DOI: 10.15514/ISPRAS-2019-31(3)-15

C# parser for extracting cryptographic protocols structure from source code

I.A. Pisarev, ORCID: 0000-0002-2055-1841 <ilua.pisar@gmail.com>
L.K. Babenko, ORCID: 0000-0003-2353-7911 <lkbabenko@sfedu.ru>
Southern Federal University, Department of Information Security,
Taganrog, Rostov region, 347928, Russia

Abstract. Cryptographic protocols are the core of any secure system. With the help of them, data is transmitted securely and protected from third parties' negative impact. As a rule, a cryptographic protocol is developed, analyzed using the means of formal verification and, if it is safe, gets its implementation in the programming language on which the system is developed. However, in the practical implementation of a cryptographic protocol, errors may occur due to the human factor, the assumptions that are necessary for the possibility of implementing the protocol, which entail undermining its security. Thus, it turns out that the protocol itself was initially considered to be safe, but its implementation is in fact not safe. In addition, formal verification uses rather abstract concepts and does not allow to fully analyze the protocol. This paper presents an algorithm for analyzing the source code of the C# programming language to extract the structure of cryptographic protocols. The features of the implementation of protocols in practice are described. The algorithm is based on the searching of important code sections that contain cryptographic protocol-specific constructions and finding of a variable chain transformations from the state of sending or receiving messages to their initial initialization, taking into account possible cryptographic transformations, to compose a tree, from which a simplified structure of a cryptographic protocol will be extracted. The algorithm is implemented in the C# programming language using the Roslyn parser. As an example, a cryptographic protocol is presented that contains the basic operations and functions, namely, asymmetric and symmetric encryption, hashing, signature, random number generation, data concatenation. The analyzer work is shown using this protocol as an example. The future work is described.

Keywords: cryptographic protocols; C#; parser; verification; tree; analysis; source code

For citation: Pisarev I.A., Babenko L.K. C# parser for extracting cryptographic protocols structure from source code. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 191-202. DOI: 10.15514/ISPRAS-2019-31(3)-15

Acknowledgment. The work was supported by the Ministry of Education and Science of the Russian Federation grant № 2.6264.2017/8.9.

C# парсер для извлечения структуры криптографических протоколов из исходного кода

И.А. Писарев, ORCID: 0000-0002-2055-1841 <ilua.pisar@gmail.com>
Л.К. Бабенко, ORCID: 0000-0003-2353-7911 <lkbabenko@sfedu.ru>
Южный федеральный университет, Кафедра информационной безопасности,
Таганрог, Ростовская область, 347928, Россия

Аннотация. Криптографические протоколы являются ядром любой защищенной системы. С их помощью передаются данные, которые нуждаются в защите от третьих лиц. Как правило, криптографический протокол разрабатывается, анализируется с использованием средств формальной верификации и, если он безопасен, реализуется на языке программирования, на котором разрабатывается система. Однако при практической реализации криптографического протокола могут возникать ошибки из-за человеческого фактора, предположений, которые необходимы для возможности реализации протокола, что влечет за собой подрыв его безопасности. Таким образом, оказывается, что сам протокол изначально считался безопасным, но его реализация на самом деле небезопасна. Кроме того, формальная верификация использует довольно абстрактные понятия и не позволяет полностью проанализировать протокол. В данной статье представлен алгоритм анализа исходного кода языка программирования C# для извлечения структуры криптографических протоколов. Описаны особенности реализации протоколов на практике. Алгоритм основан на определении ключевых областей кода, содержащих специфические для криптографических протоколов конструкции, и определении цепочки преобразований переменных из состояния отправки или получения сообщений до их начальной инициализации с учетом возможных криптографических преобразований для составления дерева, из которого будет извлечена упрощенная структура криптографического протокола. Алгоритм реализован на языке программирования C# с использованием синтаксического анализатора Roslyn. В качестве примера представлен криптографический протокол, который содержит основные операции и функции, а именно: асимметричное и симметричное шифрование, хеширование, подпись, генерация случайных чисел, конкатенация данных. Работа анализатора показана с использованием этого протокола в качестве примера. Описана будущая работа.

Ключевые слова: криптографические протоколы; C#; парсер; верификация; дерево; анализ; исходный код.

Для цитирования: Писарев И.А., Бабенко Л.К. C# парсер для извлечения структуры криптографических протоколов из исходного кода. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 191-202 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-15

Благодарность. Работа выполнена при поддержке Министерства образования и науки Российской Федерации, грант № 2.6264.2017/8.9.

1. Introduction

The problem of verifying the security of cryptographic protocols is relevant nowadays despite the existence of a large number of already verified protocols. The need to use self-written protocols that use lightweight cryptography for IoT, mobile robots, as well as the imperfection of formal verification of protocols is a new challenge for verification methods, in particular, the possibility of verifying the security of cryptographic protocols implementation. Nearly all protocols are changed and supplemented during implementation, and for their initial analysis, for example, by means of formal verification this is not taken into account. Also there can be programming mistakes and logic flaws on source code. So we need verify cryptographic protocols on their last developing iteration - on implementation level for more attack finding which can help make any system more secure. Due to this fact this work is actual nowadays. The primary task in this matter is to extract the structure of the protocol from the source code. At the moment there are works in which the problem of extracting an abstract model from the source code of programming languages C [1-3], Java [4-6], F# [7-12] is being considered. Most of them require a special programming style for the possibility of use these algorithms or the use of additional annotations in the source code. The paper proposes to analyze the source code of the C# programming language. There are no other works, in which code analysis would be carried out, not involving the use of annotations or a special programming style.

2. Cryptographic protocols

Cryptographic protocols are a set of cryptographic algorithms and functions, with a correct combination of which is obtained a secure process of transferring messages between the parties. Protocol security is defined as complying with security requirements, the main of which are mutual authentication of the parties, protection against time attacks such as replay attacks, privacy and integrity of the transmitted data. Below is an example of a test protocol that does not have a special

meaning, but contains all the basic cryptographic algorithms and functions: asymmetric and symmetric encryption, hashing, signature, random number generation.

1. $A \to B$: $E_{pkB}(A, Na)$
2. $B \to A$: $E_{pkA}(Na, Nb, B)$
3. $A \to B$: $E_{pkB}(Nb, k)$
4. $B \to A$: $E_k(M1, E_{pkA}(M2)), Hash(M1)$
5. $A \to B$: $E_k(M1, M2, M3), Sign_{skA}(M1, M2, M3)$
6. $B \to A$: $E_k(M3)$

At the beginning of this protocol, messages 1-3 use the Needham–Schroeder public key protocol (NSPK) [13] for mutual authentication of the parties. In message 3, in addition to the random number Nb, the key k is also transmitted for further communication between the parties using a symmetric cipher. In message 4, $M2$ data is transmitted, asymmetrically encrypted on partys' A public key, and some $M1$ data. All this is encrypted symmetrically using the key k, after which the data hash $M1$ is applied. In message 5, side A applies its $M3$ data to the previously sent data $M1$ and $M2$, encrypts all this symmetrically on key k, applies a signature and sends this message to side B. In message 6, B sends A $M3$ data encrypted symmetrically on key k.

3. Features of the cryptographic protocols implementation

There are a number of problems with the implementation of cryptographic protocols. One of the problems is the dynamic size of messages. In the programming language, the transfer of messages between the parties is implemented using sockets. In this case, the party that receives the message must know in advance the size of the buffer to receive. For example, in the protocol described in the previous paragraph, in the first three messages random numbers and identifiers of the parties with a fixed length are used. In this case, everything is simple and at the reception of the message by the party, it will expect a previously calculated static message length. However, messages 4-6 use data $M1, M2, M3$, which may have different lengths. For example, in message 4, $M1$ data can be a video file, the length of which can vary from 1 MB to several GB. And the question is how to tell the receiving party the size of the receiving buffer. There are various options for how this can be done, for example, to add information about its length to the beginning of a message, to put a mark at the end of the message. Let us consider in more detail the option with the addition of information about the length of the message. This option involves the use of additional data before the main message, which will contain the size of the future message. An example of a message with additional size information is shown in fig. 1.

Fig. 1. Additional information about the size of the message

The receiving party in this case receives a fixed array of bytes, which contains the size of the message, after which the second portion takes the rest of the message knowing in advance its length.

```
A send: Buffer size, Message
B receive(4 bytes): Buffer size
B receive(Buffer size): Message
```

Since *Message* is usually encrypted and, in the context of a protocol, its transmission is protected, the question arises of how to protect information in *Buffer size*. All security requirements are important for us, except secrecy. To ensure them, you can, for example, use the signature of this area with timestamps. Thus, the transmission, for example, message 4, will have the following form when implementing the protocol:

$B \to A$: $Buffer\ size, T, Sign_{skB}(Buffer\ size, T),\ E_k(M1, E_{pkA}(M2)), Hash(M1)$

Another way is to get data into a fixed-length buffer until the buffer becomes empty. In this case, problems can also arise as shown in fig. 2.

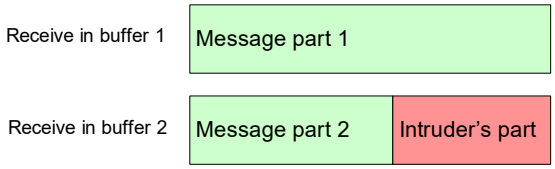

Fig. 2. Intruders' attack on the addition of real data

The result is that the message will be received longer than necessary and in some implementations, in which further processing of the message by the receiving party is tied to the use of the message length, some data may be imperceptibly corrupted when decrypting and dividing the data into the message elements (random numbers, keys, etc.). In order to avoid this, various methods of controlling the length of a message are also used.

4. Source code analysis algorithm

As an example for describing the operation of the algorithm, the previously considered protocol was taken and implemented in the C# programming language in the form of a client server application.

1. $A \to B$: $E_{pkB}(A, Na)$
2. $B \to A$: $E_{pkA}(Na, Nb, B)$
3. $A \to B$: $E_{pkB}(Nb, k)$
4. $B \to A$: $E_k(M1, E_{pkA}(M2)), Hash(M1)$
5. $A \to B$: $E_k(M1, M2, M3), Sign_{skA}(M1, M2, M3)$
6. $B \to A$: $E_k(M3)$

The analysis algorithm uses the C# Roslyn source code parser [14]. With it you can get the tree structure of the source code, and you can use filters. We need these filters:

1) InvocationExpressionSyntax – call expressions;
2) VariableDeclarationSyntax – declaration of variables;
3) AssignmentExpressionSyntax – an assignment expression;
4) IfStatementSyntax – statement with a condition statement.

Using filters, you can get the desired expression, after which you can view the tree structure of this expression. For example, using «*AssignmentExpressionSyntax*» we can find the expression «$M1enc1 = RSA.Encrypt\ (M1, true)$». The derived linear tree structure of the expression is shown in fig. 3.

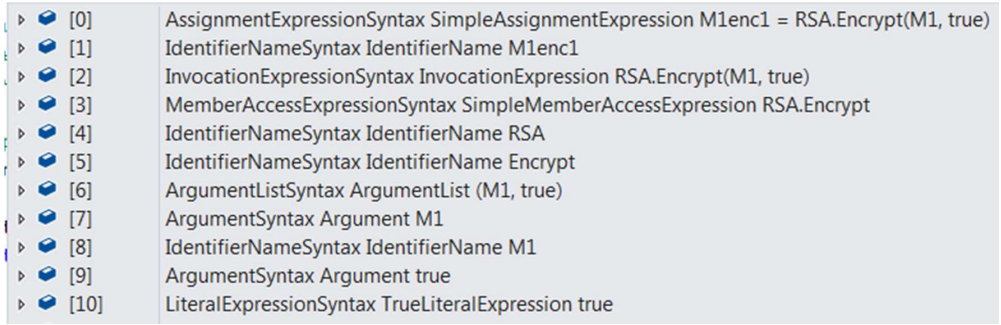

Fig. 3. Tree structure of expression in a linear form

The main purpose of using this parser is to find the transition from one variable to another. In this case, we are interested in the transition $M1enc1 \rightarrow M1$. This is achieved by searching for data such as «*IdentifierName*» together with the use of a black list of expressions. For example, it uses the call of the «*Encrypt*» method, as well as the previously declared object of the asymmetric encryption class «*RSA*», which are present in the black list, and $M1enc1$ and $M1$ that we need can be obtained from here, where the first element will be the variable to which the value will be assigned, and the rest of those that are lower and not included in the black list will be the new value assigned.

The algorithm is based on the definition of important code sections containing constructs specific to cryptographic protocols. Ultimately, the task is to find a chains of variables transformation from the state of sending or receiving messages (socket send/receive) to their initial initialization (static initialization, load from file, etc.), while taking into account possible cryptographic transformations (hash, encryption, etc.). In the course of building a chain, a tree is constructed, the nodes of which are variables with additional information about them, including data type definitions for the final leaves of the tree and cryptographic algorithms in the tree nodes. The tree structure allows you to describe all the chains of data transformations, since the data in the message is combined in various ways, the chains can be strongly branched and joined. Below is a fragment of the source code for the implementation of a part of the cryptographic protocol (messages 1-3) from participant A.

```
1     ...
2     Socket socA =
3     new Socket(ipAddress.AddressFamily,
4     SocketType.Stream, ProtocolType.Tcp);
5
6     socA.Connect(remoteEP);
7
8     RNGCryptoServiceProvider rng = new
9     RNGCryptoServiceProvider();
10
11    byte[] A = new byte[] { 132, 114 };
12    byte[] B = new byte[] { 15, 245 };
13
14    byte[] Na = new byte[64];
15    rng.GetBytes(Na);
16
17    byte[] M1 = new byte[2 + 64];
18
19    Array.Copy(A, 0, M1, 0, A.Length);
20    Array.Copy(Na, 0, M1, 2, Na.Length);
21
22    //1
23    byte[] M1enc;
24    using (RSACryptoServiceProvider RSA =
25    new RSACryptoServiceProvider())
26    {
27       RSA.ImportParameters(
28    rsaPB.ExportParameters(false));
29       M1enc = RSA.Encrypt(M1, true);
30    }
31
32    socA.Send(M1enc);
33
34    //2
35    byte[] MGet2Encr = new byte[256];
36    socA.Receive(MGet2Encr);
37
38    byte[] MGet2;
```



```
39    using (RSACryptoServiceProvider RSA = new
40    RSACryptoServiceProvider())
41    {
42       RSA.ImportParameters(
43    rsaSA.ExportParameters(true));
44       MGet2 = RSA.Decrypt(MGet2Encr, true);
45    }
46
47    byte[] BFromServer = new byte[2];
48    byte[] NaGet = new byte[64];
49    Array.Copy(MGet2, 0, BFromServer, 0, 2);
50    Array.Copy(MGet2, 0, NaGet, 0, 64);
51
52    if (!NaGet.SequenceEqual(Na) &&
53    !B.SequenceEqual(BFromServer))
54    {
55       socA.Shutdown(SocketShutdown.Both);
56       socA.Close();
57       return;
58    }
59
60    byte[] Nb = new byte[64];
61    Array.Copy(MGet2, 64, Nb, 0, 64);
62
63    byte[] k = new byte[32 + 16];
64    rng.GetBytes(k);
65
66    byte[] M3 = new byte[0];
67    M3 = Nb.Concat(k).ToArray();
68
69    //3
70    byte[] M3enc;
71    using (RSACryptoServiceProvider RSA = new
72    RSACryptoServiceProvider())
73    {
74       RSA.ImportParameters(rsaPB.ExportParameters(false));
75       M3enc = RSA.Encrypt(M3, true);
76    }
77    socA.Send(M3enc);
78    ...
79
```

First you need to define the declaration and initialization:

- objects of class *Socket*.
- class objects of the standard library cryptographic algorithms, such as the *RSACryptoServiceProvider* asymmetric encryption algorithm, the *RNGCryptoServiceProvider* random number generator, etc.

The variables of the class object *Socket: [socA]*, classes of cryptographic algorithms are defined: *[rng, RSA]*.

To find variable of the Socket class object, the sending and receiving messages is searched. In this case, there are 3 such constructions. At this stage, you can construct an interaction scheme of the following form:

1. $A \rightarrow B: M1$
2. $B \rightarrow A: M2$
3. $A \rightarrow B: M3$

To determine the structure of the message, it is necessary to build a tree, the nodes of which contain variables with additional information. Consider an example for determining the content of the first message. The order of the algorithm is as follows,

1. The expression of the first message *socA.Send (M1enc)* is taken as the root of the tree. It is necessary to understand the contents of the variable *M1enc*.
2. First you need to find the declaration of the variable *M1enc* using the filter *VariableDeclarationSyntax*. However, in our case, the variable is declared, but not initialized (line 23). In this case, the filter *AssignmentExpressionSyntax* is used and you can find in line 29 the assignment of the value to our variable. *M1enc* is added as a child node with the «*var*» tag, which means it is just a variable.
3. The simplest case of assignment is when the value of one variable is assigned to another. In this case, the situation is more difficult. The variable *M1enc* is assigned the value of the result of the work of the *Encrypt* method for an object of the asymmetric encryption class *RSACryptoServiceProvider*, which takes two parameters as input: what to encrypt and flag whether to use optimal asymmetric encryption with addition (OAEP padding). At the current stage, we remember that the content of the variable *M1* was asymmetrically encrypted and assigned to the variable for sending message 1. In the tree structure, this is displayed as adding a child node *M1* with the note «*AsymENC*», which means that the value of the variable *M1* is encrypted using an asymmetric cipher.
4. Similar to paragraph 2, we are looking for the initialization of the variable *M1*. Using the first filter, you can find out that the variable is a one-dimensional array (line 17). Using the second filter, you must find the assignment of values to our array. These are lines 19 and 20. Two children *Na* and *A* with the mark «*var*» are added to node *M1*.
5. For variable *A*, the final value can be found using the first *VariableDeclarationSyntax* filter (line 11). This is where static initialization occurs in the source code. It is enough for a person to simply understand that this is the initial value, but for the automated determination of this fact it is necessary to understand that this is not a variable. One way to solve this problem is to re-search the right side of the expression, and since more in the design code of the assignment is not detected, this value is final. In the tree structure for node *A*, the initialization leaf is added «*new byte [] {132, 114};*» marked «*DATA*», which means the presence of some semantic data in the variable *A*.
6. For the *Na* variable, the search is carried out further. Using filters, we look for the declaration of the array and its initialization. The declaration occurs in line 14, and initialization occurs in line 15 by calling some method of the *rng* variable, which in turn is an object of the *RNGCryptoServiceProvider* class of random numbers, thus, the value of this variable is defined as a random number. The last leaf «*rng.GetBytes (NaPrev);*» is added to the tree structure marked «*RANDOM*», which means generating a random number.
7. Further search initialization for current leaves gives nothing, therefore the structure of the tree is considered final. The output tree view is shown in fig. 4 in the «Full tree» area and it corresponds to the following chain: *Send (M1enc) -> M1enc = E (M1) -> M1 = {A, Na} -> A = new byte [] {132, 114}, Na = rand ()*. You can also see short tree structure and result message from it.

5. Return data problem

At the moment there is a problem in determining the returned data. For example, in message 1, a random number *Na* is sent, and then in the second message it is sent back. By default, there are currently two data concepts: *DATA* and *RANDOM*. All that is not a random number – is considered semantic data, for example: keys, identifiers, transferred files, etc. And at this stage, all values are considered different. For example, for the following protocol:

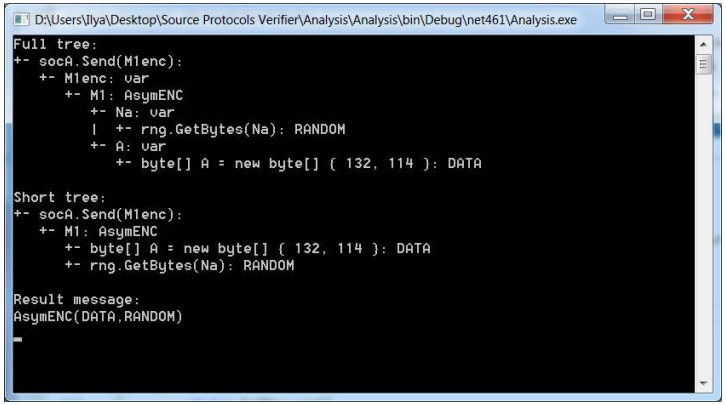

Fig. 4. Output for composing the structure of a single message

1. $A \rightarrow B: Ek(Na, A)$
2. $B \rightarrow A: Ek(Nb, B)$

The result of the work will be as follows:

1. $A \rightarrow B: SymENC(RANDOM, DATA)$
2. $B \rightarrow A: SymENC(RANDOM, DATA)$

And in our context, the default *DATA* in the first message is different from the one in the second message. If the protocol takes the following form:

1. $A \rightarrow B: Ek(Na, A)$
2. $B \rightarrow A: Ek(Nb, Na)$

There is a problem. *Na* just comes back, and on the receiving side we need to understand that this is the same data. For example, when processing message 2 (lines 34-58), we can trace the separated parts. In line 50, the value of the random number *Na* is obtained, after which it is checked for coincidence with what was sent in line 52. Most often in the context of cryptographic protocols, returned values are used for mutual authentication. There can be 2 types: the return of the same number or the return of a function from this number. In both cases, the return value is checked for a match with the one sent earlier. In our case, this is line 53. However, another value is checked here – identifier B. In this case, one of the solutions to this problem would be to find the situation when the variable was sent, and then a value is checked for a match with this variable. In this case, you can assume that this is the case of the return value. However, there may be a number of problems, in particular, just the occurrence of an error in writing code, or simply the absence of such a check of the return value. At the moment, the abstract notion of the type of the *RETURN* variable is used. This means that a variable of this type was returned in the current message.

6. Protocol output structure

Using the algorithm presented in the preceding paragraphs, the complete output structure of the protocol is constructed according to the messages. It is obtained both in short form for formal verification, and in full form for dynamic verification. The full view contains the last variable, before serving in the cryptographic function, the names of the last variables and their initial initialization, for example, static in the code or loading data from a file. Dynamic analysis will be considered in further work and therefore the contents of the full protocol can be changed.

Short view:

1. $A \rightarrow B: AsymENC(DATA, RANDOM)$
2. $B \rightarrow A: AsymENC(RETURN, RANDOM, DATA)$
3. $A \rightarrow B: AsymENC(RETURN, RANDOM)$
4. $B \rightarrow A: SymENC(DATA, AsymENC(DATA)), HASH(DATA)$
5. $A \rightarrow B: SymENC(RETURN, RETURN, DATA), Sign(RETURN, RETURN, DATA)$
6. $B \rightarrow A: SymENC(RETURN)$

Full view:

1) $A \rightarrow B: AsymENC(DATA, RANDOM)$

```
M1 | byte[] A = new byte[] { 132, 114 } | rng.GetBytes(Na)
```

2) $B \rightarrow A: AsymENC(RETURN, RANDOM, DATA)$

```
M2 | socB.Receive(MGet1) | rng.GetBytes(Nb) |
byte[] B = new byte[] { 15, 245 }
```

3) $A \rightarrow B: AsymENC(RETURN, RANDOM)$

```
M3 | socA.Receive(MGet2Encr) | rng.GetBytes(k)
```

4) $B \rightarrow A: SymENC(DATA, AsymENC(DATA)), HASH(DATA)$

```
ForEncM4 | byte[] M1forSend = File.ReadAllBytes("Mess1.txt") | M2forSend |
byte[]   M2forSend   =   File.ReadAllBytes("Mess2.txt")   |   M1forSend   |
byte[] M1forSend = File.ReadAllBytes("Mess1.txt")
```

5) $A \rightarrow B: SymENC(RETURN, RETURN, DATA), Sign(RETURN, RETURN, DATA)$

```
ConcatMess5 | socA.Receive(MGet4) | socA.Receive(MGet4) |
byte[] M3forSend = File.ReadAllBytes("Mess3.txt") | ConcatMess5 |
socA.Receive(MGet4) | socA.Receive(MGet4) |
byte[] M3forSend = File.ReadAllBytes("Mess3.txt")
```

6) $B \rightarrow A: SymENC(RETURN)$

```
M3From5 | socB.Receive(MGet5)
```

7. Experiments

For testing parser on real project we take our previous project - e-voting system based on blinded intermediaries [15], which implemented on C# language. It consists 3 main components: Voter application, Authentication server, Voting server. The protocol in main voting stage is:

1. $AS \rightarrow V: E_{vas}(N_{as})$
2. $VS \rightarrow V: E_{vvs}(N_b, N_{vs})$
3. $VS \rightarrow AS: E_{asvs}(N_{asvs})$
4. $V \rightarrow AS: E_{vas}(N_{as}, userData, E_{vvs}(N_{vs}, N_v, filledBallot))$
5. $AS \rightarrow VS: E_{asvs}(N_{asvs}, E_{vvs}(N_{vs}, N_v, filledBallot)))$
6. $VS \rightarrow AS: E_{asvs}(N_b, N_{asvs}, \text{“}good\text{”})$
7. $VS \rightarrow V: E_{vvs}(N_v, N_{vs}, checkID)$

Before the protocol session keys *vas*, *vvs*, *asvs* were generated with ECDHE (the Diffie-Hellman protocol on elliptical curves using ephemeral keys and signing the secret parts) protocol. So at the beginning of the main voting protocol session keys are created. It is necessary to say that *Nb* is a number of blinding, a non-random random number, which is regenerated each time. It is introduced in order to add some data before the semantic random number for making full search more complicated (in particular, it is necessary to select two encryption keys for message 7 in order to find *userData*). Randomly generated random numbers are sent to authenticate the parties as shown in (1)-(3). The message (4) uses the principle of blind intermediaries. The voter encrypts his vote *filledBallot* on the session key with *VS*, applies his personal data to the ciphertext, and encrypts it on the session key with *AS*. *AS* hashes the sent personal data, searches for the hash in the database

and, and, if detected, redirects the message to the *VS* component. *VS* memorizes the vote, generates a *checkID* through which the user can check his vote after the end of the election, and sends it to the user.

Code organization of cryptographic protocols in this project is simple. Message sending or receiving located in methods' block, so there is no difficult code structure. Our parser was launched for this project and we cad this result:

1. $A \rightarrow B: SymENC(RANDOM)$
2. $C \rightarrow B: SymENC(RANDOM, RANDOM)$
3. $C \rightarrow A: SymENC(RANDOM)$
4. $B \rightarrow A: SymENC(RETURN, DATA, SymENC(RETURN, RANDOM, DATA))$
5. $A \rightarrow C: SymENC(RETURN, RETURN)$
6. $C \rightarrow A: SymENC(RANDOM, RETURN, DATA)$
7. $C \rightarrow B: SymENC(RETURN, RETURN, DATA)$

As we can see from output cryptographic protocol structure was extracted correctly. It is necessary to say that in message 4 A gets «*SymENC(RETURN,RANDOM,DATA)*», but in message 5 it sends this like «*RETURN*». So side A doesn't know key for decryption and for it this is some data that was sent to it and it sends this data to another side so there is 1 element «*RETURN*» instead of 3.

7. Future work

Future work primarily includes a segmentation of DATA semantic data into classes:

1) party identifiers;
2) keys;
3) timestamps;
4) authentication Codes;
5) data received from the user.

It is also an important point to determine the ownership of a key by any of the parties in the case of asymmetric encryption, and to the list of parties in the case of symmetric encryption. Support for protocols involving more than two parties will also be needed. In addition, a complete solution to the problem of accurately determining the returned data is necessary to make it possible to build a complete structure of a cryptographic protocol and its further analysis using formal verification tools. After obtaining the structure of the cryptographic protocol, it is necessary to develop an algorithm for automated translation into the specification language of the most well-known protocol verification tools, such as Avispa [16], Scyther [17], ProVerif [18], and others. It is also necessary to improve the parser. At the moment, the structure can only be retrieved from areas of code where all functions for sending and receiving messages are combined into one block, for example, into the body of a function or class method. In the future, it is planned to improve the parser to work with complex code structures.

8. Conclusion

An algorithm was presented for analyzing the source code of the C# programming language for extracting the structure of cryptographic protocols, based on identifying important code sections that contain cryptographic protocol-specific constructions and determining the chain of variable transformations from the sending or receiving status to their initial initialization, taking into account possible cryptographic transformations to compose a tree, from which it is possible to get simplified structure of a cryptographic protocol. An example of a protocol containing all cryptographic functions is given. The output structure of the cryptographic protocol is shown. Successful practical testing on real e-voting system based on blinded intermediaries is done. For the further possibility of the application of formal verification of protocols and dynamic analysis, it is necessary to make

an additional classification of semantic data, determine whether the keys belong to any party or parties, and also solve the problem with the returned values.

References

[1] Chaki S., Datta A. ASPIER: An automated framework for verifying security protocol implementations. In Proc. of the 22nd IEEE Computer Security Foundations Symposium, 2009, pp. 172-185.

[2] Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. Lecture Notes in Computer Science, vol. 3385, 2005, pp. 363-379.

[3] Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. Technical report, Laboratoire Spécification et Vérification, Report LSV-09-18, 2009.

[4] Jürjens J. Using interface specifications for verifying crypto-protocol implementations. In Proc. of the Workshop on foundations of interface technologies (FIT). 2008.

[5] Jürjens J. Automated security verification for crypto protocol implementations: Verifying the jessie project. Electronic Notes in Theoretical Computer Science, vol. 250, № 1, 2009, pp. 123-136.

[6] O'Shea N. Using Elyjah to analyse Java implementations of cryptographic protocols. In Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008). – 2008.

[7] Backes M., Maffei M., Unruh D. Computationally sound verification of source code. In Proc. of the 17th ACM conference on Computer and communications security, 2010, pp. 387-398.

[8] Bhargavan K. et al. Cryptographically verified implementations for TLS. In Proc. of the 15th ACM conference on Computer and communications security, 2008, pp. 459-468.

[9] Bhargavan K., Fournet C., Gordon A. D. Verified reference implementations of WS-Security protocols. Lecture Notes in Computer Science, vol. 4184, 2006, pp. 77-106.

[10] Bhargavan K. et al. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems, vol. 31, №. 1, 2008.

[11] Bhargavan K. et al. Verified implementations of the information card federated identity-management protocol. In Proc. of the 2008 ACM symposium on Information, computer and communications security, 2008, pp. 123-135.

[12] Bhargavan K. et al. Cryptographically verified implementations for TLS. In Proc. of the 15th ACM conference on Computer and communications security, 2008, pp. 459-468.

[13] Needham R. M., Schroeder M. D. Using encryption for authentication in large networks of computers. Communications of the ACM, vol. 21, №. 12, 1978. pp. 993-999.

[14] Capek P., Kral E., Senkerik R. Towards an empirical analysis of. NET framework and C# language features' adoption. In Proc. of the 2015 International Conference on Computational Science and Computational Intelligence (CSCI), 2015, pp. 865-866.

[15] Babenko L., Pisarev I. Distributed E-Voting System Based On Blind Intermediaries Using Homomorphic Encryption. In Proc. of the 11th International Conference on Security of Information and Networks, 2018.

[16] Viganò L. Automated security protocol analysis with the AVISPA tool. Electronic Notes in Theoretical Computer Science, vol. 155, № 12, 2006, pp. 61-86.

[17] Cremers C. J. F. The scyther tool: Verification, falsification, and analysis of security protocols. In Proc. of the International Conference on Computer Aided Verification, 2008, pp. 414-418.

[18] Küsters R., Truderung T. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In Proc. of the 22nd IEEE Computer Security Foundations Symposium, 2009, pp. 157-171.

Информация об авторах / Information about authors

Илья Александрович ПИСАРЕВ в настоящее время является аспирантом кафедры безопасности информационных технологий Южного федерального университета. Область научных интересов включает верификацию безопасности криптографических протоколов, проверки на моделях, анализ исходных кодов программ.

Ilya Aleksandrovich PISAREV is a graduate student at the Department of Information Technology Security at the Southern Federal University. The area of scientific interests includes verification of the security of cryptographic protocols, model checks, and analysis of program source codes.

Людмила Климентьевна БАБЕНКО является профессором кафедры безопасности информационных технологий Южного федерального университета. Область научных интересов включает криптографические методы и средства обеспечения информационной безопасности, технология параллельно-векторных вычислений, оценка стойкости криптографических методов защиты информации.

Liudmila Klimentevna BABENKO is currently a professor at the Department of Information Technology Security at the Southern Federal University. The area of scientific interests includes cryptographic methods and means of ensuring information security, technology of parallel-vector computing, evaluation of the strength of cryptographic methods of information protection.