed through the traces of function return values obtained when implementing the protocols in the ideal environment and in the nonideal environments. From the experiments, we can conclude the following:

① Third-party attacks can be found.

② The deviation of the improved protocols is smaller than the deviation of the original ones. This means that the improved protocols are more secure than the original ones in the process of implementations at the source code level.

③ The protocols are insecure during implementations at the source code level, even though they are proved to be secure on theory level.

④ In the ideal environment and nonideal environments, the performance of protocol implementations is related to the similarity. It is in accordance with the security thought defined by software.

This paper aims to analyze the security of protocol implementations at the source code level. With our method, attacks can be discovered by analyzing whether there are abnormal behavior characteristics when implementing protocols, such as memory overflow and malicious code (shellcode) attacks. This paper mainly analyzes man-in-the-middle attacks.

Man-in-the-middle attacks are a common type of attacks which protocols widely suffer from [31]. It is typical to analyze the security of protocol implementations which are attacked by man-in-the-middle attacks. The thought of our new method comes from life phenomena and the theory is on the base of LTS, program operational semantics, and program refinement. And we propose Algorithm 1. The purpose of analysis is to discover whether a protocol is attacked by third-party attacks, impersonation attacks, or other attacks during the implementation at the source code level. We discover that the similarity of the improved protocols, which can resist man-in-the-middle attacks, is bigger than that of the original ones. Hence, our method is competent to analyze the protocols attacked by man-in-the-middle attacks during the implementations at the source code level.

7. Conclusion and Future Work

In the paper, we have proposed a new method to dynamically analyze the security of protocol implementations at the source code level. First, we have refined protocol source codes through strong simulation relation and have established a model for the security analysis of protocol implementations. Second, when implementing protocols, we have obtained the function return values, which constitute the sequences of trace data. We propose the ideal trace as the evaluating reference for protocol security analysis. The clustering method is exploited to analyze the sequences of trace data. We propose to exploit the deviation of the Euclidean distance between the ideal trace and nonideal traces and the similarity between the ideal trace and nonideal traces to analyze the security of protocol implementations. Last, taking some classical protocols, for example, we have carried out experiments. Our experiments show that third-party attacks can be found by analyzing the deviation and similarity of the traces of

the function return values obtained when implementing protocols. It is also shown through our experiments that the improved protocols are more secure than the original ones. Our method exploits the traces of protocol implementations to analyze its security and differs from other methods mentioned in the literature. It will be helpful and valuable for protocol design, security verification, and security evaluation.

The future study in this field lies in the automatic security analysis of protocol implementations [32] and the security analysis of parallel protocol implementations [33]. It requires improving existing protocol automatic analysis tools or developing new automatic tools for the security analysis of protocol implementations. On the other hand, since the flaws of language structures will make the security analysis of protocol implementations more complex, more studies should be carried out to solve these complex problems, such as C language memory access and pointer analysis. This field should be focused on.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Natural Science Key Foundation of China (no. 61332019) and the National Basic Research Program (NBRP) (973 Program) (no. 2014CB340601),

References

[1] H. Zhang, W. Han, X. Lai, D. Lin, J. Ma, and J. Li, "Survey on cyberspace security," *Science China Information Sciences*, vol. 58, no. 11, pp. 1–43, 2015.

[2] M. Asadzadeh Kaljahi, A. Payandeh, and M. B. Ghaznavi-Ghoushchi, "TSSL: Improving SSL/TLS protocol by trust model," *Security and Communication Networks*, vol. 8, no. 9, pp. 1659–1671, 2015.

[3] M. Backes, M. Maffei, and D. Unruh, "Computationally sound verification of source code," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, (CCS '10)*, pp. 387–398, ACM, New York, NY, USA, October 2010.

[4] J. Goubault-Larrecq and F. Parrennes, "Cryptographic protocol analysis on real C code," in *Verification, model checking, and abstract interpretation*, vol. 3385 of *Lecture Notes in Computer Science*, pp. 363–379, Springer, Berlin, Germany, 2005.

[5] S. Chaki and A. Datta, "ASPIER: An automated framework for verifying security protocol implementations," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, (CSF '09)*, pp. 172–185, IEEE, Port Jefferson, NY, USA, July 2009.

[6] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *Proceedings of the the 18th ACM conference on Computer and communications security (CCS '11)*, pp. 331–340, ACM, Chicago, Illinois, USA, October 2011.

[7] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 712–723, ACM, Raleigh, North Carolina, USA, October 2012.

[8] J. Almeida, E. Bangerter, M. Barbosa, K. Stephan, S. Ahmad-Reza, and T. Schneider, "A certifying compiler for zero-knowledge proofs of knowledge based on Σ- protocols," in *European Symposium on Research in Computer Security*, vol. 6345 of *Lecture Notes in Computer Science*, pp. 151–167, Springer, Berlin, Germany, 2010.

[9] S. Kiyomoto, H. Ota, and T. Tanaka, "A security protocol compiler generating C source codes," in *Proceedings of the 2nd international conference on information security and assurance, (SA '08)*, pp. 20–25, Busan, South Korea, IEEE, April 2008.

[10] C. Sarah Meiklejohn and C. Erway, "ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash," *USENIX Conference on Security*, pp. 193–206, 2010.

[11] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '10)*, vol. 45, pp. 445–456, ACM, Madrid, Spain, January 2010.

[12] C. Sprenger and D. Basin, "Refining key establishment," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium, (CSF '12)*, pp. 230–246, IEEE, Cambridge, MA, USA, June 2012.

[13] R. Milner, *Communicating and mobile systems:the π-calculus*, Cambridge University Press, Cambridge, UK, 1999.

[14] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: a survey," *Formal Aspects of Computing*, vol. 26, no. 1, pp. 99–123, 2014.

[15] A. Yasinsac and J. Childs, "Formal analysis of modern security protocols," *Information Sciences*, vol. 171, no. 1-3, pp. 189–211, 2005.

[16] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 256–267, Denver, Colorado, USA, October 2015.

[17] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Engineering Secure Software and Systems*, vol. 6542 of *Lecture Notes in Computer Science*, pp. 58–72, Springer, Berlin, Germany, 2011.

[18] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," *Journal of Computer Security*, vol. 22, no. 5, pp. 823–866, 2014.

[19] F. Dupressoir, *Proving C Programs Secure with General-Purpose Verification Tools [Ph.D. thesis]*, thesis Open University, 2013.

[20] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *Proceedings of the 28th IEEE Computer Security Foundations Symposium, (CSF '15)*, pp. 512–525, IEEE, Verona, Italy, July 2015.

[21] C. Fournet, C. Keller, and V. Laporte, "A certified compiler for verifiable computing," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium, (CSF '16)*, IEEE, Lisbon, Portugal, July 2016.

[22] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy, (EURO S&P)*, pp. 112–127, IEEE, Saarbrucken, Germany, March 2016.

[23] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud et al., "A messy state of the union: taming the composite state machines of TLS," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, (SP '15)*, pp. 535–552, San Jose, Calif, USA, May 2015.

[24] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 941–951, ACM, Denver, Colorado, USA, October 2015.

[25] R. Brooks, B. Husain, S. Yun, and J. Deng, "Security and performance evaluation of security protocols," in *Proceedings of the 8th Annual Cyber Security and Information Intelligence Research Workshop: Federal Cyber Security R and D Program Thrusts, (CSIIRW '13)*, ACM, Oak Ridge, Tennessee, USA, January 2013.

[26] E. M. Clarke, "My 27-year Quest to Overcome the State Explosion Problem," in *Proceedings of the 24th Annual IEEE Sympon Longic in Computer Science (LICS '09)*, IEEE, Los Angeles, Calif, USA, August 2009.

[27] "ITU-TS, Recommendation Z.120:Message Sequnce Chart (MSC)ITU-TS, Genva (1999)".

[28] Cas Cremers, *Sjouke Mauw.Operational Semantics and Verification of Security Protocols*, Springer, berlin, Germany, 2012.

[29] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley & Sons, Canada, 2005.

[30] M. M. Deza and E. Deza, *Encyclopedia of distances*, Springer-Verlag, Berlin, Germany, 2009.

[31] M. Conti, N. Dragoni, and V. Lesyk, "A survey of man in the middle attacks," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 3, pp. 2027–2051, 2016.

[32] S. A. Menesidou, D. Vardalis, and V. Katos, "Automated key exchange protocol evaluation in delay tolerant networks," *Computers and Security*, vol. 59, pp. 1–8, 2016.

[33] A. C. Yao, M. Yung, and Y. Zhao, "Concurrent knowledge extraction in public-key models," *Journal of Cryptology. The Journal of the International Association for Cryptologic Research*, vol. 29, no. 1, pp. 156–219, 2016.