Protocol Engineering Applied to Formal Analysis of Security Systems

Abstract. Every communication system requiring security properties is certainly critical. In order to study the security of communication systems, we have developed a methodology for the application of the formal analysis techniques of communication protocols to the analysis of cryptographic ones. We have extended the design and analysis phases with security properties. Our methodology uses a specification technique based on the HMSC/MSC requirement languages, and translates it into a generic schema for the SDL specification language that it is analyzed. Thus, the technique allows the specification of security protocols using a standard formal language and uses Object-Orientation for reusability purposes. The final goal is not only the formal specification of a security system, but to examine the possible attacks, and later use it in more complex systems.

1. INTRODUCTION

Nowadays, it is widely accepted that critical systems have to be analyzed formally in order to achieve well-known formal method benefits [Holz91]. These methods characterize the behavior of a system in a precise way and can verify its formal specification. Design and analysis of security systems must benefit from advantages of formal methods because of the evident criticality of such type of systems.

During last years, the cryptographic protocol analysis research area [Mead00] has seen an explosive growth, with numerous formalisms being developed. We can divide that research into three main categories: logic-based [BAN89], model checking [MCJ97][MMS97][Alur et al. 98] and theorem proving [DeMi99]. Only recently there has been a tendency to try to combine them.

We believe that the results obtained in the analysis phase of cryptographic protocol have a direct application on the design phase of a secure communication system. The reason is that there is no strong relation between security analysis tools and formal methods techniques of communication protocols.

Therefore, we have developed a new methodology to specify secure systems and to check that they are not vulnerable using well-known attacks. Our approach uses a requirement language to describe security protocols, as well as a generic formal language, together with its associate verification methods and tools. In our method a simple and powerful intruder process is explicitly added to the specification, so that the verification of the security properties guarantees the robustness of the protocol against attacks of such an intruder. The intruder controls the transmission medium and can perform the attacks [DoYa83]. His possible actions are killing, sniffing, intercepting, redirect, delaying, delivering, reordering, replaying, and faking.

Actually, secrecy and authentication [RySc01] are the security properties more widely analyzed. By analyzing secrecy we prevent the intruder from being able to derive the plaintext of messages passing between honest nodes. Our analysis consists of checking if the secret item can be deduced from the protocol messages and the intruder's database knowledge.

For an authentication protocol to be correct, it is required that a user Bob do not finish the protocol believing that has been running with a user Alice unless Alice also believes that she has been running the protocol with Bob. Our analysis consists of checking if there is a reachable state where Alice has finished correctly and Bob has not reached his final state.

The paper is structured in the following way. In section 2 we summarize SDL features and tools. In section 3 we explain how SDL can be used to specify security protocols and cryptographic operations. At the same time we show how security protocols can be modeled as safety properties and checked automatically by a model-based verification tool. Section 4 shows an example of how our methodology is applied to EKE protocol. Last section contains conclusions and future work.

2. SDL LANGUAGE

Specification and Description Language (SDL) [ITU99a] is a standard language for specifying and describing systems. It has been developed and standardized by ITU-T in the recommendation Z.100. An SDL specification/design (a system) consists of a number of interconnected modules (blocks). A block can recursively be divided into more blocks forming a hierarchy of blocks. The channels define the communication paths through which the blocks communicate with each other or with the environment. Each channel usually contains an unbounded FIFO queue that contains the signals that are transported on it. One or more communicating processes describe the behavior of the leaf blocks, and extended finite state machines describe the processes.

In addition, SDL supports object-oriented design [SEH97] by a type concept that allows specialization and inheritance to be used for most of the SDL concepts, like blocks, processes, data types, etc. The obvious advantage is the possibility to design compact systems and to reuse components, which in turn reduces the required effort to maintain a system. SDL has adopted the term "type", which corresponds to the term "class" used in many of the object-oriented notations and programming languages.

Telelogic's Tau SDL Suite provides an environment for developing SDL systems and implementations. The SDL Suite comprises a set of highly integrated tools that automate the transition from specification to real-time execution. With SDL's graphical language, SDL Suite describes, analyses, simulates, and supports generations of C/C++ applications. Thus SDL Suite simplifies testing and verifying the application by virtue of the formal semantics of the SDL language that makes tool support in early phases possible. The engineer composes diagrams, supported by a formal, well-defined graphical syntax that defines the program functionality, eliminating the need to manually write whole sections of code. We use SDL Validator tool for verification purpose. Indeed, we are going to take advantage of its exploration algorithms and its check mechanism.

2.1 SDL Validator

The SDL Validator [Hogr96] is based on state space exploration. State space exploration is based on automatic generation of reachable states of systems. Reachable state space means all possible states an application can find itself, and all possible ways it can be executed. A reachability graph is one way to conceptually view reachable state space, though one rarely computes it since it is too large for realistic applications.

In addition, a reachability graph represents the complete behavior of an application. The nodes of the graph represent SDL system states, and it contains all necessary information to describe the application state. For an SDL system, the complete description of the application states includes: the flow state of all concurrent processes, the value of all the variables, procedure call stacks, activated timers, signals in transmission and their parameter values, and so on.

The edges of the reachability graph represent SDL events that can take the SDL system from one system state to the next system state. The edges define the atomic events of the SDL system. These can be SDL statements like assignments, inputs and outputs, or complete SDL transitions depending on how the state space exploration is configured.

2.2 Exploration Algorithms

The state space can be explored using following algorithms: random walks, exhaustive exploration, bit-state exploration, and interactive simulation. The random walk algorithm randomly traverses the state space. Each time several possible transitions are available, the SDL Validator chooses one of them and executes it. The random walk algorithm is useful as an initial attempt for robustness testing of an application and when the state space is too large even for a partitioned bit state search.

The exhaustive exploration algorithm is a straightforward search through the reachability graph. Each system state encountered is stored in RAM. Whenever a new system state is generated, the algorithm compares it with previous generated states to check if the new one was reached already during the search. If it was, the search continues with the following to this state. If the new state is the same as a previously generated state in RAM, the current path is prune, and the search backs up to try more alternatives. The exhaustive exploration algorithm requires a lot of RAM, which limits its practical application.

The algorithm called bit state exploration can be used to efficiently validate reasonably large SDL systems. It uses a data structure called “hash table” to represent the system states that are generated during the exploration. When we want to analyze a particular situation, we use the interactive simulation. We guide state exploration to the goal scenario, and then we check for analysis results.

2.3 Checking Method

The Validator has several ways to check SDL specification. These are essentially scenario verification and observer process. In order to obtain scenario specifications we use the Message Sequence Chart (MSC) [ITU99b] language (Recommendation ITU-T Z.120). We can verify a MSC, checking if there is a possible execution path for the SDL system that satisfies the MSC, or checking MSC violation. Loading MSC and performing a state space exploration set up in a way suitable for verifying MSCs does this. The MSC verification algorithm is a bit state exploration that is adapted to suit the needs of MSC verification.

The more powerful way to check a SDL specification is the observer process mechanism. The purpose of an observer process is to make it possible to check more complex requirements on the SDL system than can be expressed using MSCs. The basic idea is to use SDL processes (called "observer processes") to describe the requirements that are to be tested and then include these processes in the SDL system. Typical application areas include feature interaction analysis and safety-critical systems.

By defining processes to be observer processes, the Validator will start to execute in a two-step fashion. First, the rest of the SDL system will execute one transition, and then all observer processes will execute one transition and check the new system state.

The assert mechanism enables the observer processes to generate reports during state space exploration. These reports will show up in the list of generated reports in the Report Viewer.

3. ANALYSIS MECHANISM

Our approach (figure 1) performs the design and analysis of a security protocol in the same way that we do it with a traditional communication protocol. Firstly, we define system requirements. These are specified in the Security Requirements Specification Language (SRSL), which has been designing to define the security features and analysis strategy. Following, we translate the system requirements into an SDL system in a semi-automatically manner, and we analyze it. Beside, It could be applied for code generation and testing.

The aim of the SRSL is to define a high level language in order to specify cryptographic protocols and secure systems. This language must be modular to achieve reusability, easy to learn, and have to use security concepts. The SRSL is divided into three parts; the first one is the specification of protocol elements, the second is message exchange flow, and the third is security analysis strategies.

The essential elements required to define a security protocol can be divided into several categories. These are explained as follows (keywords in cursive):

- Entities: Agent (*Initiator*/*Responder*), principal identification; *Server_Key*, provide a key; *Server_Time*, provide time token; *Notary*, register the transaction; Server Certification Authority (*SCA*), validate a certificate.
- Message: *Text*, clear text; *Random* Number, for freshness purposes; *Timestamp*, actual time; *Sequence*, count number.

- Keys: *Public_key*, for instance, PKCS#12 format; *Certificate*, public key signed by CA; *Private_key*, used to sign; *Shared_key*, secret key shared by more than one entity; *Session_key*, it is a secret key used to encrypt a transmission.

In addition, SRSL may operate with data type previously defined. Those oper-ations are: *Concatenate*, compose complex data (operator ’,’), it can be identified as a name; *Cipher/Decipher*, provide a cipher data and clear data respectively (for instance, RSAcipher/RSADecipher PKCS#1 format); *Hash*, a one-way algorithm re-sult; *Sign*, message hash encrypted with signer’s private key (for instance, RSAsign PKCS#7).

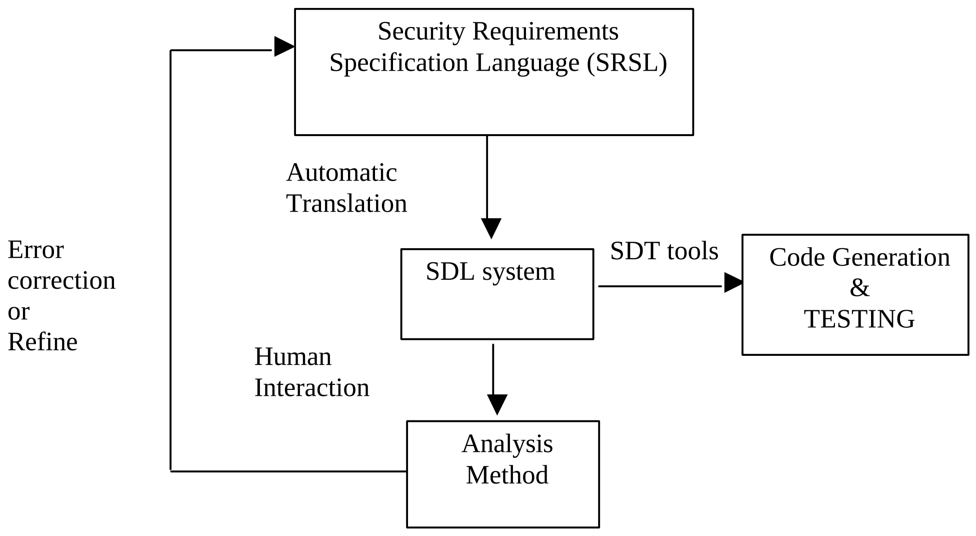

Security Requirements Specification Language (SRSL)
Automatic Translation
Error correction or Refine
SDL system
SDT tools
Code Generation & TESTING
Human Interaction
Analysis Method

Figure 1: Methodology structure

The message exchange is defined in the most widely requirements language utilized in telecommunications area, the Message Sequence Chart (MSC) and its extension High MSC (HMSC). We can specify elementary scenarios (MSC) and compose them into a complex protocol (HMSC).

We may describe security systems in multilayer way. The first layer is communication medium, besides it is used to apply attack strategy (intrude behavior). The other layers depend on security mechanisms employed in system development.

For instance, we consider a system that uses SSL security mechanism to achieve server authentication and secret communication. Although, we want to be able to send the credit card number and the product code data, and to achieve evidence in order to provide non-repudiation of origin.

The specification is described as follows:

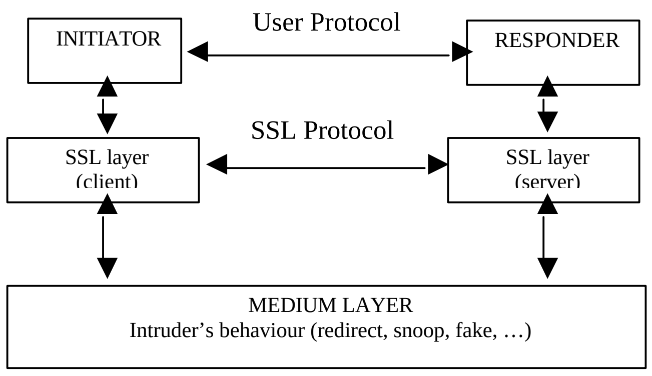


Figure 2: Modules Schema of security system specification

The SSL layer can be described in a standard security communications package. Therefore, part of Medium layer is generated automatically. Consequently, we only have to specify the Initiatior-Responder protocol. It is composed by simple scenarios described in MSC:

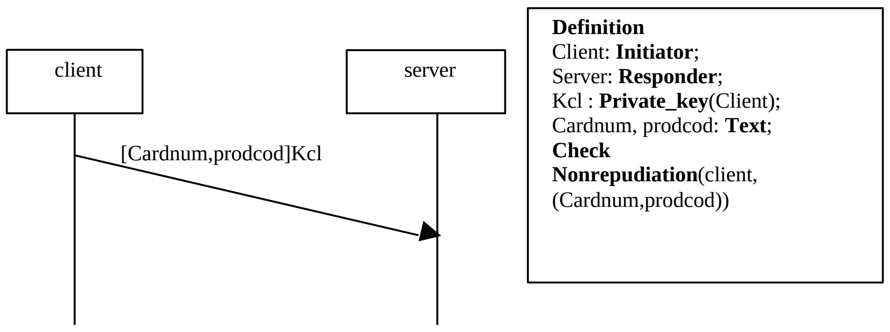


Figure 3: MSC description of User Protocol

The security analysis section describes at least the check property. This is called "check" and it has three possible conditions: *Authentication(A,B)*, for analyzing the authentication between agents A and B; *secret(X)*, to evaluate if *X* can be deduced (also called confidentiality); *Nonrepudiation(A,X)*, check if data *X* (the evidence) can be produced only by *A*. Therefore, it is possible to define a specific attack scenario using "*session_instance*" and "*intruder_behavior*" sections, in order to refine the exploration space.

An Automatic translator program is used to achieve the SDL system from SRSL. The SDL system is composed by a package where data types are defined, and other package where one type process for each protocol agent, and a group of type process observer and process medium are specified.

In order to analyze security properties, we evaluate the SDL system behavior when different kinds of attacks are applied for medium processes. Observer processes check if a determinate situation is searched and in that case result an inform report. This report corresponds on a failure scenario.

The analyzer creates the medium and observer processes for any kind of vulnerability that we want to examine. We have implemented generic process medium and observers, but they must be extended depending on system environment. A more detailed description of every part of SDL system is explained following.

3.1 Data Types Package

The messages, which are sent by protocol agents, are constructed by a concatenation of elemental data types and cryptographic operations. These data types can be divided in agent identification, number (random, time-stamping, etc…), symmetric key and asymmetric key pair. Thus, the operations used are cipher, decipher, sign and hash function.

To be more precise, we speak about certificate but not public key, although for analysis purposes it is accepted to use this simplification. Of course, for code generation we need to use the ASN.1 notation, because this obtains an unambiguous data management.

The package “analcryptlib” defines data type used by SDL specification. The SDL data types do not ort recursive definition, so we make use of enumerated and structured data types. The elemental data types defined are: (a) agent Identification, it is an enumerated sort with all possible agents name; (b) number, it is a fresh and/or random value; (c) Secret key, this represents symmetric key; (d) public key, this is composed by a key pair (private and public key); (e) encrypted message, that it is implemented with a structured sort composed by item message and item symmetric or asymmetric cipher; (f) signed message, it is defined as a structure sort with a message and the private key signer.

Freshness or temporary secrets are implemented appending an item that has the process instance values. The SDL sort PID allows doing it. Furthermore, we define a type set of knowledge for each data type. Intruder utilizes these types to store message knowledge.

3.2 Agents Package

The generic model identifies each protocol agent with a process type SDL. All process types are stored in a package in order to be used in other specifications. An agent specification is absolutely independent of the rest of the system, so they are generated in separated modules. Furthermore, this specification permits concurrent instance so that we can evaluate this behavior in the analysis stage.

The generic state transition of process agent is triggered when it receives a message and it is correct. Then, the next message is composed to be sent to the receiver agent or it will stop if it is the final state of this protocol. If the message is not correct, it will return to waiting message state.

The process in SDL is a finite state machine, so it finishes when execute a stop statement or provides a deadlock if no signal arrives. Our model has to explore all possibilities; thus, we need to develop a mechanism to ensure that all signals sent must be processed. Consequently, we have appended a state called “final” to notice the end of the protocol execution, and a general transition composed on a common “save” statement and a continuous signal, with less priority than the input statement, that check if there are some signal waiting for dealing out. By means of this structure we transform a finite state machine in an infinite one, only for analyzing purposes.

At this point, we have specified a security protocol in the same way that we might specify a traditional communication protocols, therefore we can examine the classical liveness properties. The specification must be well formed, but it is not the main aim of a secure system. In the next subsection, we are going to explain how to check the security properties.

3.3 Model Medium-Observer Processes

The intruder's behavior is divided into two aspects, exploration algorithm and check mechanism. The exploration algorithm is provided by a medium process and observer processes perform the check mechanisms.

We consider two types of medium process model. The first, it is characterized by an exploration mechanism that search all possibilities. It begins examining all combinations of different initial knowledge for each agent. Afterwards, it checks concurrent agents execution, at first we try combinations of two concurrent sessions, and so on. Our algorithm finishes when an out of memory is produced or it detects that the significant intruder knowledge is not incremented. In the general case the problem completeness [Lowe98, RuTu01] is undecidable, so it is impossible to be completely sure about problem solution.

The second is developed with an intruder specialized in finding a specific flaw. If we typify a kind of attack, we can evaluate the protocol trying to find a specific flaw. Perhaps this is not the best solution but it is very useful to protocol designer to be sure that for this kind of attack our protocol is not vulnerable. Furthermore, the majority of analysis tools that use model checking accept that this problem is undecidable, so we only get results about a definite vulnerability will not happen in the cases that we have examined.

The state transition of process medium is triggered when it receives any message. Then, it is stored in the intruder knowledge database, and an intruder decides which operations are going to be done, and go to the next state of routing. We have defined three different operations: eavesdrop, divert, and impersonate. Eavesdrop operation, meaning the intruder intercept the messages and rejects them. Divert operation, meaning the intruder intercepts the messages but they are not sent to the original receiver. Impersonate operation, meaning the intruder sends fake message to the original receiver.

The check mechanism is the observer process. This is a special SDL type of process that is evaluated in each transition of the protocol specification. It gets at all variable and state of whole process instances, so we can test it automatically.

The security properties are proved using condition rules. These rules check different situations where it is possible that there exists protocol vulnerability. These elements are agent's states, variable value, and sent signals.

Actually, we analyze secrecy and authentication properties. When we want to check secrecy properties, we examine if the intruder knowledge can be deduced a specific value that we consider secret. The authentication is examined checking that all the principals finish at the same time, when it is expected. Some authors call to this the correspondence or precedence flaw.

4. EXAMPLE OF PROTOCOL ANALYSIS

As an example to illustrate our proposal we explain the specification and analysis of a security system that make use of EKE (Encrypted Key Exchange) protocol [BeMe92], in order to gain access to a host, and cipher communications (figure 4).

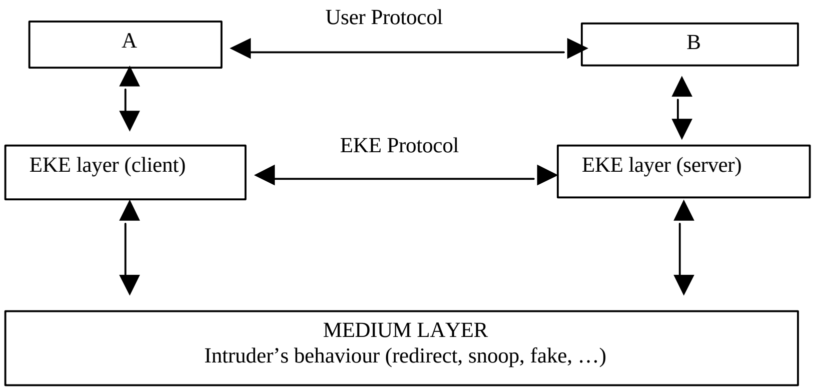
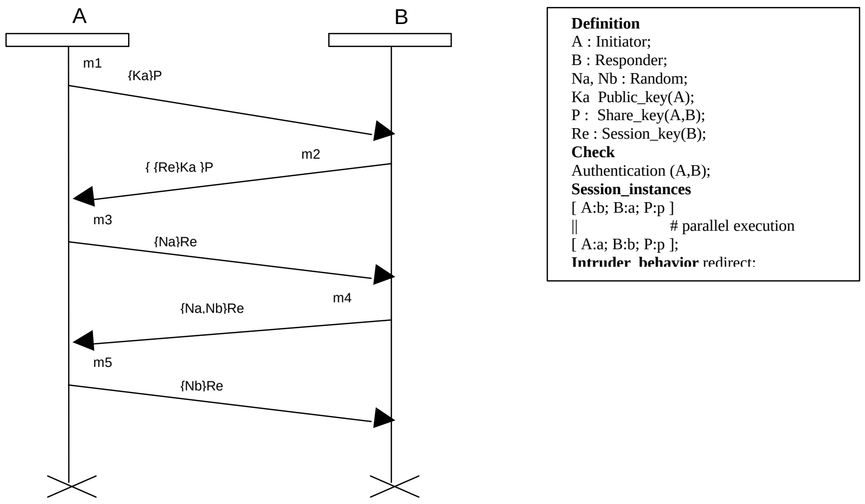


Figure 4: Multilayer structure of EKE environment

We are going to evaluate EKE protocol. This is a key exchange authentication protocol that resists dictionary attacks by giving passive attacker insufficient information to verify a guessed password. As stated, it performs key exchange as well, so both parties can encrypt their transmissions once authentication is established. In the most general form of EKE, the two communicating parties encrypt short-lived public keys with a symmetric cipher, using their shared secret password as a key. Since it was designed, EKE has been developed into a family of protocols, many of which are stronger than the original or add new desirable properties.

The basic EKE protocol is specified in SRSL (Figure 5). This represents two agents "A" and "B", A is the initiator and B is the responder. "P" is a share key (symmetric key shared by A and B). "Ka" is A's public key. "Re" is a session key (fresh symmetric key) generated by B. And "Na" and "Nb" are fresh and random number from A and B respectively.



```
Definition
A : Initiator;
B : Responder;
Na, Nb : Random;
Ka  Public_key(A);
P :  Share_key(A,B);
Re : Session_key(B);
Check
Authentication (A,B);
Session_instances
[ A:b; B:a; P:p ]
||                    # parallel execution
[ A:a; B:b; P:p ];
Intruder_behavior redirect;
```

Figure 5: EKE specification in SRSL

Firstly, we produce the SDL specification of EKE protocol, similar to an ordinary communications protocol. Then, we create the medium and observer processes, in order to analyze the correspondence flaw defined in CASRUL analysis tool [CASR00]. This flaw is produced when we execute two sessions concurrently. During first session, "A" and "B" are instanced to "a" and "b" principals identification, respectively, while during the second session, "A" and "B" are instanced to "b" and "a" principals identification, respectively. The observer process checks if agents of sessions 1 and 2 reach a final state and if theirs corresponding parties do not reach it.

4.1 Data Type Definition

The package "analcryptlib" includes all messages definition for analysis purpose. They are defined as SDL struct sort. Those belong to generic choice sort called "TMESSAGE". At the same time, it is defined a generic struct sort called "TENCMESS" where security operations are applied. Those operators are "enc" to encipher, "denc" to decipher, "sing" to do a digital signature, and "hash" to apply a hash function.

4.2 Agent Definition

Agents are defined as a process type included in SDL package called "agents". We specify agent initiator process type ("agentA"), and agent responder process type ("agentB"). Both processes type have states called "mess" plus a number of the message. Each state has an input signal that is triggered when its related message is received. This message is checked before being accepted, and it stops if it is the final state or it composes the corresponding message, sends it, and steps to next state.

In order to treat all messages that are received, we have defined an asterisk state that saves all unprocessed signals, and in a lower priority level, it checks if there are any queued messages. If that is the case, it changes its state to that one related to that signal. Fresh data types are provided adding to their definition a process identification item (PID SDL sort). It is used to differentiate every concurrent session. Those process types can be used in a more complex system where EKE protocol is the authentication procedure.

4.3 Intruder Model

The Intruder model is divided into the exploration algoritm and the check mechanism. These depend on the analysis strategy that the analyzer must evaluate. There are a number of attacks types that have been developed, which can be implemented in our analysis mechanism.

The process type called "CorresAttack" provides the exploration algorithm. This consists on a state that is triggered by any input messages and executes intruder's operation. Then, the divert operation is applied sending message form first session to second session, and reciprocally.(figure 6). This is called the man-in-the-middle attack.

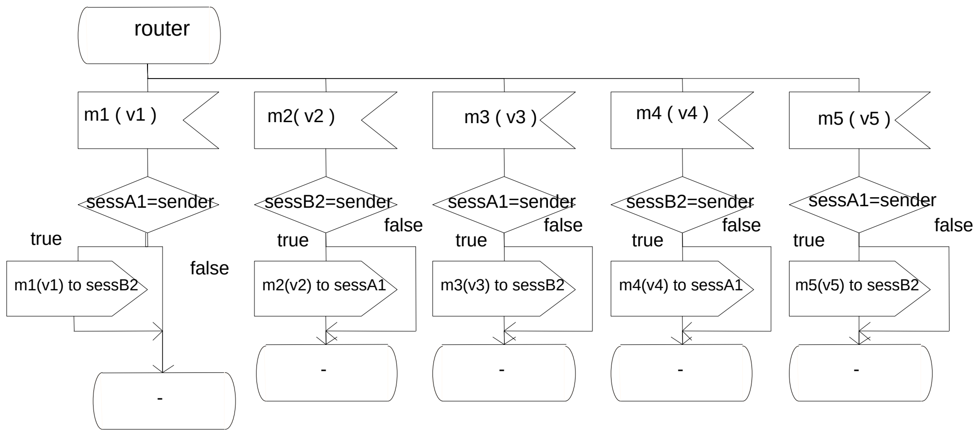


Figure 6: Main state of "CorresAttack" process

In order to check the correspondence flaw we create the process type observer called "obvcorresattack". The main state (*checking*) has two possible alternatives that we can see in the following SDL code:

```
STATE checking
if ((GetState(A2)='final') AND (GetState(B1)='final')) AND(((GetState(B2)/='final')AND  ((Getstate(A1)/='final'))))
then
  REPORT "authentication error"
  STOP
else
  if((GetState(A1)='final') AND (GetState(B2)='final'))AND(((GetState(B1)/='final')AND ((Getstate(A2)/='final'))))
  then
    REPORT "authentication error"
    STOP
  else
    NEXTSTATE checking
```

The first condition is true when agent A of session one and agent B of session two reach a "final" state, and the other agents do not reach it. The second condition checks the inverse situation. When checked condition is true, a report action is executed, and it can stop searching or continue exploring for a new failure scenario. This report is provided in MSC language.

4.4 Verification Procedure

In order to explore the correspondence failure we have defined the processes distribution that is shown in figure 7. The system is specified connecting through the instance of the process type "Corresattack" called "medattack", an instance of process type "agentA" and another of process type "agentB". Those instances are called "A" and "B" respectively.

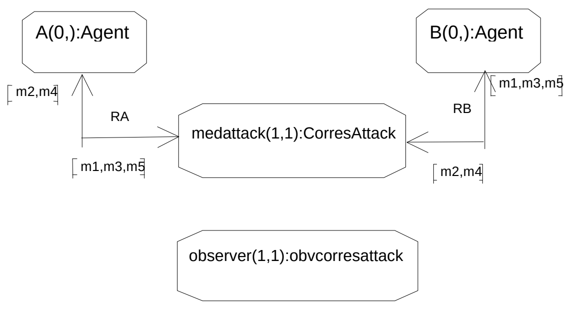


Figure 7: SDL processes distribution

This processes distribution enforce all messages, that are sent between A and B, to cross through the medium instance, where they are processed by intruder's operations. In that case, the intruder's operation provides redirect operation.

4.5 Analysis Result

Following, we load the SDL specification in the SDL Validator tool. Firstly we configure the Validator, in order to evaluate the system correctly. Then, we execute the exhaustive exploration option. It finishes quickly and it generates MSC report. The following code shows how two sessions start at the same time. When the medium process intercepts the messages from a session and sends to the other session, then the agent A of first session and agent B of second session reach a "final" state. This means that the agent A of first session believes it is communicating with agent B of its session but, in fact, it is connected with agent B of second session.

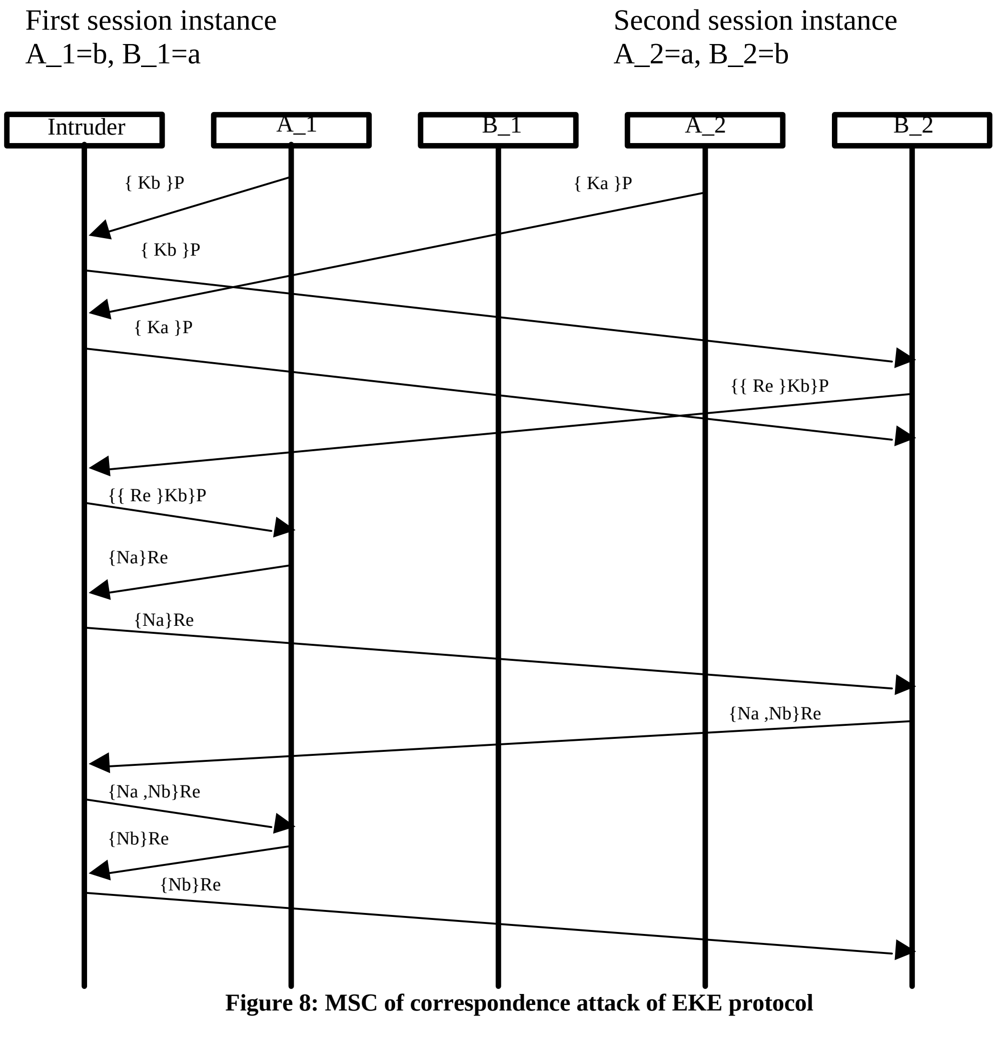


Figure 8: MSC of correspondence attack of EKE protocol

5. CONCLUSIONS AND FUTURE WORKS

We have presented a new mechanism to specify security protocols and their possible attacks. The security protocol is specified in SRSL language and translated into SDL system, and attacks are implemented as SDL processes that develop intruder's behavior and check safety properties. Protocol specification is independent of analysis procedure, so it can be used in others environments.

Several types of security attacks have been analyzed using our method. It has been essential to study how they can be produced in a real environment. We have shown a result of an analysis procedure that explains messages exchange between protocol agents and intruder process.

Actually, we are extending the SRSL in order to represent more complex protocols and to analyze others properties, such as anonymity. In order to check several kinds of attacks, we are gathering a set of generic attacks. Furthermore, we are studying how to implement those attacks in Internet environment.

REFERENCES

[Alur et al. 98] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. *Mocha: modularity in model checking*. CAV 98: Computer-aided Verification, Lecture Notes in Computer Science 1427, pages 521-525. Springer-Verlag, 1998.

[BAN89] M. Burrows, M. Abadi, and R. Needham. *A logic of authentication*. In Proceedings of the Royal Society, Series A, 426(1871):233-271, 1989.

[BeMe92] S.M. Bellovin and M. Merrit. *Encrypted key exchange: Password-based protocols secure against dictionary attacks*. In Proceedings of IEEE Symposium on Reseach in Security and Privacy, pages 72-84, 1992.

[CASR00] CASRUL. http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/.

[DeMi99] G. Denker and J. Millen. *CAPSL intermediate language*. In Formal Methods and Security Protocols,1999. FLOC '99 Workshop.

[DoYa83] D. Dolev and A. Yao. *On the security of public key protocols*. IEEE Transactions on Information Theory, IT-29:198-208, 1983.

[Hogr96] D. Hogrefe. *Validation of SDL systems*. Computer Networks and ISDN Systems, 28 (12):1659–1667, June 1996.

[ITU99b] ITU-T, Geneva, Switzerland. *Message Sequence Charts*, 1999. ITU-T Recommendation Z.120.

[ITU99a] ITU-T, Geneva, Switzerland. *Specification and Description Language (SDL),* 1999. ITU-T Recommendation Z.100.

[Holz91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, 1991.

[Lowe98] G. Lowe. *Towards a Completeness Result for Model Checking of Security Protocols*. 11th IEEE Computer Security Foundations Workshop, pages 96-105. IEEE Computer Society, 1998.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. *Model checking for security protocols*. DIMACS Workshop on Design and Formal Verification of Security Protocols, 1997.

[Mead00] C. Meadows. *Open issues in formal methods for cryptographic protocol analysis*. Proceedings of DISCEX 2000,pages 237-250. IEEE Comp. Society Press, 2000.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. *Automated analysis of cryptographic protocols using Murphi*. In Proceedings of IEEE Symposium on Security and Privacy, pages 141-151. IEEE Computer Society Press, 1997.

[RuTu01]M. Rusinowich, M. Turuani. *Protocol Insecurity with Finite Number of Sessions is NP-complete*. 14th IEEE Computer Security Foundations Workshop June 11-13, 2001

[RySc01]P. Ryan and Scheneider. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001

[SEH97] A. Sarma J. Ellsberger, D. Hogrefe. *SDL–Formal object-oriented language for communication systems*. Prentice-Hall, 1997.