Automatic Generation of the C# Code for Security Protocols Verified with Casper/FDR

Chul-Wuk Jeon, Il-Gon Kim, Jin-Young Choi
Dept. of Computer Science and Engineering, Korea University, Seoul, 136-701 KOREA
{cwjeon, igkim, choi}@formal.korea.ac.kr

Abstract

Formal methods technique offer a means of verifying the correctness of the design process used to create the security protocol. Notwithstanding the successful verification of the design of security protocols, the implementation code for them may contain security flaws, due to the mistakes made by the programmers or bugs in the programming language itself. We propose an ACG-C# tool, which can be used to generate automatically C# implementation code for the security protocol verified with Casper and FDR. The ACG-C# approach has several different features, namely automatic code generation, secure code, and high confidence. We conduct a case study on the Yahalom security protocol, using ACG-C# to generate the C# implementation code.

1 Introduction

With the rapid development of communication networks, the use of security protocols to achieve security goals such as confidentiality, authentication, integrity and non-repudiation is becoming more and more popular. However, many supposedly inviolable security protocols have been proposed in the literature and even exploited in practice, only to be found to be vulnerable later on. Therefore, various formal methods have been developed to verify the safety of security protocols. These include belief logics such as BAN[1] and various tools such as FDR, Murphi, NRL Analyzer, and Isabelle[2].

Notwithstanding the successful verification of the design of security protocols, the implementation code for them may contain security flaws, due to the mistakes made by the programmers or bugs in the programming language itself. In addition, the process of implementing a security protocol is a tedious and time-consuming task. For example, a flaw pertaining to a buffer overflow attack was found in the OpenSSH code of the SSH protocol[3]. This vulnerability did not concern the security protocol itself, but its imperfect implementation.

Accordingly, research into the generation of the program code automatically from a high-level specification of a security protocol is necessary, in order to reduce such vulnerabilities in the implementation phase. If the design of the security protocol is found to be correct, the use of automated translation guarantees that the behavior of the implementation code corresponds precisely to the formal specification.

In this paper, we propose an ACG-C# tool, which can be used to generate automatically the C# implementation code for security protocols from the high-level specification written in a variation of Casper notation. This ACG-C# tool compiles the specification to produce C# code that is a concrete implementation of the protocol.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of Casper notation. Section 3 introduce the notation, the process of code generation, architecture, API of the ACG-C# tool and the advantages of our approach. Finally, section 4 concludes this paper.

2 Casper

Casper was developed by Gavin Lowe in order to express the behavior of security protocols more easily and to allow their security properties to be verified. The Casper tool translates a high-level description of a security protocol into CSP, which is the process algebra of Communicating Sequential Processes developed by Hoare. It was developed to enable the behavior of security protocols to be specified more easily and verify the security properties, such as confidentiality and authentication, to be verified with the FDR model checking tool. The Casper/FDR approach has been very successful over the past few years and has led to the discovery of many security weaknesses in protocols that were thought to be inviolable.

A Casper script consists of two parts: a general part that specifies a model of a system running the protocol and a specific part that defines a particular image a given of a given function by instantiating the parameters of the protocol. we first introduce the basic notation of Casper using the Needham-Schroeder protocol as an example.

Figure 1 shows the “#Free variables” section contains the functions *PK* and *SK* that return a public key and a private key of the agents *a* and *b*, respectively. The term *InverseKeys* defines a pair of keys used to encrypt and decrypt a message. Therefore, *PK(a)* and *SK(a)* are inverses of one another(*PK(a)* : the public key of agent *a*, *SK(a)* : the private key of agent *b*). The “#Protocol description” section defines the sequence of messages in the protocol. The message numbers 0, 1, 2, and 3 denote the message sequence between the agents *a* and *b*. In message 0, agent *a* receives the identity of agent *b* from the environment. This message tells agent *a* to communicate with agent *b*. The “#Specification” section describes the various security properties, such as secrecy and authentication. For example, the notation *Secret(a,na,[b])* means that the agent *a* thinks that the nonce, *na* is a secret that should be known only to himself and agent *b*. The “#Actual variables” section defines the actual datatypes of two honest agents *a*(Alice), *b*(Bob), and an *intruder*(Ivor). The “#System” section defines a single initiator *Alice* and a single responder *Bob* which use nonces *na* and *nb*, respectively. The “#Intruder Information” section specifies the initial knowledge of an intruder to intercept the messages and fake the identity of the agents.

```
#Free variable
a, b : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Protocol description
0.    -> a : b
1.  a -> b : {na, a}{PK(b)}
2.  b -> a : {na, nb}{PK(a)}
3.  a -> b : {nb}{PK(b)}

#Processes
INITIATOR(a,na) knows PK, SK(a)
RESPONDER(b,nb) knows PK, SK(b)

#Specification
Secret(a,na,[b])
Secret(b,nb,[a])
Agreement(a,b,[na,nb])
Agreement(b,a,[na,nb])

#Actual variables
Alice, Bob, Ivor : Agent
Na, Nb, Nm : Nonce

#Function
symbolic PK, SK

#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Nb)

#Intruder Information
Intruder = Ivor
IntruderKnowlege
={Alice, Bob, Ivor, Nm, PK, SK(Ivor)}
```

Figure 1. Needham-Schroeder protocol Casper Script

3 The ACG-C#(Automatic Code Generator into C#) tool

3.1 Overview

The term ACG-C# stands for Automatic Code Generator into C# code. It’s basic notation and the code generation process are based on the COSP-J tool that *Xavier Didelot* of Oxford University developed[4]. The ACG-C# tool can minimize the inconsistency between the formal specification and the executable implementation code for the security protocol. The ACG-C# tool automatically generates the C# implementation code from the high-level description of the security protocol verified with Casper/FDR.

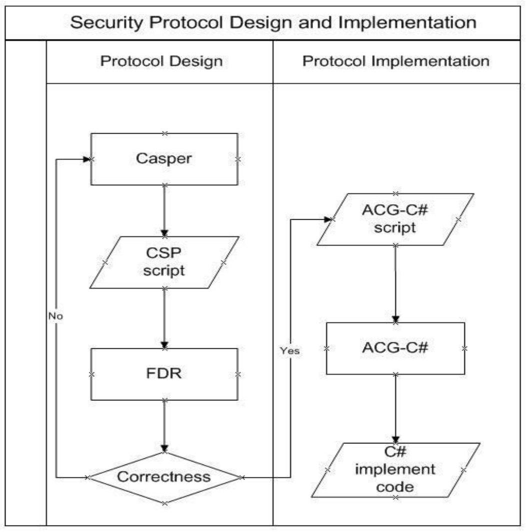


Figure 2. The verification and code generation processes for the security protocol

Figure 2 shows the overall process used to verify the safety of the security protocol and to implement it. The entire process is composed of two subprocesses, which are referred to as protocol design and protocol implementation. To guarantee the safety of the security protocol in the design phase, we first make an abstract model of the protocol with Casper and then generate CSP code with the compilation function of Casper. Next, we can run the FDR model checking tool to verify whether the security protocol satisfies the security properties or not. If the security properties are satisfied, then the designer inputs the slightly modified Casper script into the ACG-C# tool. In the implementation phase, the ACG-C# tool automatically generates the C# implementation code for the security protocol. The choice of C# as the target language is due to the support that the C# language provides for security mechanisms, such as type-

```
#Protocol description
1. a -> b : {a,na}{PK(b)}
2. b -> a : {na,nb}{PK(a)}
3. a -> b : {nb}{PK(b)}
4. a ->   : b,na,nb
5. b ->   : a,na,nb

#Free variables
a, b : Agent
na,nb : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK)

#External
key.xml : KeyStore

#Functions
PK = key.xml.PK
SK = key.xml.SK

#Processes
INITIATOR(a,b) knows PK,SK(a)
generates na
RESPONDER(b) knows PK,SK(b)
generates nb
```

Figure 3. ACG-C# input script

safety, security policy, authentication and authorization[5]. It also supports security libraries such as secret and communication service provider APIs.

3.2 The input notation in ACG-C#

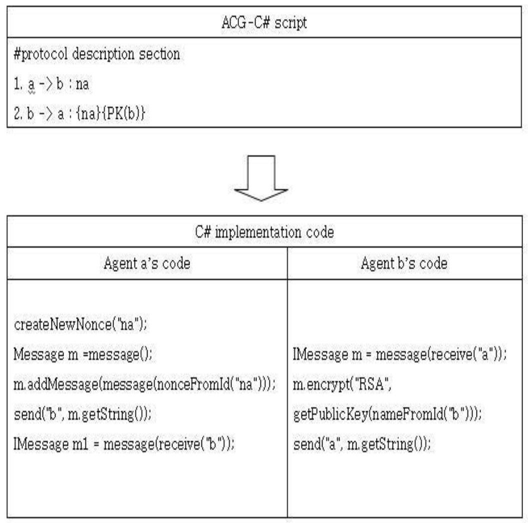
The structure of the input script of ACG-C# is adapted from the general structure of the input script of Casper. When ACG-C# generates the C# implementation code, there is no need for all eight sections of the Casper script.

The "#System" section, "#Actual variable" section, and "#Intruder Information" section in Casper are related to the definition of the agents taking part in the actual system, the roles they play and the intruder's abilities. However, generating the implementation code is not absolutely necessary. Instead, ACG-C# needs to be more specific for implementation code. In the "#Free variable" section, functions *PK* and *SK* do not really need to be defined in the Casper script. If the designer wants to RSA encryption, function "*PK: agent* $\rightarrow$ *PublicKey*" replaces "*PublicKey*" with "*RSAPublicKey*" and also "*SK: agent* $\rightarrow$ *SecurityKey*" replaces "*SecretKey*" with "*RSASecretKey*". In practice, the function *PK* and *SK* should provide running code with the secret key or public key of a given agent. In order to store these public and secret keys used in an xml file. The running code gets the public keys and the private keys in an xml file. Lastly, the "#External" section does not exist in Casper. This section describes the key's store position. The casper script for the modified Needham-Schroeder protocol is Figure 3.

3.3 Architecture

ACG-C# separates the implementation of the protocol notation from that of the network communications and cryptographic algorithms.

- Protocol notation - this part of the architecture is related to the "#Protocol description" section in Casper script. This code maintains the protocol state, determines when and if messages are sent and received, checks the contents of the outgoing messages and the incoming messages, as well as stores the message components.



ACG-C# script

```
#protocol description section
1. a -> b : na
2. b -> a : {na}{PK(b)}
```

C# implementation code

Agent a's code

```
createNewNonce("na");
Message m =message();
m.addMessage(message(nonceFromId("na")));
send("b", m.getString());
IMessage m1 = message(receive("b"));
```

Agent b's code

```
IMessage m = message(receive("a"));
m.encrypt("RSA",
getPublicKey(nameFromId("b")));
send("a", m.getString());
```

Figure 4. Protocol notation from ACG-C# to C# code

In Figure 4, the "#Protocol description" section in Casper script, the desired security protocol is separated into two agents for the sake of computer networking. Agent *a* creates the new nonce, *na* and then sends it. Agent *b* receives nonce, *na*. Similarly, Agent *b* encrypts the received nonce with RSA encryption algorithm and then sends it.

- Secret and Communication Provider(SCP) - this part of the architecture involves provider specific code that handles the concatenating and separating of message components into the byte stream, implements the cryptographic algorithms, and manages the network protocol specific aspects. The Secret and Communication Provider forms the common base when the C# implementation code is running.

3.4 SCP(Secret and Communication Provider) API

SCP API provides the C# implementation code with runtime access to the Secret and Communication library. The *agent* class provides the functions required for the two agents to communicate with each other. The *agent* class contains the agent's ID, IP, socket, and opponent fields. The ID field represents the identity of the agent as it is used in the protocol description section of the input script. The IP field is the IP address which the agent uses for computer networking. Socket is an instance of the class socket that is used to communicate with the agent. The opponent can get the IP address that the agent uses to communicate with another agent. The *message* class represents the messages sent and received during communication. The *message* class deals with operations involving the message's contents, such as encryption, decryption, and so on. The *message* class includes the encrypt method, decrypt method, hash method, and addMessage method.

The generated C# implementation code uses the *protocol* class method. The *protocol* class uses a design pattern, abstract factory, which allows different providers to be plugged into the SCP API. It provides the Protocol notation code with a single point of access to instances of the concrete provider classes that are used to implement the SCP API defined interfaces, and the cryptographic and network operations.

Compared with COSP-J, the entire architecture is changed due to the features of the C# language. In ACG-C#, the class, *MessageT*, which is itself composed of the class *Message*, is used to implement the interface *IMessage*. This architecture can easily be extended by changing the class *Message*.

3.5 Advantages

- No error-prone : If the programmer is inexperienced, the security protocol is likely to be flawed. ACG-C# precludes the possibility of the programmer making such mistakes. Furthermore, the ACG-C# tool helps the programmer to reduce tedious and time-consuming tasks.

- Secure code : With the managed code in C#, the CLR(Common Language Runtime) can perform checks for type safety, memory overwrites, memory management, and garbage collection. This results in a reduction of memory leaks and related issues. The CLR performing the memory management prevents the code from buffer overflow, accessing memory directly, eliminating pointers, and greatly reducing crashes or other memory-overwrite behaviors.

- High confidence : Using the well-defined Casper script can provide guarantees that the desired security properties are satisfied, thus imparting high confidence.

- Strong key : People tend to choose their passwords poorly[6]. To prevent guessing attacks, a good random number generator is needed. The key generator generates DES keys which have 10^8 possible variations and keys are saved in an XML file.

4 Conclusion

The security protocol implementation process can cause difficulties, due to; a lack of expertise and experience on the part of the programmer and bugs in the implementation language, which lead to the implemented security protocols behaving incorrectly. The automatic code generation approach not only eliminates these shortcomings, but offers several advantages, such as its automatic code generation, type safety, and high confidence. Using Casper provides the designer with high confidence and high quality in terms of the analysis of the security protocol. Once the security protocol has been verified with Casper/FDR, the ACG-C# tool generates the C# implementation code. This code is guaranteed to be free of designer induced buffer overflow, memory leaks, and type-flaw attacks. We believe that using formal methods and the automatic code implementation approach is the efficient way to guarantee the safety of security protocols in the design and the implementation steps.

References

[1] M.Burrows, M.Abadi, and R.Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, pp.18-36, 1990.

[2] P.Y.A. Ryan and S. A, Schneider. Modeling and analysis of security protocols: the CSP Approach. *Addison-Wesley*, 2001.

[3] M.Zalewski. Remote vulnerability in SSH daemon crc32 compensation attack detector. Available from http://razor.bindview.com/publish/advisories/adv_ssh1crc.html, 2001.

[4] X.Didelot. COSP-J: A Compiler for Security Protocols. *MSc dissertation*, Available from http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/COSPJ/, 2003.

[5] K.Shakil. Security Features in C#, The article, Available from http://www.developersdex.com/gurus/articles/5.asp, 2004.

[6] Gavin Lowe. Analysing Protocols Subject to Guessing Attacks. *Journal of Computer Security*, 2004.