Automatic Generation of Security Protocol Implementations Written in Java from Abstract Specifications Proved in the Computational Model

Bo Meng[1], Chin-Tser Huang[2], Yitong Yang[1], Leyuan Niu[1], and Dejun Wang[1]
(Corresponding author: Bo Meng)
School of Computer, South-Central University for Nationalities, Wuhan 430074, China[1]
182, Minyuan Road, Hongshan District, Wuhan, Hubei Province 430074, P.R.China
mengscuec@gmail.com
Department of Computer Science and Engineering, University of South Carolina, Columbia 29208, USA[2]
huangct@cse.sc.edu
(Email: mengscuec@gmail.com)
(Received Dec. 7, 2015; revised and accepted Feb. 18 & March 25, 2016)

Abstract

In order to deploy security protocols in practice, it is highly important that they be implemented in a secure manner. In this study, we first introduce a model for code generation from abstract security protocol specifications. Then, we present the development of an automatic generator called CV2JAVA, which is able to translate a security protocol abstract specified in Blanchet calculus in the computational model into an implementation written in JAVA. Finally, we also use the automatic generator CV2JAVA and CryptoVerif to generate the Secure Shell Version 2(SSHV2) security protocol implementation written in JAVA from the SSHV2 security protocol implementation written in Blanchet calculus proved in the computational model.

Keywords: Code generation, code security, protocol security, security protocol

1 Introduction

Over the last twenty years, researchers including developers and users have worked a great deal on the analysis and verification of security protocol abstract specifications [10, 17]. However, we know that the final objective is to have the security protocol implemented using programming languages, for example, the Java language, to put it into practice in the information system. Hence, we need to research the methods for generating the codes for security protocols because proof of the security properties of security protocol abstract specifications is not enough to instill confidence in developers and users with regard to the degree of security during execution. In addition, even if we prove the security properties of security protocol abstract specifications, their implementation procedures may be error prone and insecure. Therefore, it is essential to analyze and prove the security properties of security protocol implementations.

Generally, there are two major approaches to implement security protocols. One approach is suitable for the legacy codes of security protocols. This approach can be divided into two types. One type is mainly based on the technologies of program analysis and verification, for example, logic and type theory, which are directly used to automatically verify cryptographic security. In addition, it also depends on adding many annotations and predicates/assertions to the security protocol implementations. The other type is model extraction, which first extracts the security protocol abstract specification from security protocol implementations and then uses the security protocol analyzer to analyze or prove the cryptographic security of security protocol implementations. The drawback of this approach is that it is difficult to implement correctly.

The other approach is code generation, which is suitable for obtaining new implementations of some security protocols. If we have developed the security protocol abstract specification and verified its security properties, we can use the code generation method to obtain the secure security protocol implementations written in programming languages. In the code generation method, the first step is to produce the security protocol abstract specifications represented in formal language, for example, by using pi calculus or probabilistic process calculus. Then, a

security protocol analyzer/prover, for example, ProVerif or CryptoVerif, is used to analyze the securities of the security protocol abstract specification. Finally, a transformer is developed to transform the security protocol abstract specifications into the security protocol implementation.

Two types of models, namely, the symbolic model and computational model, have been proposed and applied for code generation. There have been some existing approaches based on the symbolic model, including [16, 3, 11, 14, 8, 15, 2, 9, 1, 18, 13], which will be reviewed and discussed in section 2, the related work section. The symbolic model and the computational model are complementary ways for security analysis of security protocols. However, we know that one limitation of the symbolic model is that it cannot instill confidence in users with regard to its security because in the symbolic model, the cryptographic primitives are highly abstracted as black boxes, which often result in some attacks being missed. In contrast, the computational model is based on complexity and probability and uses a strong notion of security to counter all probabilistic polynomial-time attacks; therefore, its security properties are practically verifiable and it can find some attacks missed in the symbolic model. Some existing schemes [5, 6, 7] using the computational model are introduced in the related work section. However, to the best of our knowledge, none of the existing schemes can achieve mechanized generation of security protocol implementations in Java from security protocol abstract specifications proved in the computational model. Thus, in this study, we choose to use the computational model and focus on the mechanized generation of security protocol implementation in Java from security protocol abstract specifications proved in the computational model.

The main contributions of this study are summarized as follows:

1) We give a brief survey of the state-of-art schemes for automatic generation of security protocol implementations based on both the computational model and the symbolic model. We find that there is no scheme for automatic generation of security protocol implementations in Java from security protocol abstract specifications proved in the computational model.

2) We introduce a model of code generation in which the security protocol implementations in source language SP [S], for example, Blanchet calculus, are translated into the security protocol implementations in a target language SP[T], for example, the Java language. It is also necessary to prove that for the adversary Adv[S] that is constructed based on any adversary Adv[T], if the security protocol implementations in source language SP [S] are secure, for any adversary Adv[T], the security protocol implementations in target language SP[T] are also secure .

3) We develop an automatic generator CV2JAVA that can transform the security protocol abstract specifications, which are the inputs of CryptoVerif to the security protocol implementations written in the JAVA language. We use CryptoVerif to prove abstract security protocol implementations in Blanchet calculus in the computational model and then use the automatic generator CV2JAVA developed by us to generate the SSHV2 security protocol implementation written in JAVA.

4) We use the automatic generator CV2JAVA and CryptoVerif to generate the SSHV2 security protocol implementation written in JAVA. First, we develop the SSHV2 security protocol implementation in Blanchet calculus, and then CryptoVerif is used to conduct the analysis of authentication of a SSHV2 security protocol implementation written in Blanchet calculus; finally, we use the automatic generator CV2JAVA developed by us to generate the SSHV2 security protocol implementation written in JAVA.

2 Related Work

In this section, we give an overview of the state of the art of mechanized generation of security protocol implementations from abstract security protocol specifications proved in two major models: the symbolic model and computational model. According to related references, there is no scheme or tool for the mechanized generation of security protocol implementation in the Java language from abstract security protocol specifications proved in the computational model.

In the symbolic model, Pironti and Sisto [16] develop an automatic framework called Spi2Java, which can translate the abstract security protocol specifications written in typed Spi calculus to the security protocol implementations based on some special libraries written in Java. In addition, if the special libraries have met certain requirements, the generated Java implementation correctly simulates the typed Spi calculus specification. Following the work of Pironti and Sisto [16], Pironti et al. [14] first improve the Spi2Java framework and then develop a semi-automatic tool to generate the security protocol implementations; Copet et al. [8] present a visual user interface approach SPI2JAVAGUI based on the automatic framework Spi2Java to generate and verify the security protocol implementations; Pironti and Sisto [15] also present an approach based on the automatic tool FS2PV to analyze the security of the data transformation function in protocol code written in a function language. Avalle et al. [2] design a framework called JavaSPI in which the Java language is not only used as a modeling language but also as the implementation language in the security protocol. In essence, it translates the abstract security protocol specifications written in Java into the inputs, which are analyzed by ProVerif, a resolution-based theorem prover for security protocols. Based on the work

of Avalle et al. [2], Copet and Sisto [9] develop an automated tool based on the JavaSPI framework to generate the Java protocol code. Avalle et al. [1] present a survey on the methods of generating the security protocol code from the view of protocol logic and note that there are two main approaches: model extraction and code generation.

Backes et al. [3] introduce a new code generator Expi2Java that can translate the abstract security protocol specifications written in an extensible variant of the Spi calculus and is analyzed by ProVerif into the security protocol implementations written in Java, which has the features of concurrency, synchronization between threads, exception handling and a type system. At the same time, they formalized the translation algorithm of Expi2Java with the Coq proof assistant and proved that if the original models are well-typed, the generated programs are also well-typed. Li et al. [11] propose a multi-objective-language-oriented automatic code generation scheme for security protocols based on XML that describes the formal model of the security protocol. Quaresma and Probst [18] present a protocol implementation generator framework based on the LySatool and a translator from the LySa calculus in the symbolic model into C or the Java language. Modesti [13] develops an AnBx compiler that accepts a special notation AnB to generate the protocol code. However, the work does not provide the proof of the soundness of the translation process.

As for the computational model, Cade and Blanchet [5, 6, 7] develop a compiler that takes an abstract specification of the protocol as the input language of the computational protocol verifier CryptoVerif and translates it into an OCaml implementation. They also prove that this compiler preserves the security properties proved by CryptoVerif, and if the abstract protocol specification is proved secure in the computational model by CryptoVerif. Li et al. [12] develop an automatic verifier SubJAVA2CV, which is able to transform security protocols written in SubJAVA to the security protocol abstract specification written in Blanchet calculus in the computational model. However, as we mentioned earlier, none of these schemes can provide mechanized generation of security protocol implementations in Java from security protocol abstract specifications proved in the computational model, which is what we aim to achieve in this study.

3 The Model of Code Generation from Abstract Security Protocol Specifications

For the code generation in Figure 1, we should prepare the security protocol implementations SP[S] written in formal languages, such as Pi calculus, Blanchet calculus, and SPI calculus; while the target language of the security protocol implementations SP[T] are programming languages, for example, Java, C #, and the C language. This means that the target languages are programming languages, and the source languages are formal languages. If the code generation method is used to generate security protocol implementations in target languages, there are two requirements that need to be satisfied:

1) The semantic of the target language simulates the semantic of the source language;

2) For the adversary Adv[S] constructed based on any adversary Adv[T], if the security protocol implementation written in source language SP[S] is secure, for any adversary Adv[S], the security protocol implementation written in target language SP[T] is also secure.

The first requirement shows that from the view of the behaviors, the relationship between the security protocol implementations written in target language SP[T] and the security protocol implementations written in source language SP[S] is simulation or observational equivalence.

The second requirement shows how to prove the security properties of the security protocol implementations written in target language SP[T]. For any adversaries in the context, we want to prove the cryptographic security properties of the security protocol implementations written in target language SP[T]. Thus, according to any adversary Adv[T] in the security protocol implementations written in target language SP[T], we construct an adversary Adv[S] in the security protocol implementations written in source language SP[S] and then prove that for the adversary Adv[S]. If the security protocol implementations written in source language SP[S] are secure, for any adversary Adv[T], the security protocol implementations written in target language SP[T] are secure.

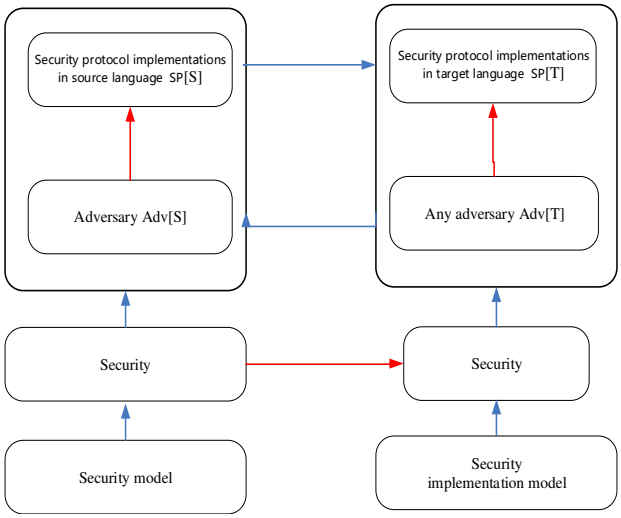

Figure 1: Model of code generation

4 Blanchet Calculus

Blanchet calculus [4] is based on probabilistic polynomial calculus and has been widely used to prove the security properties of security protocols. In Blanchet calculus, messages are modelled as bitstrings and cryptographic primitives are functions that operate on bitstrings. Blanchet calculus has two versions: channels front-end and oracles front-end in order to be as the input of CryptoVerif. Here we introduce the channels front-end version. Blanchet calculus is mainly made up of terms, conditions and processes. The adversary is modelled by an evaluation context Blanchet calculus mainly consists of terms and processes. Suppose there are m terms indexed as $1, \cdots, i, \cdots, m$; then, variable access $x[M_1, \cdots, M_i, \cdots, M_m]$ or function application $f(M_1, \cdots, M_i, \cdots, M_m)$ represent computations on bitstrings represented by these m terms. The variable access $x[M_1, \cdots, M_i, \cdots, M_m]$ outputs the content of the cell of indices $[M_1, \cdots, M_i, \cdots, M_m]$ of the m-dimensional array variable x. The function application $f(M_1, \cdots, M_i, \cdots, M_m)$ outputs the result of applying function f to $[M_1, \cdots, M_i, \cdots, M_m]$.

The processes in Blanchet calculus include the input process and output process. Input process 0 does nothing; $Q|Q'$ is the parallel composition of Q and Q'; $!^i$ represents n copies of Q in parallel; newChannel $c; Q$ produces a new private channel n and performs Q. Output process new $x[i_1, \cdots, i_m] : T; P$ chooses i_1, a new random number, uniformly, stores it in $x[i_1, \cdots, i_m]$, and then executes P. Random numbers are chosen by new $x[i_1, \cdots, i_m] : T$. Output process let new $x[i_1, \cdots, i_m] : T = M$ in P stores the bitstring value of M in $x[i_1, \cdots, i_m]$ and executes P.

Find process shows that it tries to find a branch J in $[1, m]$ such that there are values of $u_{j_1}, \cdots, u_{j_{m_j}}$ for which $M_{j_1}, \cdots, M_{j_{l_j}}$ are defined and M_j is true. In the case of success, it executes P. When event $e(M_1, \cdots, M_m)$ has been executed, the formula event $e(M_1, \cdots, M_m)$ is true.

5 Code Generation from Security Protocol Implementations Written in Blanchet Calculus

We know that the security protocol implementations written in Blanchet calculus are mainly composed of processes, and processes mainly consist of top processes and communicating party processes. The top process is the main process. The parties in security protocols are modelled as communicating party processes. Owning to the powerful ability of Blanchet calculus, the communicating parties in the security protocols are multiple parties. Here we classified it into two kinds of communicating parties: the sender processes and the receiver processes in the template for security protocol implementations written in Blanchet calculus as shown in Figure 2. In our model there can exist multiple sender parties and receiver parties. When we model the security protocol with Blanchet calculus, the problem needs to be addressed with different role names.

Generally the template file used to analyze the security protocols through Blanchet calculus is made up of the following sections: Type declaration, function declaration, cryptographic primitive declaration, channel declaration, security property, sender process, receiver process and top process. The type declaration section is composed of the related type in formalizing the security protocols, in which some types are owned by Blanchet calculus and other types are defined according to the requirement in formalizing the security protocols. The cryptographic primitives, for example, indistinguishability under adaptive chosen ciphertext attack public key enc, which are used in analyzing the security protocols, is declared in the Cryptographic primitive section. Channels are used to model the communication channel between the sender and receiver processes that are declared in the Channel declaration section. In the template, we assume that there are two roles in the security protocols: One is the sender, and the other is the receiver. Hence, the sender is modelled as the sender process in the Sender process section, and the receiver is modelled as the receiver process in the Receiver process section. The top process is composed of the sender process and the receiver process.

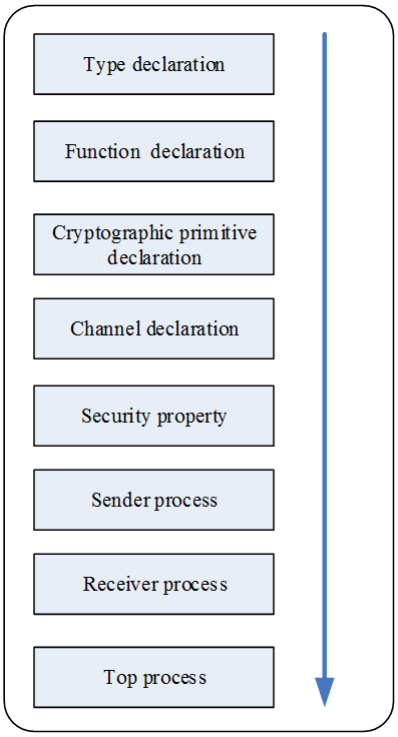

Figure 2: Template for security protocol implementations written in Blanchet calculus

Hence, in the translation of the code generator, the top process is mapped into the main program in security protocol implementations written in JAVA, which is

a class. Regarding the type in Blanchet calculus, we directly transform it into the corresponding type in JAVA because in Blanchet calculus, we can define the required types for the types in JAVA showed in Table 1. The function in Blanchet calculus is translated into the function in JAVA. The cryptographic primitive in Blanchet calculus is transformed into the security package in JAVA. The communicating channel processes, which include input channels and output channels, are mapped into the classes, which are implemented based on the socket and are the parties in security protocol implementations written in JAVA. These classes are responsible for sending and receiving messages between the communicating parties. The sender process and the receiver process are mapped into the sender class and receiver class, respectively, in Figure 2. Figure 3 presents the generators from Blanchet calculus to JAVA in detail.

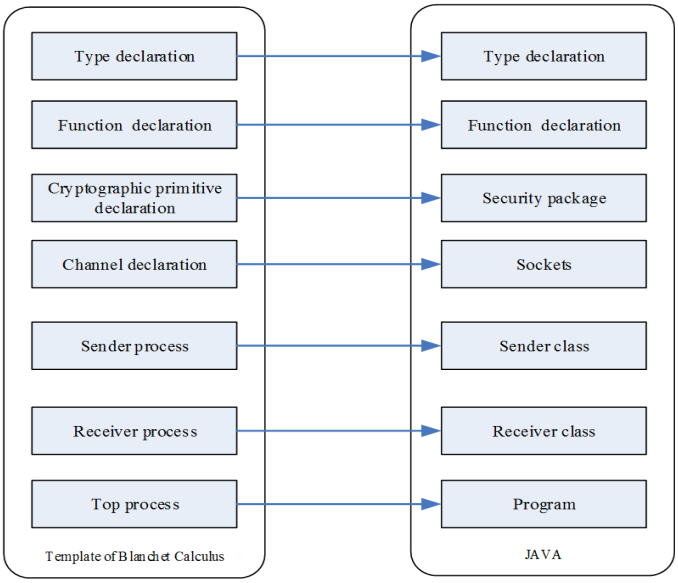

Figure 3: The model of code generation from Blanchet calculus

$$[\![x[M_1,\cdots,M_m]]\!] \triangleright \boxed{\begin{array}{l}\text{int}[]\ \text{x=new int}[\text{m}];\\ \text{x}[\text{M}_1];\ \text{x}[\text{M}_2];\ \text{x}[\text{M}_3];\ldots;\ \text{x}[\text{M}_m];\end{array}}$$

Figure 4: GeneratorVariableAccess(x[:T])

Generation function GeneratorVariableAccess(x[:T]) in Figure 4 maps variable access x[:T] in Blanchet calculus into variable (int[] x= new int[m]; x[M_1]; x[M_2]; x[M_3];... x[M_m]) in JAVA.

Table 1: The transforming from some types in Blanchet calculus to some types in JAVA

Types in Blanchet calculus	Types in JAVA
keyseed	SecureRandom
key	Key
cleartext	String
ciphertext	String
seed	SecureRandom
mkeyseed	SecureRandom
mkey	key
macinput	String
macres	String
pkey	PublicKey
skey	PrivateKey
signinput	String
signature	byte[]
hashinput	String
hashoutput	String
input	int
output	int

$$[\![f[M_1,\cdots,M_m]]\!] \triangleright \boxed{\text{public Type FunName(Type1 M}_1\text{);}}$$

Figure 5: GeneratorFunctionApplication(f)

Generation function GeneratorFunctionApplication(f) in Figure 5 transforms function f in Blanchet calculus into function (public Type FunName(Type1 M_1);) defined by us in JAVA.

$$[\![in\ (c[M_1,\cdots,M_l],(x_1[\tilde{i}]:T_1,\cdots,x_k[\tilde{i}]:T_k));P]\!]$$

```
ServerSocket c;
c = new ServerSocket(8080);
Socket c1=c.accept();
InputStreamReader in=new InputStreamReader(c1.getInputStream());
▷ BufferedReader br=new BufferedReader(in);
pw=new PrintWriter(c1.getOutputStream(),true);
while(true) {
receive_message =br.readLine();
}
```

Figure 6: GeneratorInput in

Regarding the generation of input process in Blanchet calculus, we mainly use the network statements, for example, socket statement, in JAVA to implement it. A special socket is defined to implement the input process. The property of the input process socket of the sender is the address and port number of the sender. The property of the input process socket of the receiver is the address and port number of the receiver. Hence, input process

in() in Blanchet calculus can be translated into JAVA as shown in Figure 6.

$[\![out\ (c[M_1,\cdots M_l],\langle N_1,\cdots,N_k\rangle);Q]\!]$

```
ServerSocket c;
c = new ServerSocket(8080);
Socket c1=c.accept();
▷ InputStreamReader out =new InputStreamReader(c1.getInputStream());
BufferedReader br=new BufferedReader(out);
pw=new PrintWriter(c1.getOutputStream(),true);
pw.println(send_message);
```

Figure 7: GeneratorOutput out

Regarding the generation of output process out in Blanchet calculus, we also use the network statements in JAVA to implement it. A special socket is defined to implement the output process. The property of the output process socket is the address and port number of the receiver. The property of the output process socket of the receiver is the address and port number of the sender. Hence, output process out in Blanchet calculus can be translated into JAVA showed in Figure 7.

$[\![new\ x[i_1,\cdots,i_m]:T;P]\!] \triangleright$

```
T x_i = new T( );
P.main();
```

Figure 8: Generator new x

Regarding the process new x in Blanchet calculus, we mainly use the class new in JAVA to implement it. Hence, the new process in Blanchet calculus is transformed into JAVA code in Figure 8.

$[\![let\ x[i_1,\cdots,i_m]:T=M\ in\ P]\!] \triangleright$

```
for(int i=0;i<m;i++)
{
x[i]=M_i;
}
```

Figure 9: GeneratorAssignment Let

The assignment process let x: T=M in P is transformed into the statementsfor (int i=0; i¡i++)x[i]=$M_i$in JAVA in Figure 9.

$[\![if\ defined(M_1,\cdots M_l)\wedge M\ then\ P\ else\ P']\!] \triangleright$

```
if(l<=M.length)?
    if(arryContains(M,m))
                P.main();
Else? P?main();
```

```
public static boolean arryContains
(String[] stringArray, String source)
{
 List<String>tempList= Arrays.asList(stringArray);?
 ?if(tempList.contains(source))){?return true;} else
 { return false;?}
}
```

```
public static <T> boolean contains
( final T[] array, final T v )
{
  for ( final T e:array )
    if ( e==v||v!= null&&v.equals( e ) )
       return true;
  return false;
}
```

Figure 10: Generator additional if then

In the transformation of the conditional process if defined $(M_1,\cdots,M_1)M$ then P else P', there are two methods that can be used to implement the key part defined $(M_1,\cdots,M_1)M$. One method is that the array is first transformed into the list; then, we can use the method contained in the list to implement it. Hence, the arryContains is implemented in JAVA as:

```
public static boolean arryContains
        (String[] stringArray,String source)
{List¡String¿tempList=Array.asList(stringArray);
if (tempList.Contains(source)){return true;}
else {return false;}}
```

In the other method, the traversing array is used to implement it. So, the key part defined $(MM_1,\cdots,M_1)M$ is implemented in JAVA as in Figure 10:

```
public static boolean
        arryContains(final T[]array,fianl T v)
(final T e:array)if (e==v||v!=null&&v.equals(e))
return true;return false).
```

6 Automatic Generator CV2JAVA from Blanchet Calculus to JAVA

According to the generation functions introduced in the previous section, we develop an automatic generator CV2JAVA that accepts the security protocol model expressed by Blanchet calculus as an input and produces security protocol implementations in JAVA as an output.

Figure 11 presents the application of automatic generator CV2JAVA. First, we model the security protocols with Blanchet calculus, and then, the tool CryptoVerif is used to implement the analysis of security, and finally, the automatic generator CV2JAVA is used to generate the security protocol implementations written in JAVA.

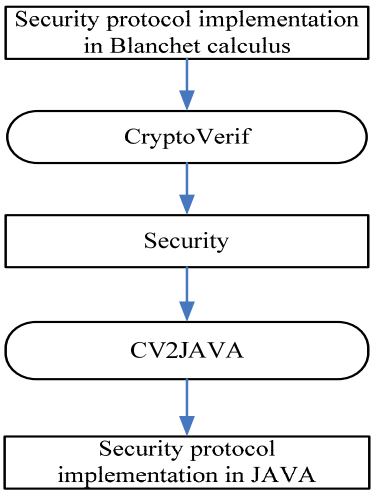

Figure 11: Application of automatic generator CV2JAVA

Figure 12 shows the development of automatic generator CV2JAVA. First, according to the informal specification of the security protocol, we produce the security protocol implementations written in Blanchet calculus, and then, based on the syntax of Blanchet calculus, the lexical analyzer developed by us is used to analyze and verify the correctness of security protocol implementations written in Blanchet calculus. If verification succeeds, lexical elements, for example tokens, are produced. Next, the parser developed by us is used to process tokens and produces an abstract syntax tree to represent the structure of security protocol implementation written in Blanchet calculus. The goal is to generate secure security protocol implementations written in JAVA, so the structures and elements that are not related to the programming implementation of the security protocol, for example, irrelevant events, are deleted, and the simplified abstract syntax tree is produced. After this step, the translator mapping the simplified abstract syntax tree in Blanchet calculus into the abstract syntax tree in JAVA is derived.

Next, we develop the code generator to produce the security protocol implementations written in JAVA. In the next section, we mainly concentrate on the development of automatic generator CV2JAVA based on JavaCC. We use JavaCC to develop the lexical analyzer and parser. First, we need to construct the .jj file and then use JavaCC to implement the lexical analyzer and parser for Blanchet calculus. The .jj file is mainly composed of options, function declarations, specification for lexical analysis and BNF notations for Blanchet calculus.

6.1 Simplifier

Owing to the specialty of Abstract Syntax Tree of security protocol implementation written in Blanchet calculus, it is not easy to directly transform it into Abstract Syntax Tree of security protocol implementation written in JAVA. At the same time, it may produce some mistakes in the transformations.

To reduce Abstract Syntax Tree of security protocol implementation written in Blanchet calculus, we need to program a simplifier to perform the function. Simplifier accepts the Abstract Syntax Tree as inputs and produces the simplified Abstract Syntax Tree. Here, we implement the simplifier by using visitor pattern.

Visitor pattern is one type of design pattern. Visitor pattern defines an operation on the elements of an object structure. Without modifying the structure of the type, visitor pattern can access the different objects of a type and make different operations on them. In most cases, we do not add a new node in the Abstract Syntax Tree after it has been generated. Hence, we find that it is a good method to develop the simplifier visitor to address the Abstract Syntax Tree. At the same time, different operations can be made on the different nodes. Hence, it is proper to use visitor patterns to address this situation.

Simplifier visitor in Figure 13 visits, adds and deletes the nodes according to the Abstract Syntax Tree for JAVA. There are two different methods for processing the keywords in simplified visitors. One is the JAVA untransforming method. The other is the JAVA transforming method. The JAVA untransforming method visits the Abstract Syntax Tree for Blanchet calculus and obtains syntax keywords and then compares it to the syntax keywords in JAVA. If they are the same, the syntax keywords in Blanchet calculus are not modified. At the same time, it abstracts the main keyword nodes and variable nodes in Blanchet calculus to implement the simplification of the Abstract Syntax Tree. The JAVA transforming method visits the Abstract Syntax Tree for Blanchet calculus and obtains syntax keywords and then compares them to the syntax keywords in JAVA. If they are not same, the syntax keywords in Blanchet calculus are modified. At the same time, it abstracts the main keyword nodes and variable nodes in Blanchet calculus to implement the simplification of the Abstract Syntax Tree for the main key variable nodes.

JJTree has additional good support for the visitor design pattern. If we set the VISITOR option to true, a jjtAccept() method and childrenaccept() method are inserted into all of the node classes it generates by JJTree. Apart from that, a visitor interface Visitor.java is also produced that can be implemented and passed to the nodes to accept. Our simplifier is an instance of a visitor interface Visitor.java. JJTree generates SimpleNodes, and they are processed differently.

Here is the process of the statement new a:T;P in Blanchet calculus. It generates a new variable for which the type is T and then performs process P. There is also the key word new. Hence, we construct the simplifier visitor according to the new statement syntax in JAVA.

The syntax of the statement new a:T;P in Blanchet calculus is presented in Figure 14. The syntax of statement new in JAVA is shown in Figure 15.

According to the production of the new statement, the syntax tree of statement new a:T;P in Blanchet calculus

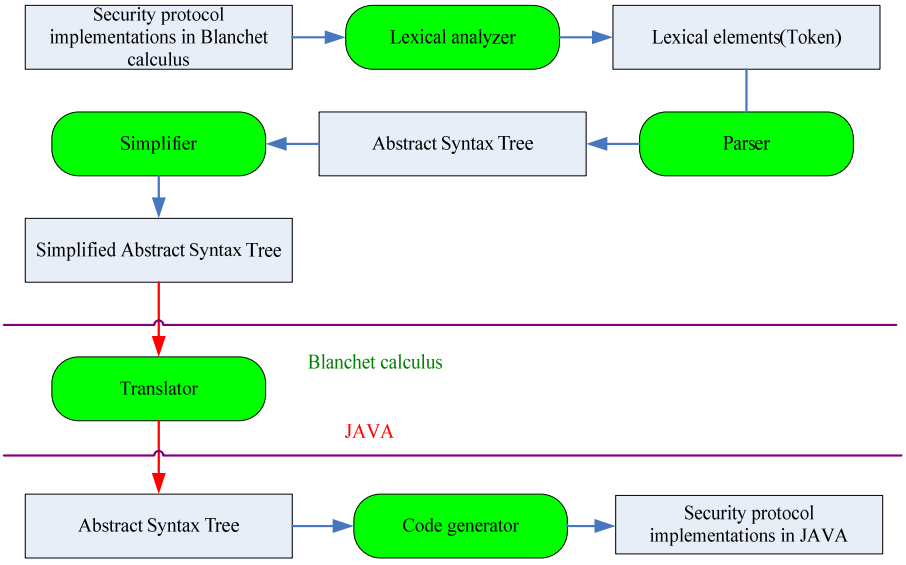

Figure 12: Development of automatic generator CV2JAVA

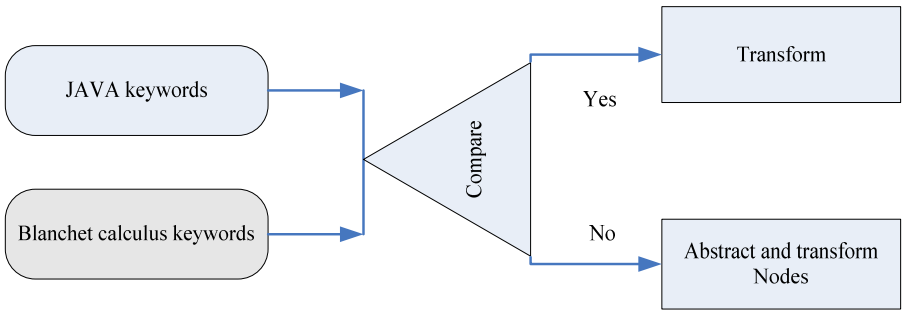

Figure 13: The idea of the simplifier visitor

```
void newterm():
{ } {
  new   vartype() scolon() term()
}
```

Figure 14: The syntax of statement new a: T; P in Blanchet calculus

is expressed in Figure 16. The key node is newex node, and the key variables are constant variable identex and type variable indentex.

Simplifier visitor travels the Syntax Tree of new statement in Blanchet calculus and abstracts the key nodes and adjusts the sequence of the key nodes of Syntax Tree for Blanchet calculus. Finally, it adds the correspondent Java nodes. The method is the JAVA untransforming method. The key word is new, and the node term() should be deleted by the case statement in Figure 17. The method used to delete the nodes is implemented in the method for TreeVisitor and can be used to travel the nodes of the syntax tree.

```
Type a = new Type();
```

Figure 15: The syntax of statement new in JAVA

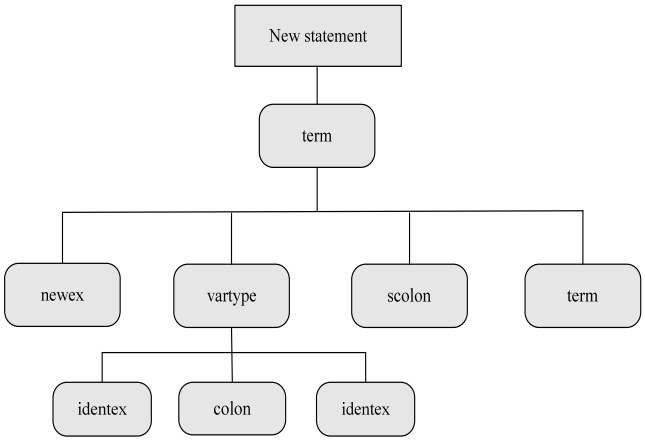

Figure 16: Syntax tree of new statement in Blanchet calculus

Based on the new statement syntax tree, TreeVisitor travels the nodes following the sequence. First, the jjtAccept(Visitor, Object) method of the new statement node is invoked by the main method in parser. Parameter Visitor is the object addvisitor. Then, the visit method of object addvisitor is called, and the new statement node is sent to visit method. The post-order traversal is applied. So, with parameter addvisitor, the visit method is called childrenAccept(Visitor, Object) in new statement node to travel to subnodes. The jjtAccept(Visitor, Object) method in subnodes to perform the traversal of the subtree is invoked by the childrenAccep method. Then, it invokes the addchildren() method to add new nodes. In addition, the children fields in the new node are revalued to avoid the logical error. At the same time, the fourth node is deleted by invoking the method visitor of the object deletevisitor.

```
case EG2TreeConstants.JJTNEWEX:
  node.childrenAccept(this, data);
  deletechildren(node, 4);
  break;
case EG2TreeConstants.JJTNEWEX:
  node.childrenAccept(this, data);
  ((StringBuffer) data).append(node.jjtGetValue().toString());
  break;
```

Figure 17: Case statement

6.2 Translator

Translator is also a visitor that takes the simplified Abstract Syntax Tree for Blanchet calculus as an input and outputs the Abstract Syntax Tree for JAVA. This means that translator is a mapping function from language elements in Blanchet calculus to language elements in JAVA based on the definitions. In the next section, we show how to use the visitor to perform the mapping function GeneratorRandomNumber x, which implements mapping from a new statement in Blanchet to a new statement in JAVA.

We have used the simplifier to delete the node term(). To implement the translation from the new statement in Blanchet calculus and form the new statement in JAVA, we also need to adjust the position according to the syntax tree of the new statement in JAVA. The content node, identex node, colon node, sclon node and newex node are not changed. The identex node needs to be changed through the method exchangenode(node,1). According to the syntax nodes in JAVA, the method nodeExchange in translator can be used to exchange the nodes if the jjtNodeName of the node is the same as the predefined JJTLOCALVARIABLEDECLARATION. Finally, we generate the model for the Abstract Syntax Tree in Blanchet calculus shown in Figure 18.

When translator has visited all the nodes in the Abstract Syntax Tree for Blanchet calculus, the corresponding Abstract Syntax Tree for Blanchet calculus will be generated according to the model for the Abstract Syntax Tree in Blanchet calculus in Figure 18.

Figure 19 shows the Abstract Syntax Tree for the new statement in JAVA.

6.3 Code Generator

If we have derived the Abstract Syntax Tree of Blanchet calculus, the next task is to implement the code generator to generate code of Blanchet calculus, which is the security protocol implementation written in Blanchet calculus. Although the Abstract Syntax Tree mainly consists of the type name, variable name, method name and parameter list, we are not able to directly use the leaf node to generate security protocol implementation in JAVA because

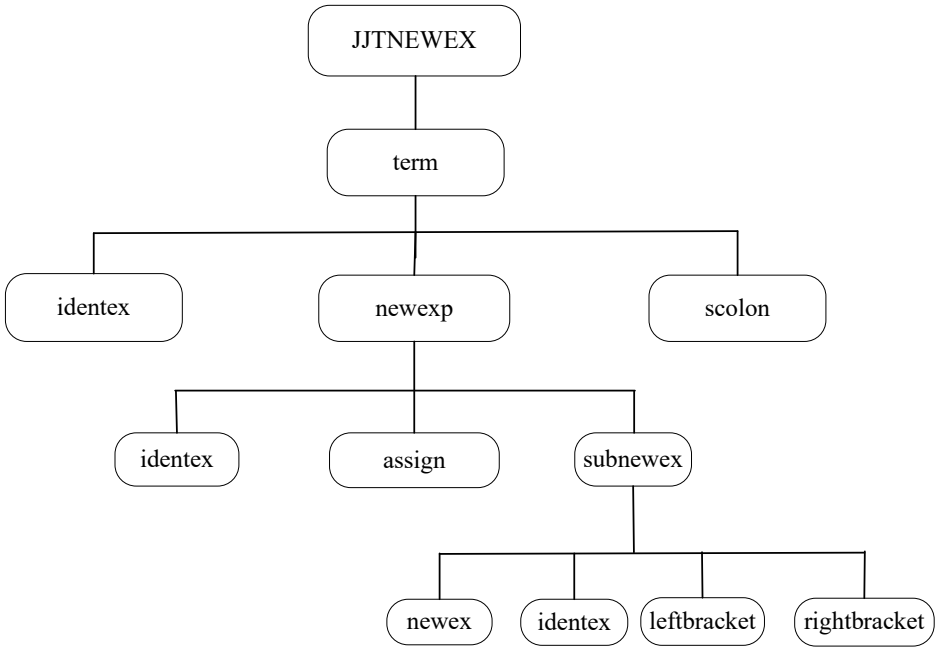

Figure 18: The model for the abstract syntax tree of code exchange method in Blanchet calculus

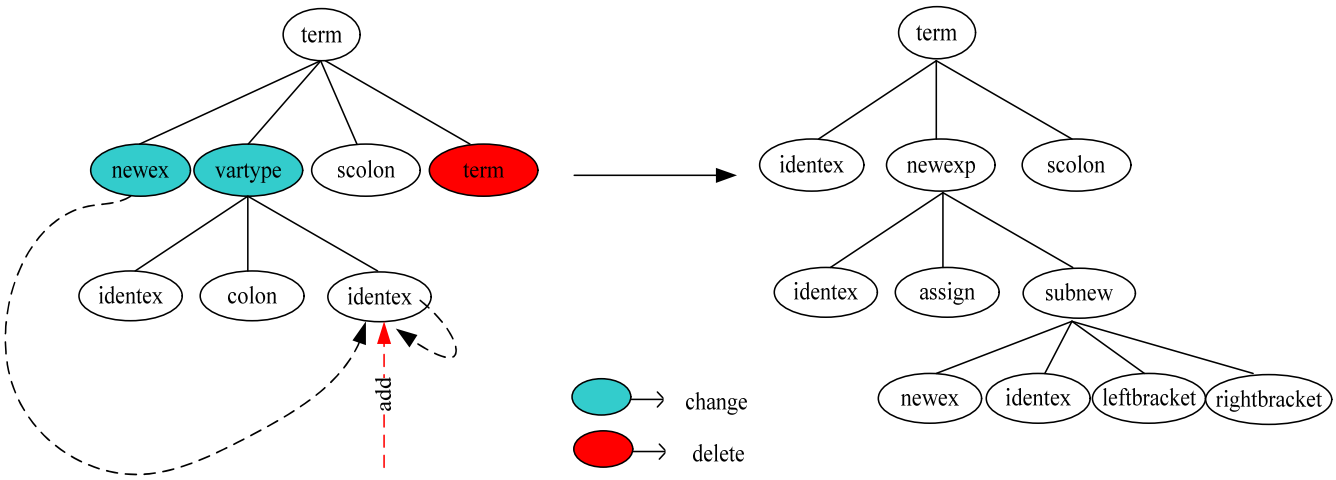

Figure 19: Abstract syntax tree of new statement

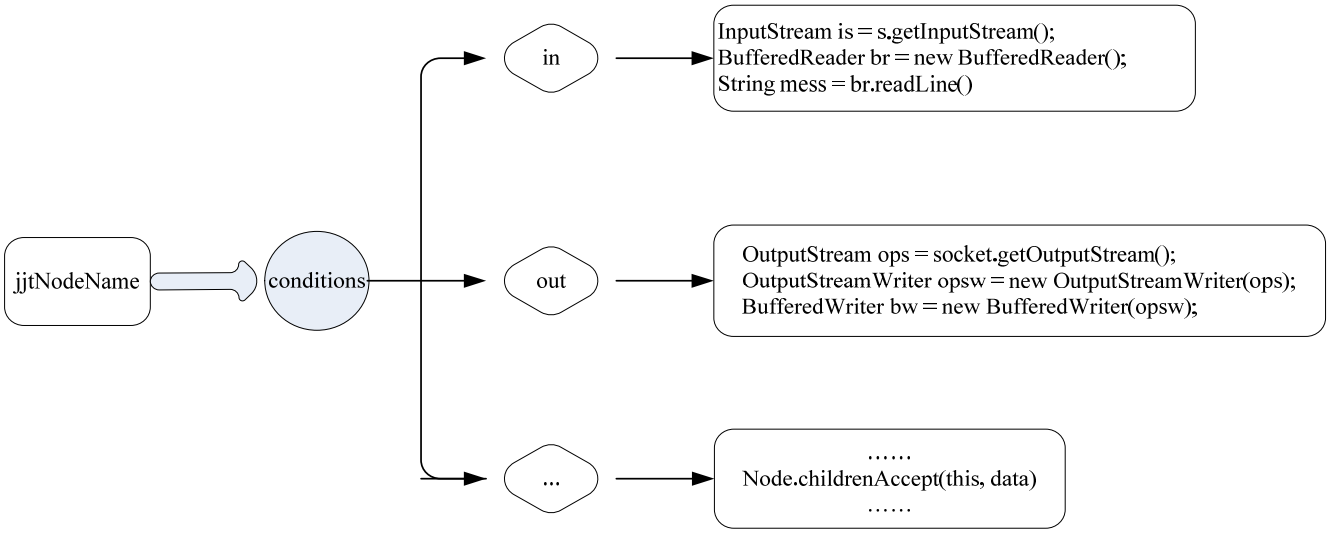

Figure 20: The structure of the code generator visitor

the channel declaration, security definition and declaration have not been transformed. Hence, we design a visitor code generator for channel declaration, security definition and declaration. When the code generator traverses the syntax tree and adds input channels, out channels, operators and security definitions and declarations are in the proper position.

When the code generator traverses the syntax tree, it first adds the declaration methods and security properties verification and then translates it into JAVA code. For example, to verify the security of sessionkey, the statement in Blanchet calculus is query secret sessionkey; therefore, the corresponding method verifSecret(sessionkey) in JAVA is added to verify the sessionkey.

Figure 20 shows the structure of the visitor code generator. The visitor method first accesses the jjtNodeName of nodes and then creates different operations on different nodes. The switch statement is used to match the value of the jjtNodeName, and the task of processing the node is performed in the case statement. It mainly adds some keywords, operators and special characters and so on into the object StringBuffer.

Security protocol implementations in JAVA is stored in the object StringBuffer after the code generator visitor travels through all of the Abstract Syntax Tree. Finally, we write the contents in the object StringBuffer into a file, and thus, the security protocol implementation in JAVA is completed.

6.4 Templator

Templator is used to add the security object expressed by events in Blanchet calculus into security protocol implementation in CryptoVerif. Thus, we can use CryptoVerif to verify the security properties of security protocol implementation in Blanchet calculus.

7 Case: SSHV2 Security Protocol Implementation in JAVA

In this section, we use the automatic generator CV2JAVA and CryptoVerif to analyze the authentication of the Secure Shell Version 2 (SSHV2) security protocol and generate its implementation written in JAVA. We first develop the SSHV2 security protocol implementation written in Blanchet calculus. Then, we use the automatic verifier CryptoVerif to analyze the authentication of the SSHV2 security protocol. Finally, we use automatic generator CV2JAVA to generate SSHV2 security protocol implementation written in JAVA.

7.1 Review of the SSHV2 Security Protocol

The SSHV2 security protocol was issued in 2006 by IETF and is designed to implement remote secure login and other secure network services over an insecure network. The SSHV2 protocol is made up of three major sub-protocols: The Transport Layer Protocol implements server authentication, confidentiality, and integrity with perfect forward secrecy. The User Authentication Protocol authenticates the client identity to the server. The Connection Protocol provides the encrypted tunnel in several logical channels. The SSHV2 security protocol uses the Digital Signature Algorithm (DSA) to replace the RSA algorithm to implement the secret key exchange in the Transport Layer Protocol and use the Hash Message Authentication Code (HMAC) to guarantee the integrity of the message.

To implement the security of data over a public network, the message exchange is made up of the following four phases: the protocol version negotiation, encryption algorithms and key negotiation, key exchange and authen-

tication. In the protocol version negotiation phase, the client and server chose the same version to communicate.

In the encryption algorithms and key exchange phases in the Transport Layer Protocol, parameter negotiation is allowed to minimize the number of round trips. The key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated. Hence, there are nine messages to be exchanged between the client and the server: Protocol Version Exchange between the client and the server SSH-protoversion-softwareversion, the client and server Algorithm Negotiation SSH_MSG_KEXINIT, key exchange request SSH_MSG_KEXDH_GEX_REQUEST, key exchange parameters response SSH_MSG_KEXDH_GEX_GROUP, key exchange parameter initialization SSH_MSG_KEXDH_GEX_INIT, key exchange response SSH_MSG_KEXDH_GEX_REPLY and the new key message SSH_MSG_NEWKEYS.

In the authentication phase, the server implements the authentication of the client. The server initiates the authentication by telling the client which authentication methods can be chosen to continue the exchange. The client can choose the proper methods listed by the server in any order. This gives the server complete control over the authentication process. There are three methods that can be used by the client: public key, password, and host-based client authentication methods. Here, we chose the public key authentication method.

In the public key authentication method, the client sends the Authentication Requests message to the server: (1) SSH_MSG_USERAUTH_REQUEST, which is made up of username, servicename, methodname and method_specific fields. The value of methodname is "public key" based on the authentication method we chose. The method_specific field consists of signature_algorithm, public key, signature fields. The signature field is made up of session ID,SSH_MSG_USERAUTH_REQUEST, user name, service, "public key", TRUE, algorithm name, public key fields.

When the server accepts authentication, it sends the message (2) SSH_MSG_USERAUTH_SUCCESS to the client; otherwise, it sends the messages SSH_MSG_USERAUTH_FAILURE SSH, which means that the authentication request has failed.

7.2 SSHV2 Security Protocol Implementation Written in Blanchet Calculus

The formal model implemented in Blanchet calculus includes initialization process, the client process and server process.

The initialization process generates the public key pkeyrsa and private key skeyrsa for the server, the signature public key signpkey, the signature private key signskey for the server, the public key pkey_c, private key skey_c for the client and the signature public key psignkey_c, the signature private key ssignkey_c for the client. Then, the public keys pkeyrsa, skeyrsa, signpkey, pkey_c, keyhash are published through the channel c. After that, the client process ClientProcess and the server process ServerProcess are launched.

The client process produces the version information clientversion and sends it through the channel c1. Then, it receives the version information messagetwo_s from the channel c4 and sets the version equal to messagetwo_s. The client process also generates the client algorithm parameters message_algoclient and sends it through the channel c5. Next, it receives the server algorithm parameters messagetwo_s from the channel c8 The client process accepts messagetwo_s. The client process generates the key exchange parameters minc, dp, maxc using the statements new minc: Z; new dp: vlue; new maxc: Z; it then sends them to the server process through channel c9. The random number r_c produced using the statement new r_c: Z accepts as the input of the function exp(); then, we can get the dh_e, which is sent to the server process through the channel c13. When the server receives the message dh_e, it will send a message dh_f_s as the response. Thus, through the channel c16, the client process gets the message dh_f_s generated by the server process and r_c to compute the client share key clientshareK using the function Gtokey(). The client process verifies the digital message dh_signature_s received from the server process using the function check(htomes(clienthash), psignkey_c, dh_signature_s) with the signature key psignkey_c.

If the verification is successful, the client authenticates the server. Next, the client process generates the new session key sessionkey using the function hashtokey(), and then, the new session key sessionkey is encrypted with the public keypkeyrsa of the server, and the ciphertext messenc is produced and is sent to the server process through the channel c17. The four parameters of the function hashtokey() are client share key clientshareK, the hash of client message clienthash, client message clientmes and session id sessionid. The session key sessionkey is the hash value of the four parameters: client shares key clientshareK, the hash of client message clienthash, client message clientmes and session id seesionid. Next, the client generates the authentication request message SSH_MSG_USERAUTH_REQUEST, which includes username, servicename, methodname, method_specific. Then, it uses the client signature key ssignkey_c to generate the digital signature request_signature of the authentication request message SSH_MSG_USERAUTH_REQUEST and send it to the server process through channel c19.

The server process receives the version information messageversion_c through channel c6. Then, it generates the server version information serverversion and sends it through channel c3. The server process also receives the client algorithm parameters message_algoc through the channel c2. Then, it generates the server algorithm parameters message_algoserver and sends it through the channel c7. The server receives the message message_DHrequ through the channel c10. The message message_DHrequ is made up of the key parameters dhmic,

dhdp and dhmaxc, which are accepted as the inputs of function exp() to generate the message message_DHgroup and send it to the client process.

The server receives the message message_c and computes the shared key k through k=exp(message_e,r_s). Next, it calculates the hash value hashserver by the hash function hash(keyhash,messagedhhash). Then, the signature signtext is generated by fhashtosign(hashserver) and the client uses the signature private key ssingkey_c to produce the digital signature dh_signature. The server process generates the authentication message dhgexreply through the function concatreply, which accepts dh_f, pkeyrsa, session_id_s,dh_signature as the inputs. Finally, the server process sends the authentication message dhgexreply to the client through the channel c15.

The server process receives the authentication request message messageauth_c from the client process through the channel c20. Then, it constructs the message messageauth_c through the function concatsshuser(). Next, the server process uses the function check to verify the authentication request message with the public key cpkey_c. If the verification is successful, the server process sends the message SSH_MSG_USERAUTH_SUCESS through the channel c21.

7.3 Authentication of SSHV2 Security Protocol Implementation in Blanchet Calculus

In this section, we give a brief overview of the mechanized prover CryptoVerif, which is used to automatically analyze authentication of SSHV2 security protocol implementation in Blanchet calculus.

Here, we use non-injective correspondences in Figure 21 to model the authentication from server to client and from client to server. Event client(x, y, kx, px, gx, gy, krsa)==> server(x, y, kx, px, gx, gy, krsa) is used to authenticate the client by server. Event server A(ya)==>client A(ya) is used to authenticate the server by client. These events and non-injective correspondences had been added into the model implemented in Blanchet calculus by hand.

The analysis was performed by CryptoVerif and succeeded. The result is shown in Figure 22, and the SSHV2 security protocol is proved to guarantee authentication. "All queries proved" in Figure 22 shows that event client(x, y, kx, px, gx, gy, krsa)==> server(x, y, kx, px, gx, gy, krsa) is true, which shows that the server authenticates the client. Event server A(ya)==> client A(ya) is true, which shows that the client authenticates the server; query secret sessionkey is proved, which shows that the session key is secure.

```
event client(version,version,key,value,G,G,rsapkey).
event server(version,version,key,value,G,G,rsapkey).
event clientA(signature).
event serverA(signature).

query x:version,y:version,px:value,gx:G,gy:G,kx:key,krsa:rsapkey;
event client(x,y,kx,px,gx,gy,krsa)==>server(x,y,kx,px,gx,gy,krsa).
query ya:signature;
event serverA(ya)==>clientA(ya).
query secret sessionkey.
```

Figure 21: Events

7.4 SSHV2 Security Protocol Implementation Written in JAVA

According to the SSHV2 protocol, a protocol implementation written in Blanchet Calculus, we use the automatic verifier CV2JAVA to generate the SSHV2 protocol implementation in JAVA, as shown in Figure 23. But in our current version, there are some limitations on it. we also need to implement its method in the class. Then, we run the client and server code, which is shown in Figure 24. In addition, it also shows that the server authenticates the client.

8 Conclusions

In the last twenty years, many security protocols have been introduced and deployed in all kinds of information systems. Hence, the verification of the security properties of these protocols has received plenty great deal of attention. Now, there is a popular issue: analysis of the security protocol implementations, which is introduced from the security field. In this study, we first present the model of code generation from abstract security protocol specifications. Then, we develop an automatic generator CV2JAVA, which is able to translate security protocol abstract specifications written in Blanchet calculus in the computational model into security protocol implementations written in JAVA. Moreover, we also use the automatic generator CV2JAVA and CryptoVerif to generate the SSHV2 security protocol implementation written in JAVA from the SSHV2 security protocol implementation written in Blanchet calculus proved in the computational model.

One of our main work is to develop a generator form security protocols implementations written in Blanchet calculus to JAVA code and is not to develop a complex complier. Hence the compile time error and compiler optimization etc. have not been addressed in current work. Compared to the works of Cade and Blanchet [12,13], there are six big difference. Firstly, the target languages

Figure 22: The result

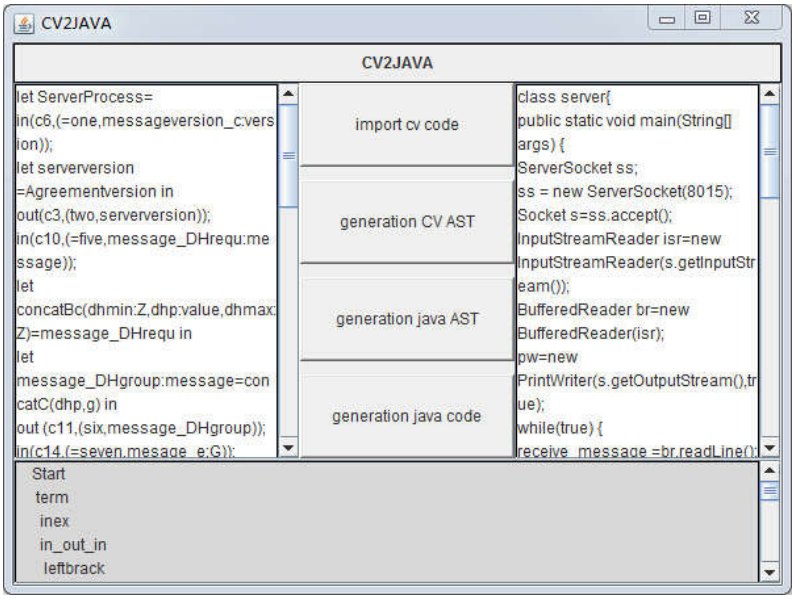

Figure 23: Generating SSHV2 protocol implementation written in JAVA

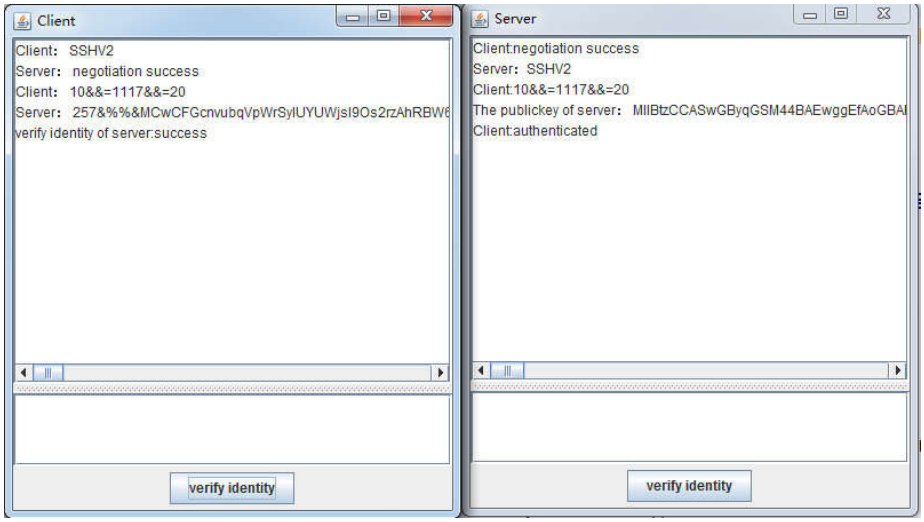

Figure 24: The result of executing the client and server code

are different. Firstly, the target languages are different. The target language in Cade and Blanchet [12,13] is the Ocaml language which is a function language. By contrast, the target language in our work is Java language which is an imperative language. Secondly, the formal languages are different. In Cade and Blanchet [12,13], the modified Blanchet calculus is used, in which some statements are added and some other statements are removed in order to deal with transforming from it to Ocaml language. By contrast, we use the original Blanchet calculus to deal with transforming from it to JAVA language. Thirdly, the Blanchet calculus has two versions: channels front-end and oracles front-end as the input language of CryptoVerif. Cade and Blanchet [12] use the oracles front-end to model the SSH protocol. However, we use the channels front-end to model the SSH protocol. Hence the formal code are very different. Fourthly, the technologies based upon during the development of the translator may be different because of the different target languages. They use the Ocaml language to develop the complier. But we chose the JAVACC as the base in order to process the transformation from Blanchet calculus to JAVA language.. Fifthly, The SSH protocol modelled in our work is nice distinction. For example, the cryptographic primitives. Finally, the proof of the correctness of translator are different. Owning to the differences of the formal languages and the target languages, although the frameworks are similar, the technologies used are different. We will prove it from the view of operational semantics. Moreover, the operational semantics are different due to the different target.

In our current version of CV2JAVA, there are some limitations on it. At a time the only one process in Blanchet calculus can be translated into classes in JAVA code. At the same time we also need to implement its method in the class. In future work, we will improve the generator CA2JAVA and use the formal method to prove the correctness of translation from Blanchet calculus to JAVA and enhance the ability of the tool.

Acknowledgments

This work was supported by natural science foundation of Hubei Province under the Grants No. 2014CFB249 and by the foundation of China Scholarship Council.Bo Meng and Chin-Tser Huang contributed equally to this work.The authors gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] 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.

[2] M. Avalle, A. Pironti, R. Sisto, and D. Pozza, "The Java SPI framework for security protocol implementation," in *the Sixth International Conference on Availability, Reliability and Security*, pp. 746–751, Vienna,Austria, Aug. 2011.

[3] M. Backes, A. Busenius1, and C. Hritcu, "On the development and formalization of an extensible code generator for real life security protocols," in *Proceeding of The 4th International Conference on NASA Formal Methods*, pp. 371–387, Norfolk, USA, Apr. 2012.

[4] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[5] D. Cade and B. Blanchet, "From computationally-proved protocol specifications to implementations," in *Proceeding of The Seventh International Conference on Availability, Reliability and Security*, pp. 204–208, Prague, Czech, Aug. 2012.

[6] D. Cade and B. Blanchet, "From computationally-proved protocol specifications to implementations and application to ssh," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 4, no. 1, pp. 4–31, 2013.

[7] D. Cade and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications1," *Journal of Computer Security*, vol. 23, no. 3, pp. 331–402, 2015.

[8] P. B. Copet, R. S. A. Pironti, D. Pozza, and P. Vivoli, "Visual model-driven design, verification and implementation of security protocols," in *The 14th IEEE International Symposium on High-Assurance Systems Engineering*, pp. 62–65, Omaha, Nebraska, USA., Oct. 2012.

[9] P. B. Copet and R. Sisto, "Automated formal verification of application-specific security properties," in *The 6th International Symposium on Engineering Secure Software and Systems*, LNCS 8364, pp. 45–59, Springer, 2014.

[10] A. Kartit, H. K. Idrissi, and M. Belkhouraf, "Improved methods and principles for designing and analyzing security protocols," *International Journal of Network Security*, vol. 18, no. 3, pp. 523–528, 2016.

[11] X. H. Li, D. Li, S. T. Li, and J. F. Ma, "Multi-language oriented automatic realization method for cryptographic protocols," *Journal on Communications*, vol. 33, no. 9, pp. 152–159, 2012.

[12] B. Meng, Z. M. Li, and W. Chen, "Mechanized verification of cryptographic security of cryptographic security protocol implementation in java through model extraction in the computational model," *Journal of Software Engineering*, vol. 9, no. 1, pp. 1–32, 2015.

[13] P. Modesti, "Efficient java code generation of security protocols specified in anb/anbx," in *the 10th International Workshop on Security and Trust Management*, pp. 204–208, Wroclaw, Poland, Sep. 2014.

[14] A. Pironti, D. Pozza, and R. Sisto, "Formally based semi-automatic implementation of an open security protocol," *Journal of Systems and Software*, vol. 85, no. 4, pp. 835–849, 2012.

[15] A. Pironti and R. Sisto, "Safe abstractions of data encodings in formal security protocol models," *Formal Aspects of Computing*, vol. 26, no. 1, pp. 125–167, 2014.

[16] A. Pirontia and R. Sisto, "Provably correct java implementations of spi calculus security protocols specifications," *Computers & Security*, vol. 29, no. 3, pp. 302–314, 2010.

[17] Q. Qian and Y. Long, J. R. Zhang, "A lightweight rfid security protocol based on elliptic curve cryptography," *International Journal of Network Security*, vol. 18, no. 2, pp. 354–361, 2016.

[18] J. Quaresma and C. Probst, "Protocol implementation generator," in *The 15th Nordic Conference on Secure IT Systems*, pp. 256–268, Espoo, Finland, Oct. 2010.

Bo Meng was born in 1974 in China. He received his M.S. degree in computer science and technology in 2000 and his Ph.D. degree in traffic information engineering and control from Wuhan University of Technology at Wuhan, China in 2003. From 2004 to 2006, he worked at Wuhan University as a postdoctoral researcher in information security. From 2014 to 2015, he worked at University of South Carolina as a Visiting Scholar. Currently, he is a full Professor at the school of computer, South-Center University for Nationalities, China. He has authored/coauthored over 50 papers in International/National journals and conferences. In addition, he has also published a book "secure remote voting protocol" in the science press in China. His current research interests include security protocols and formal methods.

Chin-Tser Huang received his Ph.D. in Computer Science at the University of Texas at Austin, Austin, Texas and is now an associate professor in the Department of Computer Science and Engineering at the University of South Carolina. He is also the director of the Secure Protocol Implementation and Development (SPID) Laboratory at the University of South Carolina. His current research interests include network security, network protocol design and verification, secure computing, and distributed systems.

Yitong Yang was born in 1991 and is now a postgraduate at the school of computer, South-Center University for Nationalities, China. Her current research interests include security protocols and formal methods.

Leyuan Niu was born in 1990 and currently is a postgraduate in the school of computer, South-Center University for Nationalities, China. Her current research interests include security protocols and formal methods.

Dejun Wang was born in 1974 and received his Ph.D. in information security at Wuhan University in China. Currently, he is an associate professor in the school of computer, South-Center University for Nationalities, China. He has authored/coauthored over 20 papers in international/national journals and conferences. His current research interests include security protocols and formal methods.