Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 9 (1): 1-32, 2015
ISSN 1819-4311 / DOI: 10.3923/jse.2015.1.32
© 2015 Academic Journals Inc.

Mechanized Verification of Cryptographic Security of Cryptographic Security Protocol Implementation in JAVA through Model Extraction in the Computational Model

Zimao Li, Bo Meng, Dejun Wang and Wei Chen

School of Computer, South-Central University for Nationalities, Min Yuan Road No. 708, Hong Shan Section, Wuhan, Hubei, 430074, China

Corresponding Author: Bo Meng, School of Computer, South-Central University for Nationalities, Min Yuan Road No. 708, Hong Shan Section, Wuhan, Hubei, 430074, China Tel: 0086-18602707481

ABSTRACT

Many security protocols have been designed, implemented and deployed in various types of information systems in the last several decades. Hence, the security properties of security protocols have received serious attention. The hot issue has changed from the analysis of security properties of security protocol abstract specifications to the analysis of security properties of security protocol implementations. In this study, the model of analysis and verification of cryptographic security in security protocol implementations is presented. Therefore, in addition, the SubJAVA language is presented which is a subset of JAVA language and an automatic verifier SubJAVA2CV is developed which is able to transform security protocols written in SubJAVA to the security protocol abstract specification written in Blanchet calculus in the computational model. Finally, we use the automatic verifier SubJAVA2CV and CryptoVerif to analyze the authentication of a Needham-Schroeder protocol implementation in SubJAVA.

Key words: Security protocol implementations, automatic verification, computational model, JAVA

INTRODUCTION

In the last several decades, from the design of security protocols, to the analysis and verification of security protocols, to the implementation of security protocols, researcher have carried out much work on the analysis and proof of security protocol abstract specifications. However, we know that the objective of the development of a security protocol is the implementations and deployment of the security protocol in information systems. Thus, the current hot issue in the security protocol field has changed from the analysis of security protocol abstract specifications to the analysis of security protocol implementations, owing to the fact that proof of the security of security protocol abstract specifications is not enough to inspire strong confidence in people. At the same time, although the security of security protocol abstract specifications is proven, security protocol implementations may contain errors or be otherwise insecure. Thus, it is very important to analyze and prove the security properties of security protocol implementations and we need to research security protocols at the code level.

To obtain secure security protocol implementations, there are two approaches. One approach is code generation used for non-existing implementations for some security protocols. For code generation from security protocol abstract specifications, we first generate the security protocol

abstract specification represented in a formal language, for example, pi calculus and probabilistic process calculus and then we use the security protocol analyzer/proveres, for example, ProVerif or CryptoVerif, to analyze the security properties of security protocol abstract specifications. After that, we use or develop the transformer to translate security protocol abstract specifications to security protocol implementations.

The other approach is for existing security protocol implementations. The approach can be subdivided into two categories. One category is mainly based on the technologies of program analysis and verification, for example, logic, type theory and the combination of logic and type theory which are directly used to automatically verify its cryptographic security. The approach also depends on a significant number of annotations and predicates/assertions added to the security protocol implementations. The other category is model extraction which first extracts the security protocol abstract specification from security protocol implementations and then uses the security protocol analyzer/prover to analyze or prove the cryptographic security of the security protocol implementations through proof of the security protocol abstract specification. The approach is sound and is suitable to analyze or prove the cryptographic security of security protocol implementations on a small scale.

In this method, based on symbolic models, Bhargavan *et al.* (2008) developed an automatic tool, FS2PV, that extracts security protocol abstract specifications which are inputs of ProVerif from security protocol implementations written in the F# language and then uses ProVerif to automatically prove securities of security protocol implementations in F# through the security of security protocol abstract specifications. Aizatulin *et al.* (2011) introduced a method to use symbolic execution for security protocol implementations written in the C language to obtain symbolic descriptions for the network messages and then obtain security protocol abstract specifications represented in process calculus by algebraic rewriting which can be analyzed by ProVerif. In computation models, Bhargavan *et al.* (2009) developed an automatic tool, FS2CV, that extracts security protocol abstract specifications which are inputs of CryptoVerif (Blanchet, 2008) from security protocol implementations written in the F# language and then uses CryptoVerif to automatically prove security protocol implementations in F# through the security protocol abstract specification. Aizatulin *et al.* (2012) proposed a method to obtain security protocol abstract specifications represented in Blanchet calculus from security protocol implementations written in the C language which can be analyzed by CryptoVerif.

According to the related references, the mechanized verification of cryptographic securities in security protocol implementations in the JAVA language through model extraction in computational model has not been previously examined. Thus, in this study, we mechanized the verification of the cryptographic security in security protocol implementations written in the JAVA language in computational model.

Many security protocols have been designed, implemented and deployed in various types of information systems. Hence, the security properties of security protocols have received serious attention. In the beginning of the 1980s, two models were proposed to analyze security protocols. One model is the symbolic model, also called the Dolev-Yao model, in which cryptographic primitives are abstracted as black boxes. The attacker only performs some computation based on cryptographic primitives. The other model is the computational model, in which cryptographic primitives are modeled as functions on bitstrings. The adversary is any probabilistic polynomial-time turing machine. The computation model is more realistic but it was difficult to mechanize proof until the introduction of the mechanized tool CryptoVerif.

Currently, the hot issue in the security protocol field has changed from the analysis of security protocol abstract specifications to the analysis of security protocol implementations. To obtain security protocol implementations with security in computational model, there are mainly two approaches used: Code generation and model extraction. Model extraction is sound and is suitable for analyzing or proving the cryptographic security of security protocol implementations on a small scale. According to the related references, the mechanized verification of cryptographic security in security protocol implementations in JAVA language through model extraction in computational model has not been previously examined. Thus, in this study, we verify cryptographic security in security protocol implementations in the computational model.

Main contributions of this study are summarized as follows:

- The status of analysis in mechanized verification of cryptographic security in security protocol implementations, including in the symbolic model and in the computational model, is presented. We find that security protocol implementations in the C, F# languages are analyzed in the symbolic model or in the computational model. Until now, there has not existed an analysis of the security properties of security protocol implementations in JAVA through model extraction in the computational model
- We proposed a model of analysis and verification of cryptographic security in security protocol implementations, shown in Fig. 1. For the model extraction, the security protocol implementations in the source language SP(S), for example, the JAVA language, are transformed to security protocol implementations in the target language SP(T), for example, Blanchet calculus; then, the security protocol implementations in the target language SP(T),

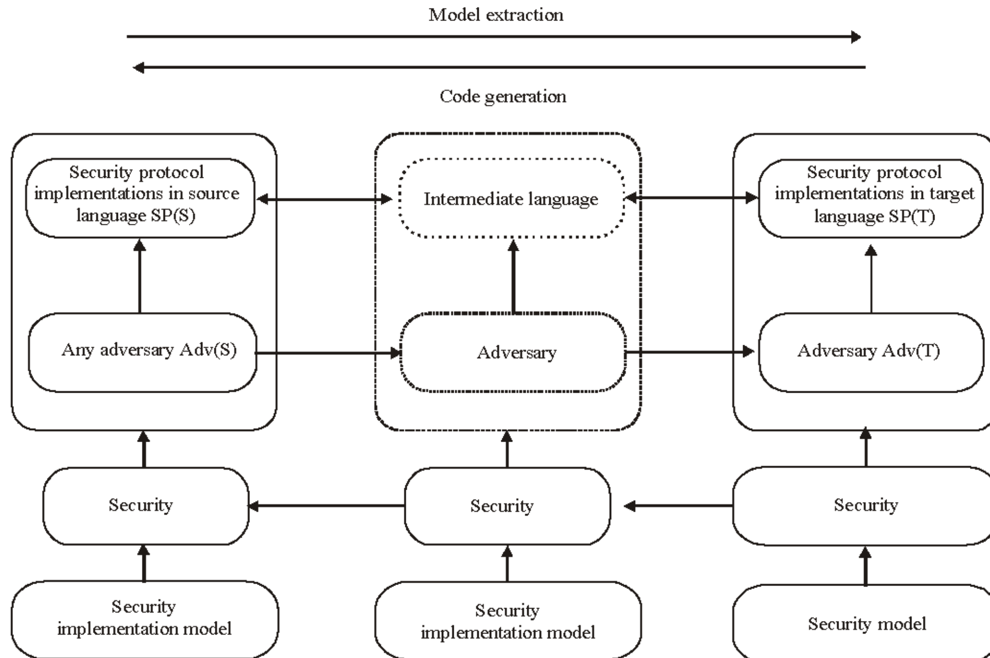

Fig. 1: Model of analysis and verification of cryptographic security in security protocol implementations

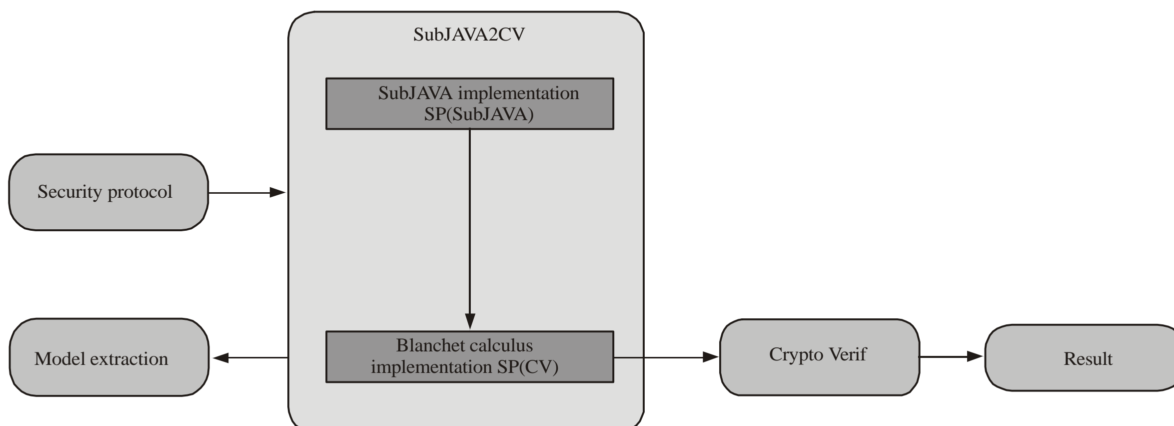

Fig. 2: Model of an automatic tool SubJAVA2CV

are analyzed by the automatic security protocol analyzer/prover. For the code generation, the security protocol implementations in the target language SP(T), for example, Blanchet calculus, are transformed to the security protocol implementations in the source language SP(S), for example, the JAVA language. It also needs to be proven that, if the security protocol implementations in the target language SP(T), for the adversary Adv(T), constructed according to any adversary Adv(S), is secure, then the security protocol implementations in the source language SP(S), for any adversary Adv(S), is also secure

- We presented the SubJAVA language which is a subset of JAVA language and developed an automatic verifier, SubJAVA2CV, that can translate security protocols written in SubJAVA to the security protocol abstract specification which is input to CryptoVerif, a mechanized prover for security protocols in the computational model developed by Blanchet. Then, we use CryptoVerif to automatically prove the security protocol implementation in SubJAVA through the security protocol abstract specification. Figure 2 shows the model of the automatic verifier SubJAVA2CV
- The automatic verifier SubJAVA2CV and CryptoVerif are used to analyze the authentication of the Needham-Schroeder protocol implementation in SubJAVA. First, we develop the Needham-Schroeder protocol implementation in SubJAVA and then we use the automatic verifier SubJAVA2CV to generate the Needham-Schroeder protocol implementation in Blanchet calculus; finally, CryptoVerif is used to perform the analysis of the authentication of the Needham-Schroeder protocol implementation in Blanchet calculus. The analysis result is consistent with the analysis result of the Needham-Schroeder protocol abstract specification

The state of art in mechanized verification of cryptographic security in security protocol implementations is presented here. The approaches of mechanized verification of cryptographic security in security protocol implementations can be classified into two categories. One is mainly based on the technologies of program analysis and verification, for example, logic, type theory and the combination of logic and type theory which are directly used to automatically verify its cryptographic security. This approach depends on a significant number of annotations and predicates/assertions added to the security protocol implementations. In the symbolic model,

Goubault-Larrecq and Parrennes (2005) developed the first mechanized tool, CSur, to verify the SSL protocol implementation in the C language. The method first uses the control flow graph to model the execution of the SSL protocol implementation in the C language and then generates the abstract model of security based on trust assertions by hand; finally, it uses the H1 solver to verify its security. Jurjens (2009) used automated first-order logic provers to verify the JAVA implementation, Jessie, of the SSL protocol. He first used the control flow graph to model the behaviors of the security protocol implementations and then adds the assertions by hand to the abstract model based on HOL formulas; finally, he used the ATP to verify the confidentiality.

Based on technology regarding the invariants in security protocol implementations in F# language, Bhargavan *et al*. (2010) developed a type checker for F7 based on an SMT solver to verify its security. Bengtson *et al*. (2011) also developed a type checker for the functional language F# to verify security properties of the source code of cryptographic protocols in F#. The type checker generates verification conditions for security properties by annotating programs with preconditions and post conditions by hand and sends it to the SMT solver to verify its security. Swamy *et al*. (2011) also developed a type checker for F7 and a compiler that translates F7 to NET bytecode to verify the security of security protocol implementations in F7. Backes *et al*. (2014) presented a new type system for verifying the authentication of reference implementations of cryptographic protocols written in a core functional programming language. Chaki and Datta (2009) developed an automatic tool, ASPIER, that is based on the iterative abstraction-refinement method and control flow graph with a standard protocol security model to automatically analyze the authentication and secrecy of OpenSSL protocol implementations in the C language. They find the version-rollback attack in the OpenSSL 0.9.6c source code. Dupressoir *et al*. (2011) used the general-purpose verifier VCC to verify security properties of C code for cryptographic protocols by writing suitable header files and annotations in implementation files. In the computation model, Backes *et al*. (2010) used the CoSP framework to prove the soundness of the type checker F7. In other words, they give some conditions under which the symbolic security of RCF programs using cryptographic idealizations implies computational security of the same programs using cryptographic algorithms. Apart from that, they implement a computationally sound automated verification of F# code containing public-key encryptions and signatures.

The other approach is model extraction which first extracts the abstract security protocol specification from security protocol implementations, then uses the security protocol analyzer/prover to analyze or prove the cryptographic security of security protocol implementations through proof of the abstract security protocol specification. The approach is sound and is suitable to analyze or prove the cryptographic security of security protocol implementations on a small scale. In the symbolic model, Bhargavan *et al*. (2008) developed an automatic tool, FS2PV, that can translate security protocols written in the F# language to abstract security protocol specifications which are inputs of ProVerif. Then, they use ProVerif to automatically prove a security protocol implementation in F# through the abstract security protocol specification. Based on special reference implementations, O'Shea (2008) designed an automatic tool, Elyjah, that transforms security protocol implementations in JAVA to abstract security protocol specifications written in LySa process calculus which can be analyzed by the automatic tool LySatool for authentication. Aizatulin *et al*. (2011) introduce a method using symbolically executed security protocol implementations written in the C language to obtain symbolic descriptions for network messages and then obtain abstract security protocol specifications represented in process calculus by algebraic rewriting which can be analyzed by ProVerif. In the computational model, Bhargavan *et al*. (2009)

develop an automatic tool, FS2CV, that can translate security protocols written in F# to the corresponding abstract security protocol specification which is an input to CryptoVerif, a mechanized prover for security protocols in the computational model. They then use CryptoVerif to automatically prove security protocol implementations in the F# language through their abstract security protocol specifications. Aizatulin *et al.* (2012) proposed a method to obtain the abstract security protocol specification represented in Blanchet calculus from a security protocol implementation written in the C language which can be analyzed by CryptoVerif.

According to the previous discussion, there previously did not exist an analysis of security protocol implementations in JAVA with the mechanized tool CryptoVerif in the computational model.

PROPOSED MODEL OF ANALYSIS AND VERIFICATION OF CRYPTOGRAPHIC SECURITY IN SECURITY PROTOCOL IMPLEMENTATIONS

Usually, the objective is to ensure that security protocol implementations are secure. Hence, the analysis and verification of cryptographic security in security protocol implementations should be implemented and verified. Generally, there are two important approaches that can be used. One approach is model extraction is shown in Fig. 3. The other approach is code generation is shown in Fig. 4. Model extraction is suitable for the existing security protocol implementations. Code generation is suitable for non-existing security protocol implementations. In the following, there are two models introduced: One for model extraction and the other for code generation.

Model of analysis and verification of cryptographic security in security protocol implementations for model extraction: For model extraction, the existing security protocol implementation SP(S), is written in programming languages, for example, JAVA, C# and C while the target language of the security protocol implementation SP (T), is formal languages (e.g., Pi calculus, Blanchet calculus and SPI calculus). In other words, the source languages are program languages and the target languages are formal languages. If the model extraction is used to analyze and verify the cryptographic security in security protocol implementations, then there are two conditions that need to be proven:

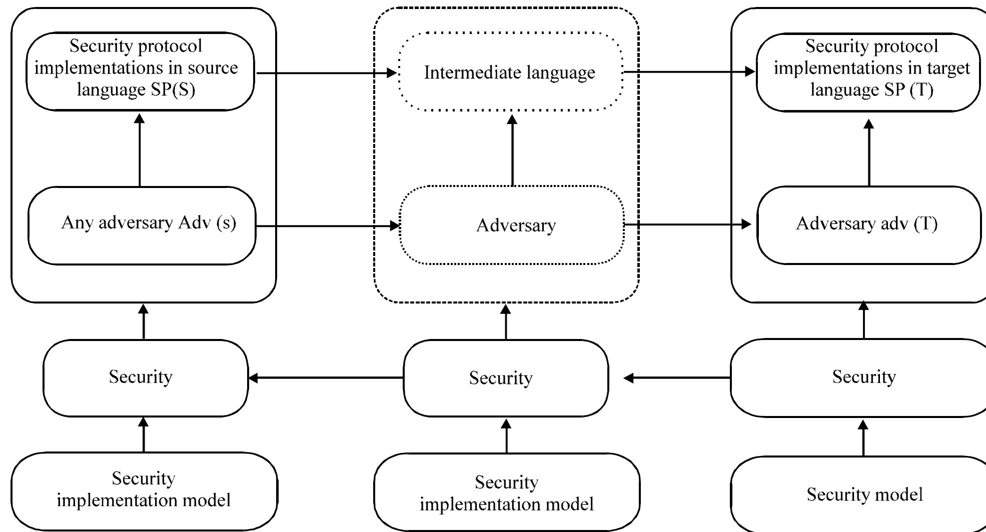

Fig. 3: Model for model extraction

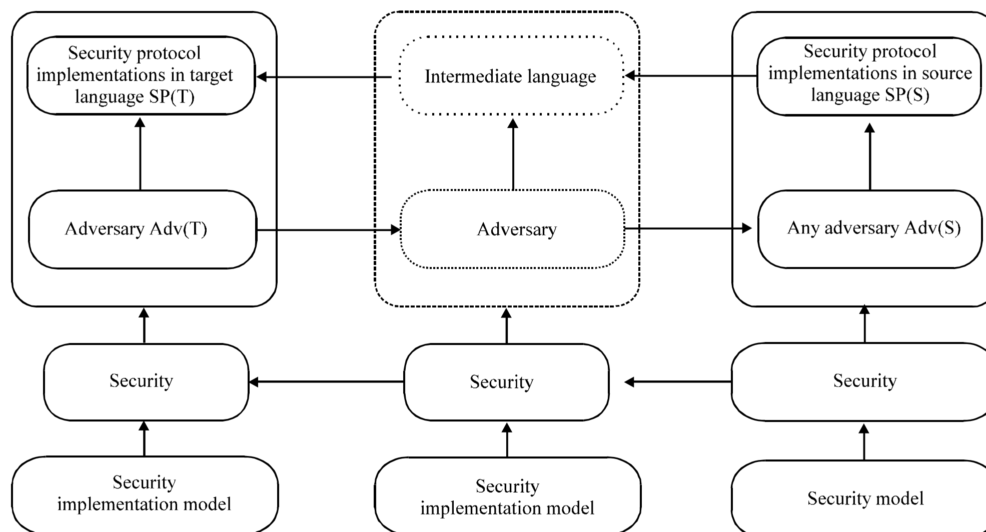

Fig. 4: Model for code generation

- Semantics of the source language simulate the semantics of the target language
- If the security protocol implementation in the target language SP (T) for the adversary Adv(T), constructed according to any adversary Adv(S), is secure, then the security protocol implementation in the source language SP(S) for any adversary Adv(S) is secure

Condition one describes the relationship between security protocol implementations in the source language SP(S) and security protocol implementations in the target language SP (T) and the relationship is a simulation or observational equivalence from the view of the behaviors of security protocol implementations in the source language SP(S) and security protocol implementations in the target language SP (T).

Condition two describes the method of proving the cryptographic security of security protocol implementations in the source language SP(S). The objective is to prove the cryptographic security of security protocol implementations in the source language SP(S) with any adversaries. Hence, we should construct an adversary Adv (T) in the security protocol implementations in the target language SP (T) according to the adversary Adv(S) in the security protocol implementations in the source language SP(S); then, we prove that the security protocol implementations in the target language SP (T) for the adversary Adv(T) are secure and therefore, the security protocol implementations in the source language SP(S) for any adversary Adv(S) are secure.

Model of analysis and verification of cryptographic security in security protocol implementations for code generation: For code generation, the existing security protocol implementation SP(S) is written in formal languages, for example, Pi calculus, Blanchet calculus and SPI calculus while the target language of the security protocol implementation SP (T) is programming languages, for example, JAVA, C# and C. In other words, the source languages are formal languages and the target languages are programming languages. If code generation is used to analyze and verify the cryptographic security in security protocol implementations, then there are two conditions that need to be proven:

- Semantics of the source language simulate the semantics of the target language
- If the security protocol implementation in the source language SP(S) for the adversary Adv(S), constructed according to any adversary Adv(T), is secure, then the security protocol implementation in the target language SP(T) for any adversary Adv(S) is secure

Condition one describes the relationship between security protocol implementations in the source language SP(S) and security protocol implementations in the target language SP(T) and the relationship is a simulation or observational equivalence from the view of the behaviors of security protocol implementations in the source language SP(S) and security protocol implementations in the target language SP(T).

Condition two describes the method of proving the cryptographic security of security protocol implementations in the target language SP(T). The objective is to prove the cryptographic security of the security protocol implementations in the target language SP(T) in the context of any adversaries. Hence, we must construct an adversary Adv(S) in the security protocol implementations in the source language SP(S) according to any adversary Adv(T) in the security protocol implementations in the target language SP(T); then, we prove that the security protocol implementations in the source language SP(S) for the adversary Adv(S) are secure and therefore, the security protocol implementations in the target language SP(T) for any adversary Adv(T) are secure.

SUBJAVA LANGUAGE AND BLANCHET CALCULUS

SubJAVA language: We know that the JAVA language is a very complex programming language. To extract the model from a security protocol implementation in JAVA, we should reduce the complexity of JAVA. One method is to use only specified expressions and statements in JAVA which is called SubJAVA. SubJAVA is made up of expressions, statements, modifiers, member variables and methods.

SubJAVA expressions are presented in Fig. 5. The a, b, c denote the value and type x, y, z is a type declaration expression, x, y, z are basic data types, such as short, byte, int, float, long, double, char or Boolean type, x, y, z are variables, $e_1|e_2$ is the conditional OR expression, $e_1\delta e_2$ is

e ::=	Expression
a, b, c	Value
type x, y, z	Variable_declare
x, y, z	Variable
$e_1 \mid e_2$	ConditionalOr
$e_1 \delta e_2$	ConditionalAnd
$e_1 == e_2$	Equality
y = b	Assign
y = x	Assign
new Class $(e_1, \cdots, e_m)$	New operation
Class.Method $(e_1, \cdots, e_m)$	Call expression

Fig. 5: SubJAVA expressions

the conditional AND expression. The $e_1 == e$ is the equality expression, y = b and y = x are assignment expressions. New class $(e_1, ..., e_m)$ is the new operation expression. Class method $(e_1, ..., e_m)$ is the call operation expression.

Figure 6 lists SubJAVA statements. SubJAVA statements consist of the empty statement, expression statement e;, block statement {S*}, if-else statement if(e) P else Q, return statement return e; throw statement throw e; and try-catch statement try S catch (e) P. We can use those statements to implement security protocols.

Figure 7 presents SubJAVA modifiers. SubJAVA provides several access modifiers to set access levels for classes, variables, methods and constructors. Public, Protected and private are access modifiers. The public modifier means that classes, variables, methods and constructors are visible to the world. The private modifier means that classes, variables, methods and constructors are visible to the class only. The protected modifier means that classes, variables, methods and constructors are visible to the package and all subclasses. The static modifier static is used for creating class methods and variables. These modifiers are closely related to security models of security protocol implementations and the abilities of adversaries.

A SubJAVA member variable and method in a class are presented in Fig. 8. A SubJAVA method is a collection of statements that are grouped together to perform an operation. When we call a method, the system actually performs several statements in the method.

Blanchet calculus: Blanchet calculus is a probabilistic polynomial calculus and has been used to prove security properties of security protocols. Messages are bitstrings and cryptographic primitives are functions that operate on bitstrings. Blanchet calculus consists of terms, condition and processes. The adversary is modeled by an evaluation context, C.

The terms in Fig. 9 (replication index i, variable access $x[M_1, ..., M_m]$ or function application $f[M_1, ..., M_m]$) represent computations on bitstrings. The replication index i is an integer that serves to distinguish different copies of a replicated process$v!i \le n$. The variable accesses $x[M_1, ..., M_m]$ and

S ::=	Statement
;	Empty statement
e;	Expression
$\{S^*\}$	Block statement
if (e) P else Q	If statement
return e;	Return statement
throw e;	Throw statement
try S catch (e) P	Try statement

Fig. 6: SubJAVA statements

$$\left\| \begin{array}{l} \text{Modifiers ::=} \\ \qquad \left| \text{public} \mid \text{static} \mid \text{protected} \mid \text{private} \right| \end{array} \right\|$$

Fig. 7: SubJAVA modifiers

$$\left[\begin{array}{lll} M, N ::= & & \text{Member} \\ & |\ \text{Modifiers type } x, y, z; & \text{Field_declare} \\ & |\ \text{Modifiers } x = a, y = b, z = c; & \text{Field_declare} \\ & |\ \text{Modifiers type Class}(e_1, ..., e_m)\{S^*\}; & \text{Constructor} \\ & |\ \text{Modifiers type Method}((e_1, ..., e_m)\{S^*\}; & \text{Method_declare} \end{array}\right]$$

Fig. 8: SubJAVA member variable and method

$$\left[\begin{array}{lll} M, N ::= & \text{terms} & \\ & |\ i & \text{replication index} \\ & |\ x[M_1, \cdots, M_m] & \text{variable access} \\ & |\ f[M_1, \cdots, M_m] & \text{function application} \end{array}\right]$$

Fig. 9: Terms

$$\left[\begin{array}{lll} \text{Cond} ::= & & \\ & |\ M = N & \text{Equality} \\ & |\ M <> N & \text{Inequality} \\ & |\ M \,||\, N & \text{Disjunction} \\ & |\ M \,\&\&\, N & \text{Conjunction} \end{array}\right]$$

Fig. 10: Conditions

$$\left[\begin{array}{lll} Q ::= & & \\ & |\ 0 & \text{Nill} \\ & |\ Q|Q' & \text{Parallel composition} \\ & |\ !^{i \le n} Q & \text{Replication n times} \\ & |\ \text{newChannel } c; Q & \text{Channel restriction} \\ & |\ \text{in}[M_1, \cdots, M_l]\left(c, \left(x_1[\tilde{i}] : T_1, \cdots, x_k[\tilde{i}] : T_k\right)\right); P & \text{Input} \end{array}\right]$$

Fig. 11: Input process

returns the content of the cell of indices M_1, ..., M_m of the m-dimensional array variable x. The function application f (M_1, ..., M_m) returns the result of applying function f to M_1, ..., M_m. The condition is described in Fig. 10 which includes equality, inequality, disjunction and conjunction conditions.

Processes are made up of the input processes in Fig. 11 and the output processes in Fig. 12. Input process Q receives a message on a channel; output process P outputs a message on a channel after executing some internal computations. Input processes can be nil0, parallel composition Q|Q', replication times !I≤nQ, channel restriction new channel c; Q: in $[M_1, \cdots, M_l]\left(c, \left(x_1[\tilde{i}] : T_1, \cdots, x_k[\tilde{i}] : T_k\right)\right); P$

$$
\left[\!\left[\begin{array}{ll}
P ::= & \\
\quad \mathrm{out}[M_1,\cdots M_l]\langle c,(N_1,\cdots,N_k)\rangle;Q & \text{Output} \\
\quad \mathrm{new}\ x[i_1,\cdots,i_m]:T;P & \text{Random number} \\
\quad \mathrm{let}\ x[i_1,\cdots,i_m]:T=M\ \mathrm{in}\ P & \text{Assignment} \\
\quad \mathrm{if\ defined}(M_1,\cdots M_l)\wedge M\ \mathrm{then}\ P\ \mathrm{else}\ P' & \text{Conditional} \\
\quad \mathrm{find}\ \left(\oplus_{j=1}^{m} u_{j1},\cdots,u_{jm_j}\left[\tilde{i}\right]\leq n_{jm_j}\mathrm{suchthat\ defined}\ \left(M_{j1},\cdots,M_{jlj}\right)\wedge M_j\ \mathrm{then}\ P_j\right)\ \mathrm{else}\ P & \text{Array lookup} \\
\quad \mathrm{event}\ e(M_1,\cdots,M_m);P & \text{Event}
\end{array}\right]\!\right]
$$

Fig. 12: Output process

input process 0 does nothing; Q|Q' is the parallel composition of Q and Q'; $!^{i\leq n}Q$ represents n copies of Q in parallel; newchannel c;Q creates a new private channel n and executes Q. Output processes are made up of out $[M_1, ..., M_l]\langle c, (N_1, ..., N_k)\rangle$;Q random number $x(i_1, ..., i_m)$: T; P, assignmunt let $x[i_1, ..., i_m]$: T = M in P, conditional if defined $(M_1, ..., M_l) \wedge$ M then P else P, array lookup find $(\oplus_{j=1}^{m} u_{j1}, L, u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ such that defined $(M_{j1}, ..., M_{jlj}) \wedge M_j$ then P_j else P and event event $e(M_1, ..., M_m)$. The output process new $x[i_1, ..., i_m]$: T; P chooses a new random number uniformly in $I_\eta(T)$, stores it in $x[i_1, ..., i_m]$ and then performs P. Random numbers must be chosen by new $x[i_1, ..., i_m]$: T. The output process let $x[i_1, ..., i_m]$: T = M in P stores the bitstring value of M in $x[i_1, ..., i_m]$ and performs P. Find $(\oplus_{j=1}^{m} u_{j1},..., u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ such that defined $(M_1, ..., M_m) \wedge M_j$ then P_j) else P means that it tries to find a branch J in [1, m] such that there are values of $u_{ji}, ..., u_{jm}$ for which $M_{ji}, ..., M_{jm}$ are defined and M_j is true. In the case of success, it performs P. In the case of failure for all branches, it performs P. The formula event $e(M_1, ..., M_m)$ is true when event $e(M_1, ..., M_m)$ has been performed.

MODEL EXTRACTION FROM SECURITY PROTOCOL IMPLEMENTATIONS IN SUBJAVA TO SECURITY PROTOCOL IMPLEMENTATIONS IN BLANCHET CALCULUS

We specify that security protocol implementations in SubJAVA are made up of classes. Apart from that, in security protocols there are several roles, such as sender, receiver and so on. Thus, we define that each role is also modeled as a class in SubJAVA. According to the policy in Blanchet calculus, the roles in security protocols are modeled as processes. Therefore, there is a mapping relation from a class in SubJAVA to a process in Blanchet calculus. In the SubJAVA language, many classes and APIs are defined to implement the transmission of messages, for example, Socket, ServerSocket, Inputstream and Outputstream. Here, we use JAVASend and JAVAReceive to express the sending and receiving of messages. In Blanchet calculus, channels are used to model the classes and APIs responsible for the sending and receiving of messages. Hence, there are mapping relations between JAVASend/JAVAReceive and channels. Finally, in the call method, arithmetic calculations in SubJAVA are mapped to functions in Blanchet calculus. Fig. 13 describes model extraction of Blanchet calculus from SubJAVA.

Figure 14 presents extraction functions from SubJAVA to Blanchet calculus in detail. The extraction function ExtractorValue transforms value α in SubJAVA into const α in Blanchet

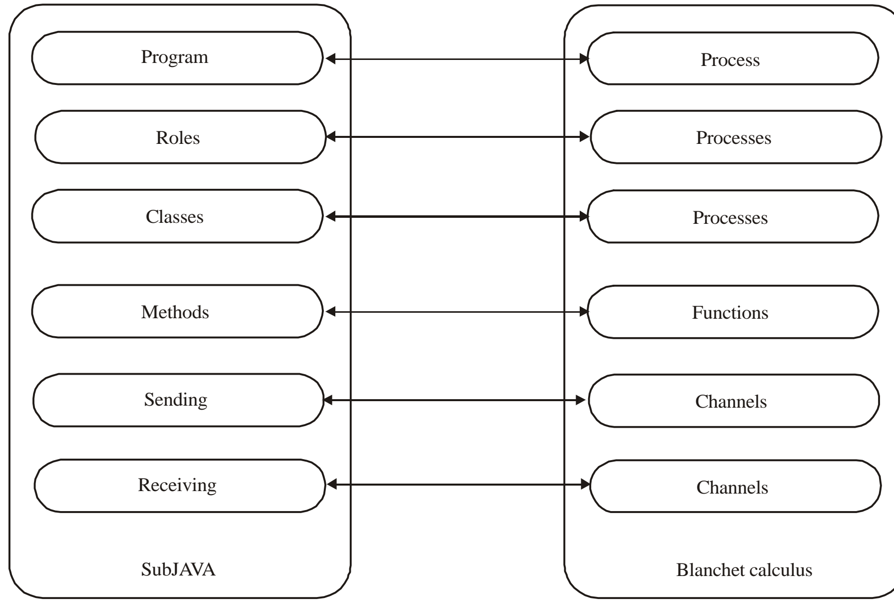

Fig. 13: Model extraction of Blanchet calculus from SubJAVA

$$
\left[\!\left[
\begin{array}{l}
\text{ExtractorValue}(a) = \text{const}\ \ a \\
\text{ExtractorVariable}(x) = x[:T] \\
\text{ExtractorVariableDeclaration}(\text{type } x) = x[:\text{type}] \\
\text{ExtractorExpression}(e) = M \\
\text{ExtractorOr}(e_1 \mid e_2) = \{\text{ExtractorExpression}(e_1) \,\|\, \text{ExtractorExpression}(e_2)\} \\
\text{ExtratorAnd}(e_1 \delta e_2) = \{\text{ExtractorExpression}(e_1)\ \&\&\text{ExtractorExpression}(e_2)\} \\
\text{ExtractorEquality}(e_1 == e_2) = \{\text{ExtractorExpression}(e_1) = \text{ExtractorExpression}(e_2)\} \\
\text{ExtractorInequality}(e_1 \,!{=}\, e_2) = \{\text{ExtractorExpression}(e_1) <> \text{ExtractorExpression}(e_2)\} \\
\text{ExtractorAssign}(y = b) = \text{let}\ \ y : T = b\ \ \text{in} \\
\text{ExtractorNew}(\text{new Class}(e_1, \cdots, e_m)) = \text{new}\ \ x : T \\
\text{ExtractorCall}(\text{Class.Method}(e_1, \cdots, e_m)) = f[\text{ExtractorExpression}(e_1), \cdots, \text{ExtractorExpression}(e_1)] \\
\text{ExtractorEmpty}(\ \ ) = 0 \\
\text{ExtractorStatement}(S) = P \,\|\, Q \\
\text{ExtractorBlock}(\{S^*\}) = \text{ExtractorStatement}(S^*) \\
\text{ExtractorIf}(\text{if}\,(e)\,P \text{ else } Q) = \text{if}\ \langle \text{ExtractorExpression}(e) \rangle \text{ then } \langle \text{outprocess} \rangle\ [\text{else}\ \langle \text{outprocess} \rangle] \\
\text{ExtractorReturn}(\text{return } e) = \text{out}(\langle \text{channel} \rangle, \text{ExtractorExpression}(e))[;\langle \text{inprocess} \rangle] \\
\text{ExtractorSend}(\text{JavaSend}(e)) = \text{out}(\langle \text{channel} \rangle, \text{ExtractorExpression}(e))[;\langle \text{inprocess} \rangle] \\
\text{ExtractorReceive}(\text{Java Receive}(e)) = \text{in}(\langle \text{channel} \rangle, \text{ExtractorExpression}(e))[;\langle \text{inprocess} \rangle] \\
\text{ExtractorType}(\text{Type}) = \text{Type}
\end{array}
\right]\!\right]
$$

Fig. 14: Extraction functions from SubJAVA to Blanchet calculus

calculus. The extraction function ExtractorValue maps variable x in SubJAVA to variable access x:[:T] in Blanchet calculus. The extraction function ExtractorExpression transforms expression e in SubJAVA into term M in Blanchet calculus. The extraction function ExtractorOr transforms the conditional OR $e_1|e_2$ in SubJAVA into the disjunction expression ExtractorExpression(e_1)‖ ExtractorExpression (e_2) in Blanchet calculus. The extraction function ExtractorAnd transforms the conditional AND $e_1\delta e_2$ in SubJAVA into the conjunction expression ExtractorExpression (e_1) and ExtractorExpression(e_2) in Blanchet calculus. The extraction function ExtractorEquality transforms the equality expression e_1 = =e_2 in SubJAVA into the equality expression ExtractorExpression(e_1) = ExtractorExpression(e_2) in Blanchet calculus. The extraction function ExtractorInequality transforms the inequality expression e_1! = e_2 in SubJAVA into the inequality expression ExtractorExpression(e_1)<>ExtractorExpression(e_2) in Blanchet calculus. The extraction function ExtractorAssign transforms the assignment expression y = b in SubJAVA into the assignment expression let y:T = b in Blanchet calculus. The extraction function ExtractorNew transforms the new operation expression new class (e_1, ..., e_2) in SubJAVA into the new random expression new x:T in Blanchet calculus. The extraction function ExtractorCall transforms the call operation Class.Method(e_1, ..., e_m) in SubJAVA into the function f[ExtractorExpression(e_1), ..., ExtractorExpression(e_1)] in Blanchet calculus. The extraction function ExtractorEmpty transforms the empty statement; in SubJAVA into the null process 0 in Blanchet calculus. The extraction function ExtractorStatement transforms the statement S in SubJAVA into the input or output process P‖Q in Blanchet calculus. The extraction function ExtractorBlock transforms the block statement {S*} in SubJAVA into the process ExtractorStatement (S*) in Blanchet calculus. The extraction function ExtractorIf transforms the if statement if(e) P else Q in SubJAVA into the conditional process if ⟨ExtractorExpression(e)⟩ then ⟨outprocess⟩[else⟨outprocess⟩] in Blanchet calculus. The extraction function ExtractorReturn transforms the return statement return e in SubJAVA into the out process out(⟨channel⟩, ExtractorExpression (e))[; ⟨inprocess⟩] in Blanchet calculus. The extraction function ExtractorSend transforms the relative send statement JAVASend (e) in SubJAVA into the out process out(⟨Channel⟩, ExtractorExpression (e))[; ⟨inprocess⟩ in Blanchet calculus. The extraction function ExtractorReceive transforms the relative received statement JavaReceive (e) in SubJAVA into the in process in(⟨Channel⟩, ExtractorExpression(e)) [; ⟨Inprocess⟩] in Blanchet calculus. Regarding the type in SubJAVA, we directly transform it into the correspondent type in Blanchet calculus because, in Blanchet calculus, we can define the required type for the corresponding type in SubJAVA.

AUTOMATIC VERIFIER SUBJAVA2CV FOR SECURITY PROTOCOL IMPLEMENTATIONS IN SUBJAVA

According to the extraction functions introduced in the previous section, we develop an automatic verifier, SubJAVA2CV, that accepts security protocol implementations in SubJAVA as input and produces security protocol models represented by Blanchet calculus as output. After that, the output is processed by CryptoVerif to verify the security of the security protocol implementations in SubJAVA. Figure 15 presents the application of the automatic verifier SubJAVA2CV.

The development of the automatic verifier, SubJAVA2CV, is presented in Fig. 16. First, we prepare the security protocol implementations written in SubJAVA according to informal specifications of the security protocols and then we use a lexical analyzer developed by us to

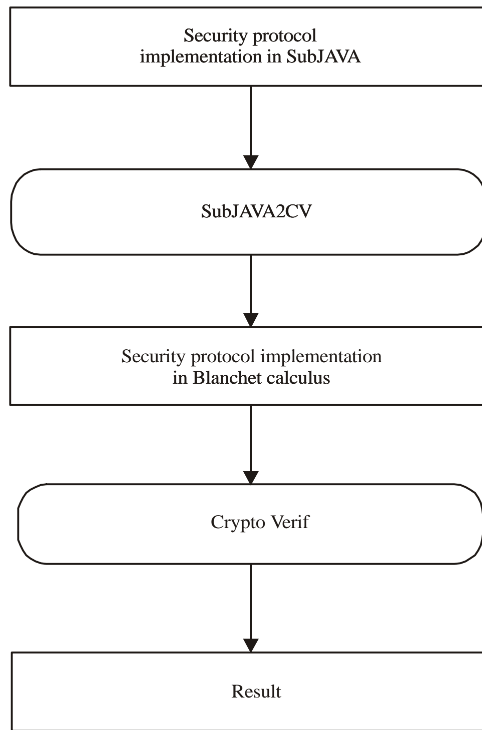

Fig. 15: Application of automatic verifier SubJAVA2CV

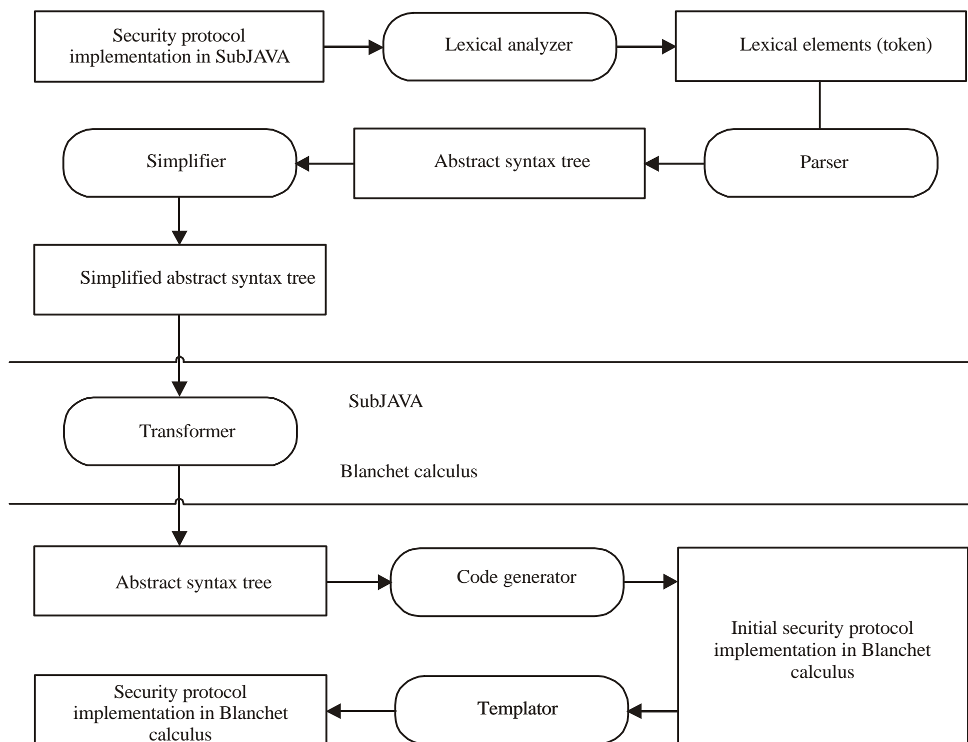

Fig. 16: Development of automatic verifier SubJAVA2CV

analyze and verify the correctness of the security protocol implementations in SubJAVA according to the syntax of SubJAVA. If verification is successful, the lexical elements, for example, tokens, are generated. After that, a parser developed by us is used to address tokens and produce an abstract syntax tree which is used to express the structure of the security protocol implementations in SubJAVA. The objective is to analyze the security of the security protocol implementations in SubJAVA; hence, the structures that are not related to security are removed and the simplified abstract syntax tree is generated. Then, a transformer developed by us is used to map the simplified abstract syntax tree in SubJAVA into an abstract syntax tree in Blanchet calculus. We developed the code generator to generate the initial security protocol implementations in Blanchet calculus. Finally, according to the template in analysis using CryptoVerif, the final security protocol implementations in Blanchet calculus are generated which can be accepted as the input of CryptoVerif. We then can use CryptoVerif to perform the analysis for security.

JAVACC is an Open Source parser generator for JAVA code developed by the SUN corporation. JAVACC generates LL parsers for context free grammars. An LL parser parses the input file from left to right and produces the leftmost derivation for a sentence. These types of parsers use the next tokens to make the parsing decisions without any back tracing. Thus, these LL parsers are simple and hence widely used. We use JAVACC to develop a Lexical analyzer and parser (Fig. 17).

First, we need to construct the .jj file and then we can use JAVACC to implement the Lexical analyzer and parser for SubJAVA.

Lexical analyzer: The lexical analyzer accepts a security protocol implementation in SubJAVA as input and produces a sequence of tokens as output. The responsibility of the lexical analyzer is to verify the grammar of the security protocol implementation in SubJAVA according to the specification of SunJAVA and it divides a sequence of characters into a subsequence called a token.

To develop the lexical analyzer in Fig. 18 in SubJAVA2CV based on JAVACC, first, we need to construct a .jj or .jjt file which is mainly made up of four sections: Options, a class declaration, a specification for lexical analysis and BNF notations. The options section is used to specify the optional parameters. The class declaration in the grammar file is used to provide the main entry point in the generated parser. The specification for lexical analysis and BNF notations sections in

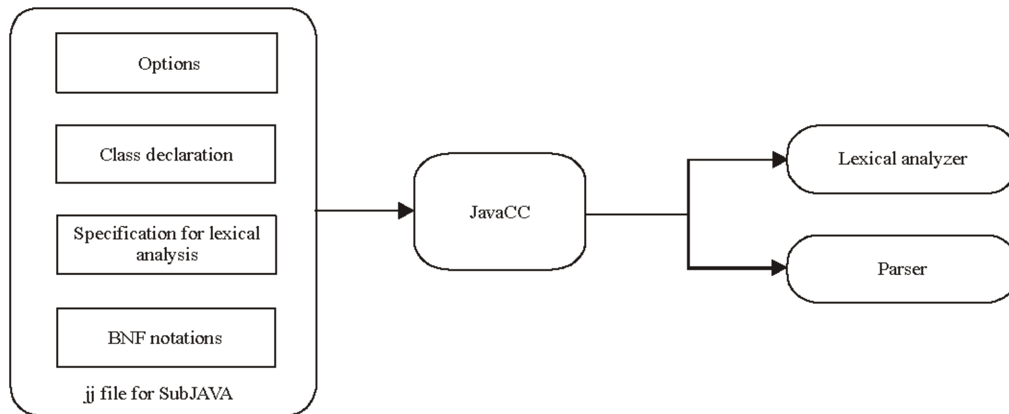

Fig. 17: Lexical analyzer and parser based on JAVACC

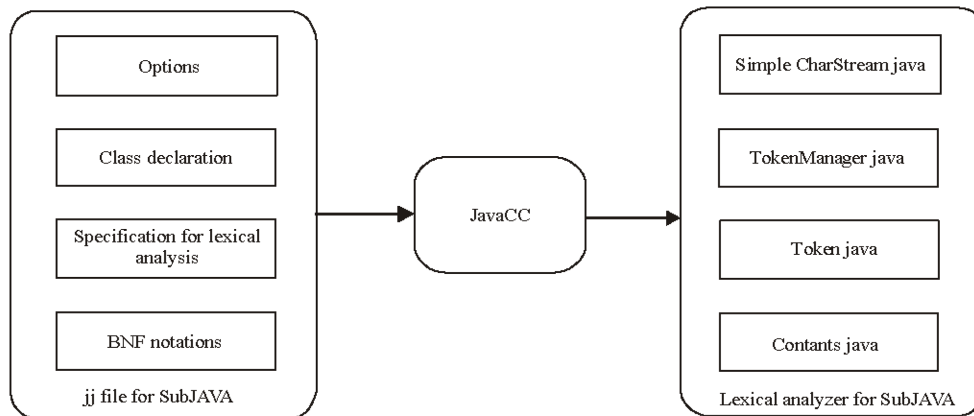

Fig. 18: Lexical analyzer for SubJAVA

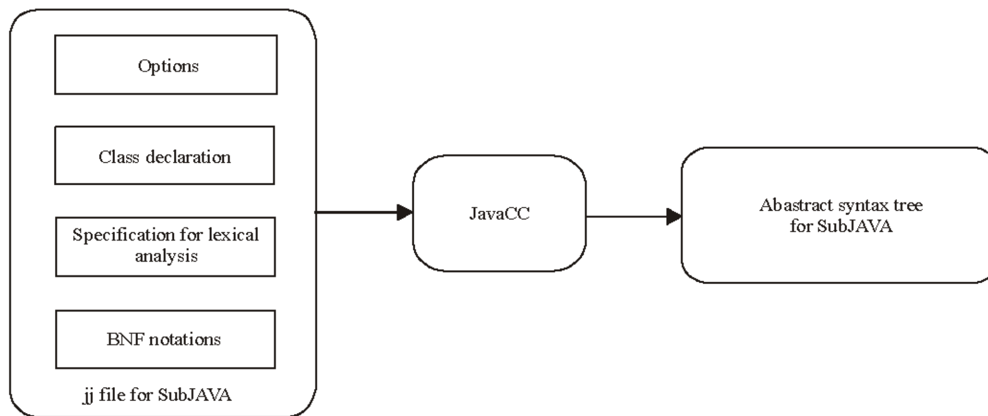

Fig. 19: Parser for SubJAVA

the grammar file are meant for the specification of a token for lexical analysis. At the same time, SKIP and TOKENS are the key part of the grammar file. If some characters are specified as SKIP terminals, they can be skipped while scanning. The tokens introduced in SunJAVA are specified in the TOKEN section.

Then, the lexical analyzer is generated by invoking JAVACC. The lexical analyzer is composed of SimpleCharStream.JAVA, TokenManager.JAVA, Token.JAVA and Constants.JAVA. SimpleCharStream.JAVA is used to deal with and represent the stream of input characters which are SubJAVA source codes and it sends the identified characters to TokenManager JAVA. TokenManager JAVA. Produces a sequence of tokens for SubJAVA source codes. Token.JAVA manages, stores and defines a data structure for the input token. Constants JAVA specifies symbolic names for token classes.

Parser: A parser based on JJtree, shown in Fig. 19, accepts the sequence of tokens for a security protocol implementation written in SubJAVA as input and produces the abstract syntax tree for SubJAVA. An abstract syntax tree is a simplified version of a parse tree (basically a parse tree without non-terminals) while a parse trees tell us exactly how a character string was parsed.

JJtree generates an abstract syntax tree from the bottom up: When the nodes are being built, they are pushed onto a node stack. Then, when a non-terminal has been fully recognized, all of the subnodes are popped from the stack; after that, it is combined into a new node and then the result is pushed back onto the stack. The JJTree defines a JAVA interface Node that all parse tree nodes must be implemented using. The interface provides methods for operations such as setting the parent of the node and for adding children and retrieving them. The JJTree operates in one of two modes, simple and multi. In simple mode, each parse tree node is of concrete type SimpleNode which is the general interface for all Abstract Syntax Tree nodes; in multi mode, the type of the parse tree node is derived from the name of the node. Each node is specified a node scope. User actions within this scope can access the node under construction by using the special identifier jjtThis.

Simplifier: To reduce the abstract syntax tree of a security protocol implementation in SubJAVA, we need to develop a simplifier to perform the task. A simplifier accepts an abstract syntax tree as input and produces the simplified abstract syntax tree. Here, we use the visitor pattern to develop the simplifier.

The visitor pattern is a type of design pattern. The visitor pattern specifies an operation to be executed on the elements of an object structure. The visitor lets us specify a new operation without changing the classes of the elements on which it operates. In other words, the visitor pattern can access the different objects of a type and perform different operations on them without modifying the structure of the type. According to the properties of the visitor pattern, we find that it is a suitable method to use to develop the simplifier to deal with the Abstract Syntax Tree. Generally, we do not add a new node in the Abstract Syntax Tree after it has been generated; apart from that, the different node may be needed to deal with different operations. Hence, it is proper to use the visitor pattern to deal with this situation.

JJTree provides good support for the visitor design pattern. If the VISITOR option is set to true, then JJTree will insert an jjtAccept() method and childrenaccept() method into all of the node classes it generates. At the same time, it also produces a visitor interface, Visitor.JAVA, that can be implemented and passed to the nodes to accept. Our simplifier is an implementation of the visitor interface, Visitor.JAVA. Simple Nodes generated by JJTree are processed differently based on the different IDs of the nodes. In the visit method of the simplifier, first, the value of ID is obtained and then, a switch statement is used to address different types of nodes and perform operations on different nodes. Finally, after accessing the node, the childrenaccept() method is invoked to make the Visitor object traverse the entire Abstract Syntax Tree.

Here is the process of the try statement which it is not related to the security .The following syntax is for the try statement in Fig. 20.

```
void TryStatement( ):
{ }{
"try" Block( )
("catch""("FormalParameter( )")")*
["finally" Block( )]
}
```

Fig. 20: Syntax of Try statement in SubJAVA

The section behind the first Block() should be deleted. Thus, the case statement is presented in Fig. 21.

According to the production of the try statement, the syntax tree of the statement try {}catch (Exception e){} is expressed in Fig. 22.

According to the syntax tree of the try statement, the visitor traverses the nodes by the sequence in Fig. 23.

First, the main method in the parser will invoke the jjtAccept (Visitor, Object) method of the try statement node. The parameter visitor is the object deletevisitor. Then, the visit method in the object deletevisitor is invoked. At the same time, the try statement node is transferred to the visit method. We apply a post-order traversal. Hence, the visit method first invokes childrenAccept (Visitor, Object) in the try statement node to traverse the subnodes with the parameter deletevisitor. The childrenAccep method invokes the jjtAccept (Visitor, Object) method in the subnodes to perform

```
case EG2TreeConstants.JJTTRYSTATEMENT:
        node.childrenAccept(this, data);
        deletechildren(node, 2);
        break;
```

Fig. 21: Case statement

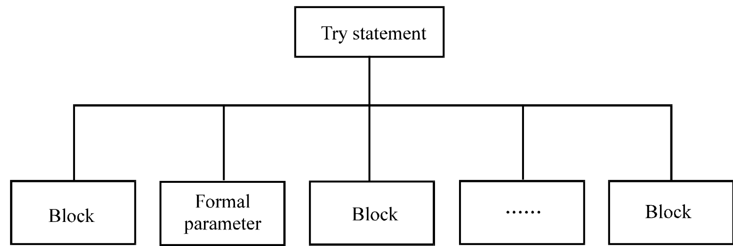

Fig. 22: Syntax tree of try statement

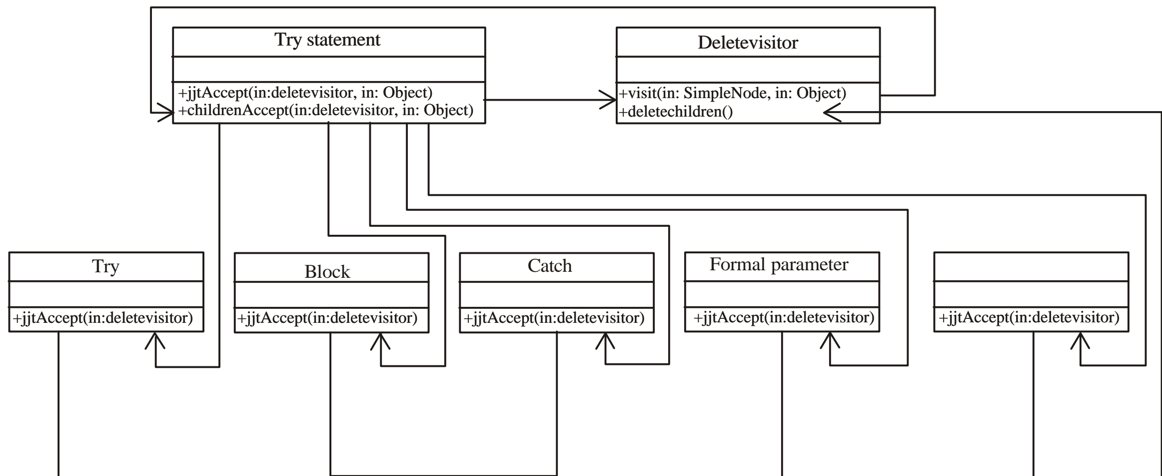

Fig. 23: Procedure of traveling by visitor

the traversal of the subtree; its root node is the try statement according to the sequence of the subnodes. Then, the deletechildren() method is invoked to delete all of the nodes behind the second children node in the try statement node. Apart from that, the children field in the try node is revalued to avoid a logical error.

Transformer: The transformer is also a visitor, one that a ccepts the simplified abstract syntax tree for SubJAVA and produces an abstract syntax tree for Blanchet calculus. In other words, the transformer is a mapping function from language elements in SubJAVA to language elements in Blanchet calculus according to the definition in Fig. 14. In the next section, we demonstrate how to use a visitor to perform the mapping function ExtractorVariableDeclaration (type x) = x[: type].

Figure 24 presents a local variable declaration abstract syntax tree in SubJAVA. First, the type of the variable is defined, then the variable name is specified and finally the variable is initialized.

A local variable declaration abstract syntax tree in Blanchet calculus can be found in Fig. 25. First, the variable name is defined, then the type of the variable is specified and finally the variable is initialized.

The transformer performs on a local variable declaration node (Fig. 26). The visitor first accesses the id field of the node and then evaluates the match between the value of id in the node and JJTLOCALVARIABLEDECLARATION in Eg2TreeConstants. If the match is successful, then the node is a local variable declaration node and some operations are performed on it. An operation is performed to exchange the position of VariableDeclaratorId node and type node (Fig. 26). The exchange is a method developed by us.

When the transformer has visited all of the nodes in the abstract syntax tree for SubJAVA, the correspondent abstract syntax tree for Blanchet calculus will be produced. Figure 27 presents the code for carrying out the exchange operations.

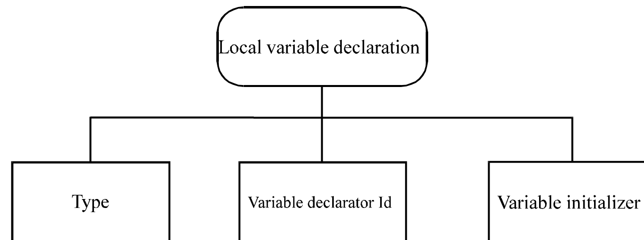

Fig. 24: Local variable declaration Abstract Syntax Tree in SubJAVA

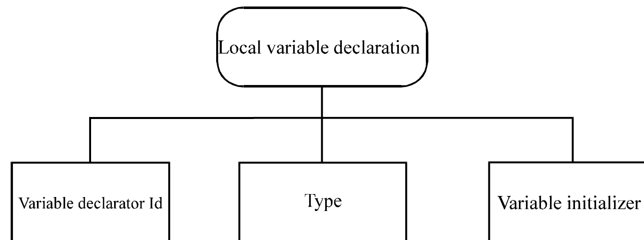

Fig. 25: Local variable declaration Abstract Syntax Tree in Blanchet calculus

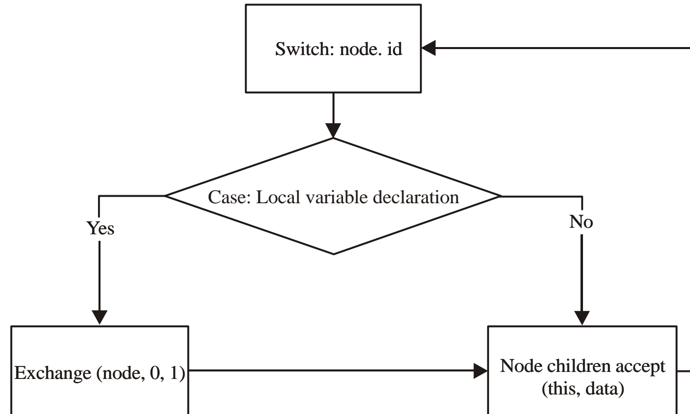

Fig. 26: Transformer performs on local variable declaration node

```
case EG2TreeConstants.JJTLOCALVARIABLEDECLARATION:
            exchange(node,0 1)
node.childrenAccept(this, data);
            break;
```

Fig. 27: Code of dealing with exchange operations

Code generator: If we have obtained the abstract syntax tree of Blanchet calculus, then the next task is to develop a code generator to generate code for the security protocol implementation in Blanchet calculus. Although the abstract syntax tree is made up of type name, variable name, method name, parameter list and so on, we can not directly access the leaf node to generate the security protocol implementation in Blanchet calculus because some information is hidden in the structure of the abstract syntax tree: for example{ } in the block and "=" in the assignment statement. Hence, we need to develop a code generator visitor to traverse the abstract syntax tree of Blanchet calculus in order to deal with some problems, such as the addition of key words, operator and brackets in the appropriate positions.

First, the object StringBuffer is created and then the jjtAccept() method is invoked with the input of the object StringBuffer and the code generator visitor. The object StringBuffer is used to store the security protocol implementation in Blanchet calculus.

Figure 28 presents the structure of the code generator visitor. The visitor method first retrieves the value of ID and then processes the different node with different operations. We use a switch statement to match the value of ID and the task of processing the node is performed in the case statement. We can find the process of the Block node and outstatement node. The block statement is the node of the statement block and the sign "{}" needs to be added. Hence, we should first add the sign "{" into the object StringBuffer and then travel the Block node before finally adding the sign "}" into the object StringBuffer. The outstatement node is processed in a similar way.

After the code generator visitor traverses the entire abstract syntax tree, the security protocol implementation in Blanchet calculus is stored in the object StringBuffer. Finally, the contents of the object StringBuffer are written to a file; thus, we can obtain the security protocol implementation in Blanchet calculus. Figure 29 presents part of the code of the code generator visitor.

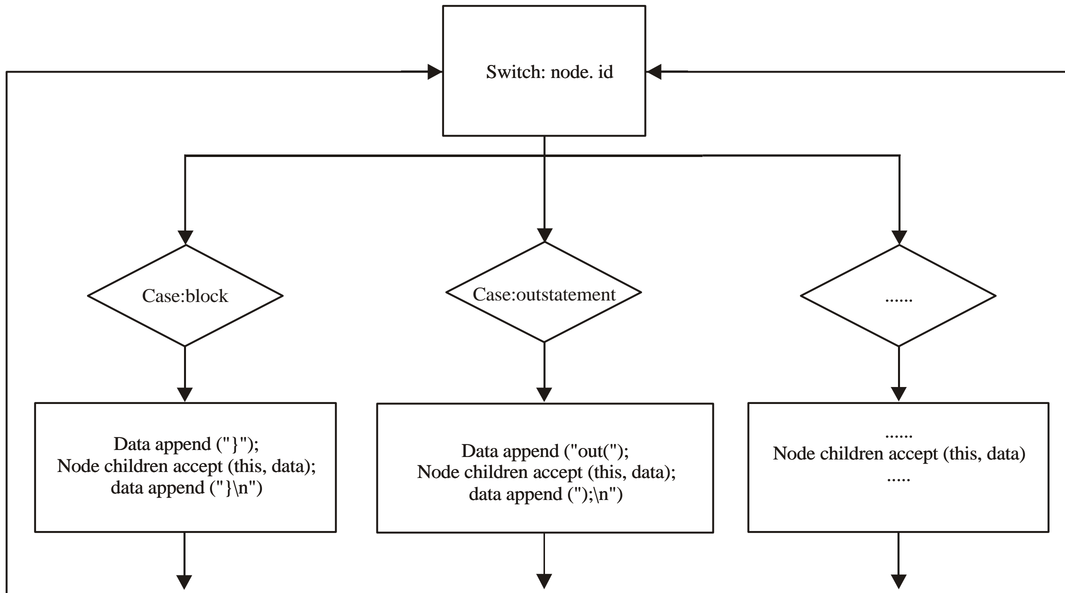

Fig. 28: Structure of code generator visitor

```
case EG2TreeConstants.JJTIDENTIFIER:
        node.childrenAccept(this, data);
        ((StringBuffer) data).append(node.jjtGetValue( ).toString( ));
        break;
case EG2TreeConstants.JJTBLOCK:
        ((StringBuffer) data).append({);
node.childrenAccept(this, data);
        ((StringBuffer) data).append(});
        break;
case EG2TreeConstants.JJTOUTSTATEMENT:
        ((StringBuffer) data).append(out();
node.childrenAccept(this, data);
        ((StringBuffer) data).append(););
        break;
```

Fig. 29: Part of code of code generator visitor

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

The automatic verifier SubJAVA2CV (Fig. 30) is made up of four functions: Loading the file of SubJAVA, generating the abstract syntax tree for SubJAVA, generating the abstract syntax tree for Blanchet calculus and outputting the file of Blanchet calculus.

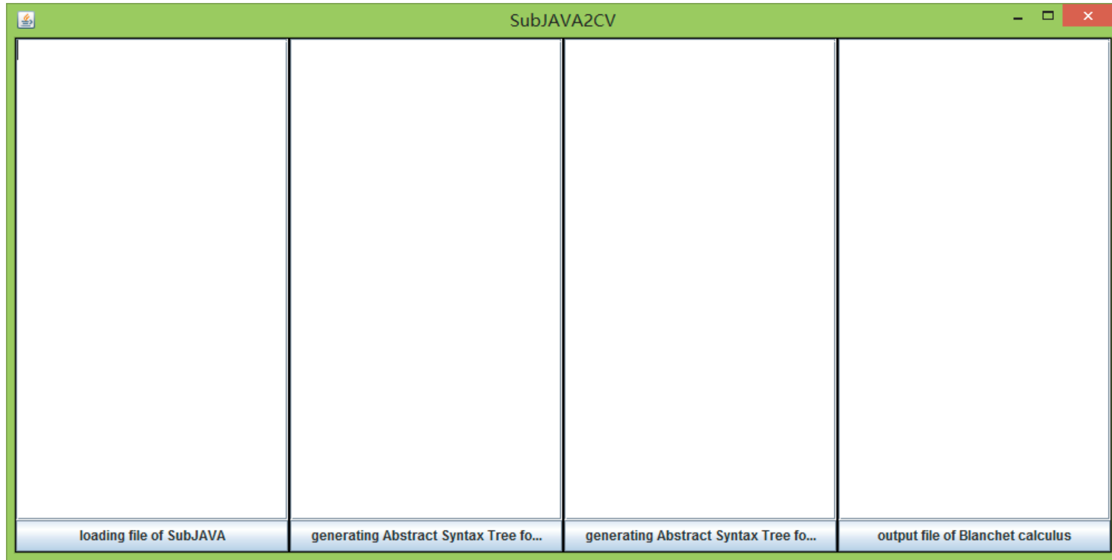

Fig. 30: Automatic verifier SubJAVA2CV

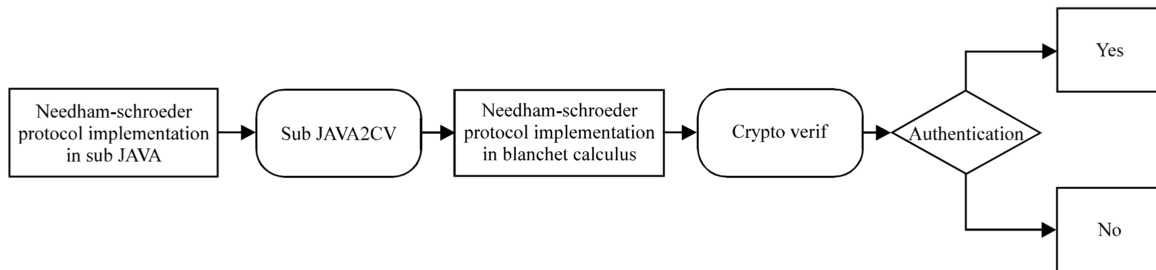

Fig. 31: Analysis the authentication of Needham-Schroeder protocol implementation in SubJAVA

CASE: NEEDHAM-SCHROEDER PROTOCOL IMPLEMENTATION IN SUBJAVA

In this section, we use the automatic verifier SubJAVA2CV and CryptoVerif to analyze the authentication of a Needham-Schroeder protocol implementation in SubJAVA, as shown in Fig. 31. First, we develop the Needham-Schroeder protocol implementation in SubJAVA and then we use the automatic verifier SubJAVA2CV to generate the Needham-Schroeder protocol implementation in Blanchet calculus; finally, CryptoVerif is used to perform the analysis of the authentication of the Needham-Schroeder protocol implementation in Blanchet calculus.

Review of the Needham-Schroeder protocol: The Needham-Schroeder protocol was designed by Roger Needham and Michael Schroeder and is used to implement mutual authentication with a public cipher.

The simplified Needham-Schroeder protocol in Fig. 32 is made up of two roles: Client and server. The client generates his private key, PR_{client} and public key, PU_{client}. The server also generates his private key, PR_{server} and public key, PU_{server}. The Needham-Schroeder protocol consists of three messages: $\{N_{client}, clientID\}PU_{server}$, $\{N_{server}, N_{client}\}PU_{server}$, $\{N_{server}\}PU_{server}$ where, N_{client} is the random number generated by the client, N_{server} is a random number generated by the server, clientID is the identifier of the client, PU_{client} is the public key of the client and Pu_{server} is the public key of the server. For:

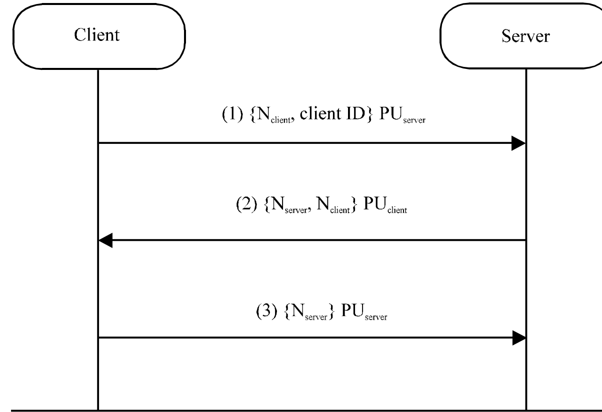

Fig. 32: Simplified Needham-Schroeder protocol

{N_{client}, clientID}PU_{server} the client generates a random number, N_{client}, uses the public key of the server, PU_{server}, to encrypt {N_{client}, clientID}, yielding the ciphertext {N_{client}, clientID} Pu_{server} and finally sends {N_{client}, clientID} PU_{server} to the server. For N_{server}, N_{client}}PU_{client} the server receives the message {N_{client}, clientID} PU_{server}, decrypts it with the private key PR_{server} and obtains N_{client} and clientID. After that, the server generates a random number, N_{server}, encrypts {N_{server}, N_{client}} with the public key of the client, PU_{client} and obtains the message {N_{server}, N_{client}}. Finally, the message {N_{server}, N_{client}} PU_{client} is forwarded to the client. For {N_{server}}, P_{server}, when the client receives the message {N_{server}, N_{client}} PU_{client}, he uses his private key, PR_{client}, to decrypt it and obtains N_{server}; he then uses the server's public key, PU_{server}, to encrypt N_{server} and obtains the message {N_{server}} P_{server} which is then forwarded to the server.

Lowe (1995) has analyzed this and showed that it does not have authentication. Apart from that, he notes that there is vulnerability to a man-in-the-middle attack.

Needham-Schroeder protocol implementation in SubJAVA: The Needham-Schroeder protocol implementation in SubJAVA consists of two sections: Client and server. The client of the Needham-Schroeder protocol implementation in SubJAVA is presented in Fig. 33. First, a random number int r1 is generated using the random number generator random. Then, the random number int r1 is linked with the character string client to construct the message onetext. Finally, the message onetext is encrypted with the server's public key through a RSA cipher and it is sent through the channel out. After that, the client receives the message through the channel in. Then, it uses its private key to decrypt the message and obtains {N_{server}, N_{client}}. Finally, the client obtains the message {N_{server}} PU_{server} with the public key of the server is forwarded to server PU_{server} and sends it to the server through the channel out.

The server of the Needham-Schroeder protocol implementation in SubJAVA is presented in Fig. 34. The server first receives the message from the channel instring sent by the client. Then, the message is decrypted with his private key and the character string N_{client} and the client identifier client is recovered. The random number int nserver is generated using the random number generator as N_{server}. After that, the combination {N_{server}, N_{client}} of N_{server} and N_{client} is encrypted with the public key of the client and the ciphertext is forwarded to the client. Finally, the message is received through the channel instring and is decrypted with the private key of the server. If the plaintext is the same as nserver, then the authentication is correct.

```
Random random = new Random();
Integer r1 = random.nextInt();
String numclient = r1.toString();
String onetext = numclient+",Client";
System.out.println("the frist cheartext is:"+onetext);
String onechipher = Rsa.encryptByPublicKey(onetext, servaerpkey);
System.out.println("the frist chipher is:"+onechipher);
out.println(onechipher);
out.flush();

String twotext = in.readLine();
System.out.println("the second chipher is:"+twotext+"\n");
String secondcleartext = Rsa.decryptByPrivateKey(twotext, privatekey);
System.out.println("the second cheartext is:"+secondcleartext+"\n");
String[] numserver = secondcleartext.split(",");
String numclient2 = numserver[1];

if (numclient.equals(numclient2))
  {String threetext = numserver[0];
   System.out.println("the third cheartext is:"+threetext+"\n");
   String threechipher = Rsa.encryptByPublicKey(threetext, servaerpkey);
   System.out.println("the third chipher is:"+threechipher+"\n");
   out.println(threechipher);
   out.flush();
  }
```

Fig. 33: Key part of client of Needham-Schroeder protocol implementation in SubJAVA

```
String onechipher = instring.readLine();
System.out.println("the first chipher is:"+onechipher+"\n");
String onetext = Rsa.decryptByPrivateKey(onechipher, privatekey);
System.out.println("the frist cleartext is:"+onetext+"\n");

String[] textone = onetext.split(",");
Random random = new Random();
Integer nserver = random.nextInt();
String numserver = nserver.toString();
String nclient = textone[0];
String secondtext = numserver+","+nclient;
System.out.println("the second cleartext is:"+secondtext+"\n");
String secondchipher = Rsa.encryptByPublicKey(secondtext, clientpk.getkey());
System.out.println("the second chipher is:"+secondchipher+"\n");
out.println(secondchipher);
out.flush();

String threechipher = instring.readLine();
System.out.println("the third chipher is:"+threechipher+"\n");
String threetext = Rsa.decryptByPrivateKey(threechipher, privatekey);
if(threetext.equals(numserver))
System.out.println("the third cleartext is:"+threetext+"\n");
```

Fig. 34: Key part of server of Needham-Schroeder protocol implementation in SubJAVA

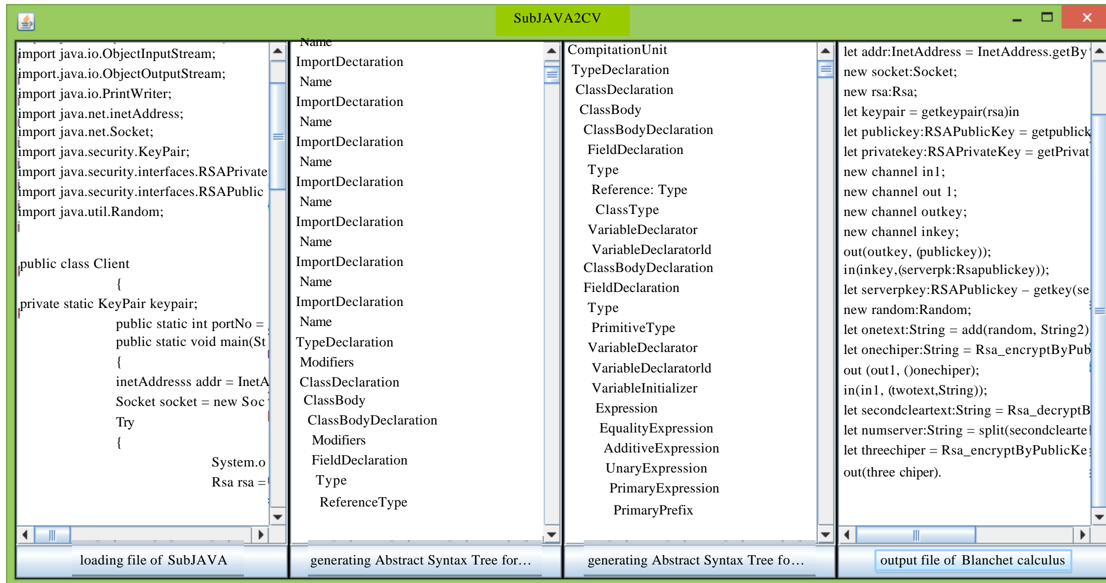

Fig. 35: Generating Needham-Schroeder protocol implementation in Blanchet calculus

Formalize the Needham-Schroeder protocol implementation in Blanchet calculus: According to the Needham-Schroeder protocol implementation in SubJAVA, we use the automatic verifier SubJAVA2CV to generate the Needham-Schroeder protocol implementation in Blanchet calculus (Fig. 35).

Figure 36-38 present the Needham-Schroeder protocol implementation in Blanchet calculus. The type, function and cryptographic primitive declaration are presented in Fig. 36-38 present the client process and server process.

Authentication of the Needham-Schroeder 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 the Needham-Schroeder protocol implementation in Blanchet calculus.

The mechanized prover CryptoVerif can directly prove security properties of cryptographic protocols in the computational model in which the cryptographic primitives are functions on bit-strings and the attacker is a polynomial-time Turing machine. It also can prove secrecy properties and events that can be executed only with negligible probability. CryptoVerif uses correspondences to model authentication.

Correspondences assert that, if some event is executed, then other events must also have been executed at least once, with matching parameters, at least with overwhelming probability. CryptoVerif can deal with both injective and non-injective properties. A non-injective correspondence is a property of the form “if some events have been executed, then some other events have been executed at least once”. Injective correspondences are properties of the form “if some event has been executed n times, then some other events have been executed at least n times”.

```
param N.
type Random [large, fixed, bounded].
type ServerSocket[fixed].
type Socket[fixed].
type Rsa[fixed].
type int[fixed].
type Integer[fixed].
type InetAddress[fixed].
type KeyPair[fixed].
type Rsapublickey[bounded].
type RSAPublicKey[bounded].
type RSAPrivateKey[bounded].
type String[large, fixed].const String1: String.
const String2: String.
fun getkey(Rsapublickey): RSAPublicKey [compos].
fun getkeypair(Rsa): KeyPair [compos].
fun getpublickey(Rsa): RSAPublicKey [compos].
fun getPrivate(KeyPair): RSAPrivateKey [compos].
fun Rsa_decryptByPrivateKey(String, RSAPrivateKey): String [compos].
fun split(String): String [compos].
fun nextInt(Random): Integer [compos].
fun add(String, String): String [compos].
fun Rsa_encryptByPublicKey(String, RSAPublicKey): String [compos].
fun InetAddress_getByName(String): InetAddress [compos].
proba Penc.
proba Penccoll.
expand IND_CCA2_public_key_enc(keyseed, pkey, skey, cleartext,ciphertext, seed,
                               skgen, pkgen, enc, dec, injbot, Z, Penc, Penccoll)
```

Fig. 36: Type, function and cryptographic primitive declaration

```
let processclient=
  new keypairc: KeyPair;
  new portNoc: int;
  let addrc: InetAddress = InetAddress_getByName(String1) in
  new socketc: Socket;
  new rsac: Rsa;
  let keypairc: KeyPair = getkeypair(rsac) in
  let publickeyc: RSAPublicKey = getpublickey(rsac) in
  let privatekeyc: RSAPrivateKey = getPrivate(keypairc) in
  new channel inc;
  new channel outc;
  new channel outkeyc;
  new channel inkeyc;
  out(outkeyc,(publickeyc));
  in(inkeyc,(serverpk:Rsapublickey));
  let servaerpkeyc: RSAPublicKey = getkey(serverpk) in
```

Fig. 37: Continue

```
 new randomc: Random;
 let r1c: Integer = nextInt(randomc) in
 let numclientc: String = toString(r1c) in
 let onetextc: String = add(numclientc, String2) in
 let onechipherc: String = Rsa_encryptByPublicKey(onetextc, servaerpkeyc) in
 out (outc,(onechipherc));
 in(inc,(twotextc:String));
 let secondcleartextc: String = Rsa_decryptByPrivateKey(twotextc, privatekeyc) in
 let numserverc: String = split(secondcleartextc) in
 let numclient2: String = numserver1(numserverc) in
 if numclient2 = numclientc then;
 let threetextc: String = numserver0(numserverc) in
 let threechipherc: String = Rsa_encryptByPublicKey(threetextc, servaerpkeyc) in;
 out(threechipherc)
```

Fig. 37: Client process

```
let processserver=
  new portNos: int;
  new sockets: ServerSocket;
  new channel ins;
  new channel outs;
  new channel outkeys;
  new channel inkeys;
  in(inkeys,(clientpk: Rsapublickey));
  let clientpkey: RSAPublicKey = getkey(clientpk) in
  new rsas: Rsa;
  let keypairs: KeyPair = getkeypair(rsas) in
  let publickeys: RSAPublicKey = getpublickey(rsas) in
  let privatekeys: RSAPrivateKey = getPrivate(keypairs) in
  out(outkeys, (publickeys));
  in(ins, (onechiphers: String));
  let onecleartexts: String = Rsa_decryptByPrivateKey(onechiphers, privatekeys) in
let textones: String = split(onetexts) in
  new randoms:Random;
  let r1s: Integer = nextInt(randoms) in
  let numservers: String = toString(r1s) in
  let nclients: String = textone0(textones) in;
  let secondtexts: String = add(numservers,nclients) in
  let secondchiphers: String = Rsa_encryptByPublicKey(secondtexts, clientpkey) in
  out (outs,(secondchiphers));
  in(ins,(threechiphers:String));
  let threetexts: String = Rsa_decryptByPrivateKey(threechiphers, privatekeys) in
  if threetexts = numservers then.
process
  ((client) | (server))
```

Fig. 38: Server process

```
query x:Random;
event serverb(x)==>clientb(x).
query x:Random;
event clienta(x)==>servera(x).
```

Fig. 39: Non-injective correspondences

```
param N.
type Random [large,fixed,bounded].
type ServerSocket[fixed].
type Socket[fixed].
type Rsa[fixed].
type int[fixed].
type Integer[fixed].
type InetAddress[fixed].
type KeyPair[fixed].
type Rsapublickey[bounded].
type RSAPublicKey[bounded].
type RSAPrivateKey[bounded].
type String[large,fixed].const String1:String.
const String2:String.
fun getkey(Rsapublickey):RSAPublicKey [compos].
fun getkeypair(Rsa):KeyPair [compos].
fun getpublickey(Rsa):RSAPublicKey [compos].
fun getPrivate(KeyPair):RSAPrivateKey [compos].
fun Rsa_decryptByPrivateKey(String, RSAPrivateKey):String [compos].
fun split(String):String [compos].
fun nextInt(Random):Integer [compos].
fun add(String, String):String [compos].
fun Rsa_encryptByPublicKey(String, RSAPublicKey):String [compos].
fun InetAddress_getByName(String):InetAddress [compos].
proba Penc.
proba Penccoll.
expand IND_CCA2_public_key_enc(keyseed, pkey, skey, cleartext,
 ciphertext, seed, skgen, pkgen, enc, dec, injbot, Z, Penc, Penccoll)
```

Fig. 40: Type, function and cryptographic primitive declaration

Here, we use non-injective correspondences (Fig. 39) to model the authentication from server to client and from client to server. The event server b(x) ==>client b(x) is used to authenticate the client by the server. The event event client a(x) ==>client a(x) is used to authenticate the server by the client.

Figure 40-43 present the code of the verification of the authentication in CryptoVerif which has been added to the events and non-injective correspondences by hand. The analysis was performed by CryptoVerif and succeeded. The results are shown in Fig. 44 and the Needham-Schroeder protocol in SubJAVA is proved not to guarantee authentication. The result is consistent with the result of the Needham-Schroeder protocol abstract specification.

We know that there is a man-in-the-middle attack in the Needham-Schroeder protocol, shown in Fig. 45. First, the adversary pretends to be the server to authenticate the identifier of the client and send his public key, $PU_{adversary}$, to the client. Thus, the client constructs the message: (1) $\{N_{client}$, clientID$\}$ $PU_{adversary}$ which is the ciphertext of $\{N_{client}$, clientID$\}$ and sends it to the adversary. The adversary receives message (1) and decrypts it using its private key, $PR_{adversary}$, obtaining $\{N_{client}$, clientID$\}$. After that, the adversary constructs the message: (2) $\{N_{client}$, clientID$\}$ PU_{server} using the public key of the server, PU_{server} and sends it to the server. The server receives message (2) and then it generates message: (3) $\{N_{server}, N_{client}\}$ PU_{client} using the public key of the client, PU_{client}. After that, the client receives message (3) and uses his private key, PR_{client}, to recover $\{N_{server}, N_{client}\}$. The client generates message: (4) $\{N_{server}\}PU_{adversary}$ using the public key of the adversary, $PU_{adversary}$ and sends

```
event servera(String).
event clienta(String).
event serverb(String).
event clientb(String).
query x:String;
event serverb(x)==>clientb(x).
query x:Sting;
event clienta(x)==>servera(x)
```

Fig. 41: Authentication specified by events

```
let processclient=
  new keypairc: KeyPair;
  new portNoc:int;
  let addrc: InetAddress = InetAddress_getByName(String1) in
  new socketc: Socket;
  new rsac: Rsa;
  let keypairc: KeyPair = getkeypair(rsac) in
let publickeyc: RSAPublicKey = getpublickey(rsac) in
  let privatekeyc: RSAPrivateKey = getPrivate(keypairc) in
  new channel inc;
  new channel outc;
  new channel outkeyc;
  new channel inkeyc;
  out(outkeyc, (publickeyc));
  in(inkeyc, (serverpk: Rsapublickey));
  let servaerpkeyc: RSAPublicKey = getkey(serverpk) in
  new randomc: Random;
  let r1c: Integer = nextInt(randomc) in
  let numclientc: String = toString(r1c) in
  let onetextc: String = add(numclientc, String2) in
  let onechipherc: String = Rsa_encryptByPublicKey(onetextc, servaerpkeyc) in
  out(outc, (onechipherc));
```

Fig. 42: Client process

```
let processserver=
  new portNos: int;
  new sockets: ServerSocket;
  new channel ins;
  new channel outs;
  new channel outkeys;
  new channel inkeys;
  in(inkeys, (clientpk: Rsapublickey));
  let clientpkey: RSAPublicKey = getkey(clientpk) in
  new rsas: Rsa;
  let keypairs: KeyPair = getkeypair(rsas) in
  let publickeys: RSAPublicKey = getpublickey(rsas) in
  let privatekeys: RSAPrivateKey = getPrivate(keypairs) in
  out(outkeys, (publickeys));
  in(ins, (onechiphers: String));
  let onecleartexts: String = Rsa_decryptByPrivateKey(onechiphers, privatekeys) in
  let textones: String = split(onetexts) in
  new randoms: Random;
  let r1s: Integer = nextInt(randoms) in
let numservers: String = toString(r1s) in
  let nclients: String = textone0(textones) in
  event servera(nclients);
  let secondtexts: String = add(numservers,nclients) in
  let secondchiphers: String = Rsa_encryptByPublicKey(secondtexts, clientpkey) in
  out (outs, (secondchiphers));
  in(ins,(threechiphers: String));
  let threetexts: String = Rsa_decryptByPrivateKey(threechiphers, privatekeys) in
  if threetexts = numservers then
  event serverb(threetexts).
process
  ((client) | (server) )
```

Fig. 43: Server process

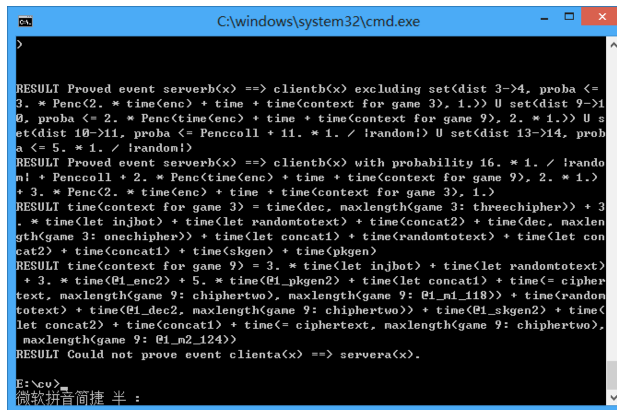

Fig. 44: Result of Needham-Schroeder protocol in CryptoVerif

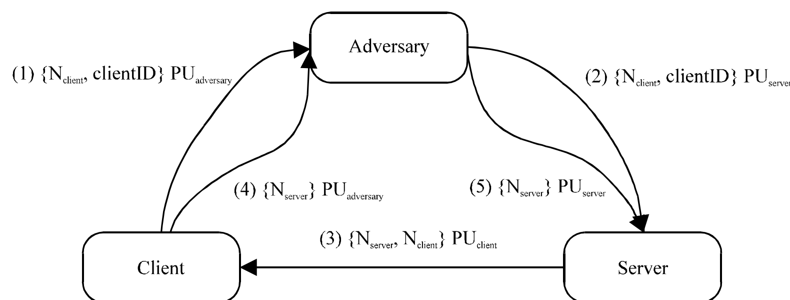

Fig. 45: Man-in-the-middle attack in Needham-Schroeder protocol

it to the adversary. The adversary receives message (4) and uses his private key, $PU_{adversary}$, to recover $\{N_{server}\}$. After that, the adversary uses the public key of the server, PU_{server}, to construct: (5) $\{N_{server}\}$ PU_{server} and sends it to the server. Then, the server use his private key, PR_{server}, to decrypt message(5) and to authenticate the identifier of the client and the authentication is successful.

Hence, we can conclude that the automatic verifier SubJAVA2CV correctly extracted the model represented by Blanchet calculus from the security protocol implementation in the SubJAVA language.

CONCLUSION

In the last several decades, many security protocols have been designed, implemented and deployed in various information systems. Hence, people have paid special attention to their security. Now, a hot issue has emerged from the security field which is the analysis of security properties of security protocol implementations. Thus, in this study, we mechanized the verification of cryptographic security in security protocol implementations in the computational model. First, we presented a model of analysis and verification of cryptographic security in security protocol implementations. Then, we developed an automatic verifier, SubJAVA2CV, that can translate security protocols written in SubJAVA to security protocol abstract specifications which are inputs of CryptoVerif. Finally, we used the automatic verifier SubJAVA2CV and CryptoVerif to analyze the authentication of the Needham-Schroeder protocol implementation in SubJAVA. In the near future, to begin with, we will use SubJAVA2CV and CryptoVerif to analyze more security protocol implementations written in JAVA language expand the ability of SubJAVA ; Apart from that we will use the proof assistant Coq to prove the correctness of translation from SubJAVA to Blanchet calculus and we also focus on mechanized generation of security protocol implementations written in JAVA language from security protocol abstract specifications proved in the computational model.

ACKNOWLEDGMENT

This study was supported in part by the natural science foundation of Hubei Province under the grants No. 2014CFB249, titled “Automatic Verification of Cryptographic Security in Security Protocol Code and Development of Software Tools”, conducted in Wuhan, China.

REFERENCES

Aizatulin, M., A.D. Gordon and J. Jurjens, 2011. Extracting and verifying cryptographic models from C protocol code by symbolic execution. Proceedings of the 18th ACM Conference on Computer and Communications Security, October 17-21, 2011, Chicago, IL., USA., pp: 331-340.

Aizatulin, M., A.D. Gordon and J. Jurjens, 2012. Computational verification of c protocol implementations by symbolic execution. Proceedings of the 19th ACM Conference on Computer and Communications Security, October 16-18, 2012, Raleigh, NC., USA., pp: 712-723.

Backes, M., C. Hritcu and M. Maffei, 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. J. Comput. Secur., 22: 301-353.

Backes, M., M. Maffei and D. Unruh, 2010. Computationally sound verification of source code. Proceedings of the 17th ACM Conference on Computer and Communications Security, October 4-8, 2010, Chicago, IL., USA., pp: 387-398.

Bengtson, J., K. Bhargavan, C. Fournet, A.D. Gordon and S. Maffeis, 2011. Refinement types for secure implementations. ACM Trans. Program. Lang. Syst., Vol. 33. 10.1145/1890028.1890031

Bhargavan, K., C. Fournet and A.D. Gordon, 2010. Modular verification of security protocol code by typing. Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 17-23, 2010, Madrid, Spain, pp: 445-456.

Bhargavan, K., C. Fournet, A.D. Gordon and S. Tse, 2008. Verified interoperable implementations of security protocols. ACM Trans. Program. Lang. Syst., Vol. 31. 10.1145/1452044.1452049

Bhargavan, K., R. Corin, C. Fournet and E. Zalinescu, 2009. Automated computational verification for cryptographic protocol implementations. MSR-INRIA Technical Report.

Blanchet, B., 2008. A computationally sound mechanized prover for security protocols. IEEE Trans. Dependable Secure Comput., 5: 193-207.

Chaki, S. and A. Datta, 2009. ASPIER: An automated framework for verifying security protocol implementations. Proceedings of the 22nd IEEE Computer Security Foundations Symposium, July 8-10, 2009, Port Jefferson, New York, pp: 172-185.

Dupressoir, F., A.D. Gordon, J. Jurjens and D.A. Naumann, 2011. Guiding a general-purpose c verifier to prove cryptographic protocols. Proceedings of the IEEE 24th Computer Security Foundations Symposium, June 27-29, 2011, Cernay-la-Ville, France, pp: 3-17.

Goubault-Larrecq, J. and F. Parrennes, 2005. Cryptographic protocol analysis on real c code. Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, January 17-19, 2005, Paris, France, pp: 363-379.

Jurjens, J., 2009. Automated security verification for crypto protocol implementations: Verifying the jessie project. Electron. Notes Theor. Comput. Sci., 250: 123-136.

Lowe, G., 1995. An attack on the Needham-Schroeder public-key authentication protocol. Inform. Process. Lett., 56: 131-133.

O'Shea, N., 2008. Using Elyjah to analyse JAVA implementations of cryptographic protocols. Proceedings of the Conference on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, June 21-22, 2008, Pittsburgh, PA., USA., pp: 221-226.

Swamy, N., J. Chen, C. Fournet, P.Y. Strub, K. Bharagavan and J. Yang, 2011. Secure distributed programming with value-dependent types. Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, September 19-21, 2011, Tokyo, Japan, pp: 266-278.