Computer Assisted Methods in Engineering and Science
30(4): 389–406, 2023, doi: 10.24423/cames.869

Trivializing Verification of Cryptographic Protocols

Jacek PIĄTKOWSKI, Sabina SZYMONIAK*

Department of Computer Science, Czestochowa University of Technology, Czestochowa, Poland; e-mail: jacekp@icis.pcz.pl

* *Corresponding Author e-mail: sabina.szymoniak@icis.pcz.pl*

One of the main problems of the digital world is information security. Every second, people process millions of pieces of information that must be protected from unauthorized access. Cryptographic protocols that define the communication plan and the cryptographic techniques used to secure the messages come to the rescue. These protocols should also be regularly verified regarding their ability to protect systems from exposure to threats from the computer network. Bearing in mind the need to secure communication, verify the correct operation of security methods and process large amounts of numerical data, we decided to deal with the issues of modeling the execution of cryptographic protocols and their verification based on the CMMTree model. In this article, we present a tool that verifies a protocol's security. The tool allows for modelling a protocol and verifying that the path in the execution tree represents an attack on that protocol. The tool implements a specially defined hierarchy of protocol classes and a predicate that determines whether a node can be attached to a tree. We conducted a number of tests on well-known cryptographic protocols, which confirmed the correctness and effectiveness of our tool. The tool found the attack on the protocols or built an execution tree for them.

Keywords: security protocols verification, tree visualisation, hierarchical data structures.

1. Introduction

We can distinguish two significant aspects of the digital world. The first is information security. Users of various devices with Internet access send billions of bytes of data daily when communicating with other users or devices, shopping online or using electronic banking. The information exchanged can be simple messages without any value and messages containing valuable data such as authentication or sensitive user data. Each message sent this way is exposed to dishonest network users who aim to intercept data using various methods and

then use them. A dishonest user who attacks computer networks, users or applications is referred to as an Intruder. Due to the fraudulent activity of the Intruder, it is necessary to secure each processed information against unauthorized interception.

Information security, therefore, involves securing data against unauthorized access, use, disclosure, disruption or modification. Information sent during communication is secured using cryptographic protocols that define the communication plan and the cryptographic techniques used to secure the messages. Cryptographic protocols pursue various information security goals, including mutual or unilateral user authentication, reconciliation and distribution of session keys, and information exchange. Thus, the security of a cryptographic protocol is related to its purpose and cryptographic techniques that make it impossible to break it. The secure cryptographic protocol means that an Intruder will not compromise it; for example, he will fail to authenticate as an honest user to another protocol user.

Over the past few decades, many cryptographic protocols have emerged. These protocols pursued mentioned security goals. The most characteristic example of a cryptographic protocol is the Needham–Schroeder Public-Key (NSPK) protocol from [27], which pursued the purpose of mutual authentication. The Kerberos protocol from [28] ensures authentication and authorisation on a computer network using a key distribution centre. The DNP3 standard also includes solutions related to securing communication, and DNP3 contains a set of protocols used between components in process automation systems. Another example of security protocol is the Amelia protocol from [34]. This protocol ensures mutual user authentication and also protects users against false links.

Constantly appearing new threats to security make it necessary to verify the protocols. We must check if they provide an adequate level of security. Also, over the past few decades, various research teams dealt with the issue of protocol security, their modelling and verification. We can find the most interesting approaches in the works [3, 4, 10, 15, 31]. These studies often require the use of specialised tools and techniques to analyse risks and simulate potential attacks designed to breach the security of a protocol.

The second important aspect of the digital world is number processing. Each issue or information is ultimately after encoding a sequence of numbers. The numbers processing requires using coding tools to trivialise this non-trivial problem. During data processing, we often process their value and want to reduce the data to their value. After reducing data to a number, the data must be values.

Piątkowski in [29] proposed the Conditional Multiway Mapped Tree (CMMTree). The CMMTree enables the modelling and analysing problems described by the processed data. This model is not for data storage, and the implementation of CMMTree makes it possible to examine the paths of associations

of multi-path trees. We must remember the problem of encoding the input data, which arises before processing data. In this method, the authors use data encoding for numbers in the initial stage. They observed different types of data problems in the initial stage of processing, and they solved these problems by data dumping into "black boxes" of various data types.

Also, like all computer-based research, modelling and testing cryptographic protocols always require proper encoding of the input data. In this article, we decided to deal with the issues of modelling the execution of cryptographic protocols using the CMMTree model. We implemented a tool, including the CMMTree. We can model the protocol and check if there is a path representing the attack on it in the execution tree. We use the ProToc language from [13] to describe a protocol. We saved the protocol as a set of objects of a hierarchy of classes defined by us. Also, we accept the possibility that types may arise along the way that are combinations of existing types. Thanks to this, we can generate something that is almost trivial. We can represent a whole series of objects by a set of their addresses, which means they can be input into a tool that allows one data type. This paper's main contribution is the adaptation of CMMTree to the verification of cryptographic protocols.

Our main contribution is the implementation of the tool for automated security protocols verification realized with the following:

- adaptation of CMMTree to the verification of cryptographic protocols,
- preparation of a proper set of classes,
- supplementation the ProToc language specification with a set of initial knowledge,
- research conducted on the set of well-known security protocols using a developed tool.

The CMMTree builds a protocol execution tree. Then, it finds paths representing possible protocol attack(s). We check whether there is such a protocol course (composed of interleaving several executions) during which the Intruder can, for example, authenticate himself as another user or steal confidential data. We do not consider and evaluate the strength and unbreakability of cryptographic techniques such as ciphers or hash functions used in the protocol. If the Intruder intercepted a ciphertext, he can use other tools and try to break it using various methods.

The rest of the article is organised as follows. In Sec. 2, we discuss related works. Section 3 presents the process of modelling the cryptographic protocol. Here are described the subsequent stages related to preparing the protocol specification, loading it, creating protocol executions and preparing these executions to create a tree. Sections 4 and 5 include our experimental results and conclusions.

2. Related works

We can model the cryptographic protocol in several ways. One of them is the so-called specification languages. Such languages are formal definitions used to describe the system. It is worth mentioning here about such languages as CAPSL [24], HLPSL [8], ProToc [13], or SDL [14]. Each mentioned specification language can present a protocol as a sequence of steps using its security aspects.

CAPSL language allows for expressing protocol properties, assumptions and goals. This language does not specify user knowledge changes during the protocol execution. The HLPSL language is role-based. This language specifies changes in users' knowledge during the protocol execution. The user's role defines the abilities of knowledge changes. Unfortunately, the protocol specification in HLPSL is very complex. ProToc language allows the presentation of external and internal actions performed during the protocol execution. Also, this language specifies user knowledge changes during the protocol execution. The protocol specification is affordable and not too complex, like in the HLPSL language. SDL language allows the specification and description of distributed systems. Similarly, as HLPSL or ProToc, this language can specify activities performed during the protocol execution, but the protocol specification in HLPSL is very complex too. The main advantage of HLPSL, ProToc and SDL languages is their ability to specify the knowledge flow during the protocol execution. The main disadvantage of HLPSL and SDL languages is the specification complexity. The ProToc language can specify the same information in a few specification lines.

Another way of modelling protocols is CSP (Communicating Sequential Processes) [30]. In this approach, processes represent all actions performed under the cryptographic protocol. Petri Nets [6] were also used to model the protocols. In this case, agents with assigned roles were used to model the protocol. In turn, the roles relate to the previously defined knowledge-dependent behaviour. Both methods allow specifying activities performed during protocol execution. Unfortunately, protocol specifications in these methods are complex.

Trees are one of the most important abstract data types. They are used to represent arithmetic expressions [1, 19, 36], to manage the order of function calls [19], to represent knowledge and numerical prediction [37], in the sort, and search algorithms [2, 19, 25] and for modelling attacks on the cryptographic protocols too [18, 23]. In [18, 23], the authors suggested a general formal definition and semantic of the attack tree and its extension into attack–defense trees, including interactions between an attacker and a defender of a system.

Most often, to implement trees, we use the generic programming paradigm, and this paradigm makes it possible to create reusable software. This approach aims to obtain the most general and useful algorithm or data structure [12, 16, 33]. The work of Siek and Lumsdaine [33] is worth mentioning, the au-

thors compared languages supporting generic programming. They selected the following programming languages for the research: C++, SML, Haskell, Eiffel, Java, and C#, each with a subset of the Boost Graph Library. The authors concluded that none of the studied programming languages is ideal for generic programming.

It is also worth paying attention to the problems of analysing complex trees, several or tens of millions of elements and relations. The techniques of visualisation of large trees, as well as tree boosting, can be indicated here. We can identify the techniques of visualisation of large trees [21], and also tree boosting techniques for such analysis. The visualisation method suggested by Liang *et al.* in [21] provided adaptability to any enclosure-shaped containers. The second mentioned technique has found its application in machine learning in which it is necessary to analyse a large number of decision trees, and this process is time-consuming [22].

2.1. Motivations

Bearing in mind problems with communications security, we decided to prepare a tool for automatic cryptographic protocol verification using CMMTree. These protocols should be constantly verified because intrusion techniques continuously evolve. Methods and tools used for verification should be lightweight and fast so that the verification takes place in a short time with an appropriate load on the computing units. Preparing our tool, we first compared existing specification languages and chose ProToc because it can specify each protocol's features and is more lightweight than HLPSL or SDL language. Next, we considered different approaches to cryptographic protocol modelling and verification. We found that the approach proposed by Siedlecka-Lamch *et al.* in [31, 32] allows descriptions of each activity users perform during step execution, and also it is a lightweight method that is significant in the case of security analysis. We decided to implement mentioned approach as a predicate for CMMTree.

3. Cryptographic protocol modelling

This section will outline the following steps in cryptographic protocols modelling.

3.1. Preparation of the protocol specification

Modelling a cryptographic protocol is a very complex and difficult process. We must consider many actions that are carried out during the protocol execution. These are internal and external actions, such as:

- generating confidential information for the current session,
- message encryption and sending by the sender,
- receiving and decrypting messages by the recipient,
- knowledge acquisition [35].

Confidential information includes nonces[1)], timestamps and keys. In most well-known cryptographic protocols, generating one object occurs once for each honest user appearing in a single protocol session.

Encryption and decryption depend on the protocol structure and cryptography used in it. For asymmetric cryptography, we use a private and public key pair. We use the public key for encryption, while the private key is used for decryption. Private keys must not be sent in ciphertext or plaintext to avoid their interception. For symmetric cryptography, we share a symmetric key between two users. The users often agree upon the shared key before the session, using a trusted server. Some protocols are intended to establish a new session key (for example, the Wide-Mouth Frog protocol from [7]). During the execution of these protocols, the key is generated and then sent in a ciphertext.

Messages sent during the execution of the protocol may have different forms. They can be sent as plaintext, without encryption, or in an encrypted form, i.e., a ciphertext. Also, messages can be sent in a concatenated form that connects the ciphertext and plaintexts.

During the protocol modelling, we must check the user's decryption capabilities. The decryption process affects the ability of users to acquire knowledge. From each message, the user reads all objects sent as plaintexts. Also, he reads all message elements that can be extracted from the ciphertext while decrypting it. Sometimes, the user cannot decrypt the ciphertext, and the sender encrypts the ciphertext with a cryptographic key unknown to the recipient. In this case, the recipient acquires knowledge of the entire ciphertext. The process of acquiring knowledge has been extensively described in [35].

It is also worth noting that each protocol will have a set of publicly available objects. These will include user IDs and public keys. Also, each user will have initial knowledge. Users' initial knowledge includes their private, public and symmetric keys shared with other users. To model the protocol, we use the ProToc language [13]. We must indicate the objects needed to complete each step according to its structure.

We will show the issues in modelling cryptographic protocols on the example of the well-known Needham–Schroeder protocol [27]. We use the NSPK protocol as an example because this protocol is very suitable for presenting and testing solutions for cryptographic protocol analysis. The syntax for this protocol in Alice and Bob notation is as follows:

[1)] Nonce (N number used *once*) – large pseudo-random numbers.

$$\begin{aligned} \alpha_1 \quad & A \ \rightarrow \ B \ : \ \langle N_A \cdot i(A) \rangle_{K_B^+}, \\ \alpha_2 \quad & B \ \rightarrow \ A \ : \ \langle N_A \cdot N_B \rangle_{K_A^+}, \\ \alpha_3 \quad & A \ \rightarrow \ B \ : \ \langle N_B \rangle_{K_B^+}. \end{aligned} \tag{1}$$

Let us analyse the first step of this protocol. The sender is Alice (A) and the recipient is Bob (B). Alice wants to send Bob the following ciphertext $\langle N_A \cdot i(A) \rangle_{K_B^+}$, so she needs the following objects: N_A, $i(A)$ and K_B^+[2)]. The mentioned objects will be included in the set of needs of the first step. Identifier $i(A)$ is in the set of publicly available objects, just like the K_B^+ key. Nonce N_A must instead be generated. Now, Alice can compose the ciphertext and send it to Bob. Alice's knowledge collection at this stage has grown by N_A.

After receiving the ciphertext, Bob can decrypt it because he has the private key K_B^- in his knowledge. He reads objects $i(A)$ and N_A from it. Bob's knowledge collection thus increased by N_A.

We should consider other steps in the same way.

According to the structure of the ProToc language described in [13], we can prepare a specification of Needham–Schroeder protocols' steps. The protocol specification in Alice and Bob notation is presented in Table 1. We divided specification into the four data sets and the information column *step*.

Table 1. NSPK protocol specification.

Step	Users	Needs	Generated	Message
1.	A, B	$i(A)$, K_B^+, N_A	N_A	$\langle K_B^+, i(A) \vert N_A \rangle$
2.	B, A	N_A, N_B, K_A^+	N_B	$\langle K_A^+, N_A \vert N_B \rangle$
3.	A, B	N_B, K_B^+	–	$\langle K_B^+, N_B \rangle$

Using this specification, we get to know the following:

- sender and recipient at each step (*Users* column),
- set of objects from which the message is composed (*Needs* column),
- set of objects that must be generated (*Generated* column),
- sent message (*Message* column).

We supplemented the protocol specification in the ProToc language with a set of initial knowledge. This action was dictated by the need to standardise the storage of such information and the preparation of our predicate. For the NSPK protocol, initial users' knowledge contains four elements: A's and B's identifiers and public keys.

[2)] Designation K_B^+ means the public key of Bob.

3.2. Loading the protocol

When the file with protocol specification is ready, we can proceed with the protocol loading into the tool. During this, the tool validates the protocol. First, our tool checks the correctness of the definition saved in the file. In addition, the tool checks whether the objects appear in the appropriate sections. For example, an identifier object cannot appear in generated entities, and Identifiers are publicly available and do not need to be generated. Afterwards, a set of objects (of the designed class hierarchy) mapped a correctly defined and loaded protocol.

The protocol is a vector of steps, i.e., objects composed of indicators vectors. As mentioned earlier, the protocol step consists of four sets of information – four vectors. Each of these sets is a general vector-type content pointer, and each object in the vector contains properties specific to a particular data type. Therefore, the step has the Users, needs, generated objects and message vectors. The first vector can contain only Player type objects, i.e., objects representing protocol participants. Objects of this type cannot appear in other vectors. Also, the first vector can only contain two entities representing the sender and receiver in a given step. Only nonces and keys can appear in the third vector. The last vector contains one object representing the message sent during protocol execution.

The structures loaded from the file are mapped to appropriate objects by the designed class hierarchy. Figure 1 shows the division of loaded structures into objects for the Needham–Schroeder protocol.

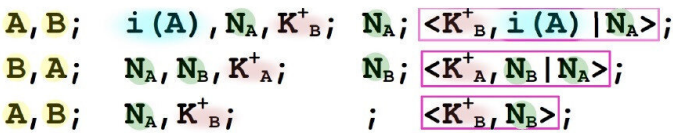

FIG. 1. The division of loaded structures into objects for the Needham–Schroeder protocol.

It is possible to observe five basic types of data here. As mentioned, there are always two Player objects in the first vector. The second vector contains the objects needed to compose a message. In the case of the first step of the Needham–Schroeder protocol, these are objects of type: identifier, nonce and asymmetric key. The third vector for the first and second steps contains one nonce object. The last vector includes a vector of Message type objects, and the message type has objects of the asymmetric ciphertext type.

3.3. Protocol executions

Then, we process a correctly loaded protocol. To learn the specifics and capabilities of the cryptographic protocol, we should generate all its possible execu-

tions. Executions are scenarios according to which the protocol can be executed, and they differ in two elements. First is the order in which users appear as the sender and recipient. The protocol can be started by Alice, as well as by Bob. Also, when considering the protocol, we should consider a dishonest user called an Intruder.

The Intruder is a user whose goal is to break the protocol and capture the confidential information of honest users. The Intruder's activities depend on the used model. These can be the following models: Dolev–Yao [11], lazy Intruder [17, 26], limited Dolev–Yao model [20] and limited lazy Intruder [20].

In the Dolev–Yao model, an Intruder can unlimitedly control the network. The Intruder has access to all transmitted messages, which may be intercepted, blocked, processed and sent by him contrary to the protocol. It is worth noting that the Intruder will only have access to the information contained in the ciphertext if he knows the appropriate decryption key. We can compare the lazy Intruder model to malware. For this model, the Intruder can only send the entire message without the option of modifying them. This solution limits the constructed space to messages that an honest user sends. Also, we can indicate models of limited Dolev–Yao and limited lazy Intruder. In both cases, the Intruder's capabilities are limited to using messages sent directly to him (when he impersonates an honest user). We assumed that an Intruder does not generate nonce during protocol execution, and the Intruder has a pool of nonces (generated by him before the start of protocol execution).

The second element that distinguishes the executions is the objects used by the Intruder when he communicates with honest users. During the execution of the protocol, the Intruder can act as a regular user and impersonate other users. When the Intruder is a regular user, he uses his identity, nonces and keys during communication. In the second case, the Intruder uses the identity of the impersonating user, his nonce and keys. Also, he can use his nonces and keys.

Knowing the set of protocol participants (Alice, Bob, Trudy[3)]) and the set of confidential information of these users, we can generate a set of all possible protocol executions. This operation is performed based on a function that produces non-repetitive variations for users and cryptographic objects. The tool modifies protocol steps according to the generated variations for users and objects.

Table 2 presents a set of executions for the Needham–Schroeder protocol. We divided the table symmetrically into two parts. The first one captures the communication between Alice (A) and Bob (B). In contrast, the second captures the communication between Bob and Alice. Both parts consider Trudy's occurrence (T).

[3)]Intruder in Alice and Bob notation.

Table 2. The summary of NSPK protocol executions.

No.	Parts	Parameters	No.	Parts	Parameters
1	$A \to B$		10	$B \to A$	
2	$T \to B$	N_T, K_T	11	$T \to A$	N_T, K_T
3	$T \to B$	N_A, K_T	12	$T \to A$	N_B, K_T
4	$T(A) \to B$	N_T, K_A	13	$T(B) \to A$	N_T, K_B
5	$T(A) \to B$	N_A, K_A	14	$T(B) \to A$	N_B, K_B
6	$A \to T$	N_T, K_T	15	$B \to T$	N_T, K_T
7	$A \to T$	N_B, K_T	16	$B \to T$	N_A, K_T
8	$A \to T(B)$	N_T, K_B	17	$B \to T(A)$	N_T, K_A
9	$A \to T(B)$	N_B, K_B	18	$B \to T(A)$	N_A, K_A

Column *No.* contains a serial number assigned to the exercise to simplify reference to it during analysis. The *Parts* column lists the participants in the protocol in the order in which they appear during the protocol. $T(A)$ means an Intruder impersonating Alice. The notation $T(B)$ means an Intruder impersonating Bob. The *Parameters* column contains a cryptographic object that the Intruder uses during execution. For example, execution number 5 is between Trudy (who impersonates Alice) and Bob. During this execution, Trudy uses Alice's nonce and her public key.

3.4. Set of information

The next step in preparing the protocol to model it as a tree is generating a set of information. This procedure implements to methodology mentioned in [20, 31, 32]. This structure reflects the possibilities of executing a given step. Each set of information contains the following information:

- sender and recipient of the step,
- execution number and step number,
- set of objects that the sender needs to complete the step,
- set of objects that the sender must generate,
- set of objects that the recipient will learn after completing the step.

The structure of the information set is based on the user's initial knowledge set and a set of publicly available objects. Each set contains only the information necessary to build a tree; other information is not included. The content of information sets in a given step depends on the structure of the protocol.

To understand the essence of creating this set of information, let us analyse execution No. 14. The specification for this execution in Alice and Bob notation is as follows:

$$\begin{aligned} &14.1. \quad T(B) \ \rightarrow \ A \ : \ \langle N_B \cdot i(A) \rangle_{K_A^+}, \\ &14.2. \quad A \ \rightarrow \ T(B) \ : \ \langle N_B \cdot N_A \rangle_{K_B^+}, \\ &14.3. \quad T(B) \ \rightarrow \ A \ : \ \langle N_A \rangle_{K_A^+}. \end{aligned} \tag{2}$$

In execution No. 14, Trudy impersonates Bob and communicates with Alice. During this execution, Trudy uses Bob's nonce and the public key.

To execute step 14.1, Trudy needs to make a ciphertext $\langle N_B \cdot i(A) \rangle_{K_A^+}$. The key K_A^+ and the identifier $i(A)$ are publicly available.

One way to do this is when Trudy will have the entire ciphertext obtained from another execution. In this situation, the sender's set of needs will be the ciphertext $\langle N_B \cdot I(A) \rangle_{K_A^+}$, the set of generated objects will be empty, and the set of the recipient's knowledge will contain nonce N_B because the recipient can decrypt the transmitted ciphertext. For this capability, the set of information will look as follows (semicolons separate elements):

$$T(B) \rightarrow A \ ; \ [14.1] \ ; \ \langle N_B \cdot i(A) \rangle_{K_A^+}; \ ; N_B. \tag{3}$$

The second option to execute step 14.1 is to create the entire ciphertext after acquiring knowledge of the nonce N_B by establishing communication with Bob. In this case, the sender's set of needs will contain only N_B, the generated objects will be empty, and the recipient's knowledge will also include nonce N_B. The set of information for this capability will be as follows:

$$T(B) \rightarrow A \ ; \ [14.1] \ ; \ N_B; \ ; N_B. \tag{4}$$

The rest of the steps should be the same way. Table 3 presents information for all steps in the 14th execution.

Table 3. The summary of a set of information for fourteen execution.

Parts	Execution & step	Needs	Generated	Knowledge
$T(B) \rightarrow A$	[14.1]	$\langle N_B \cdot i(A) \rangle_{K_A^+}$		N_B
$T(B) \rightarrow A$	[14.1]	N_B		N_B
$A \rightarrow T(B)$	[14.2]	N_B	N_A	$\langle N_B \cdot N_A \rangle_{K_B^+}$
$T(B) \rightarrow A$	[14.3]	$\langle N_B \cdot N_A \rangle_{K_B^+}$		N_A
$T(B) \rightarrow A$	[14.3]	N_A		N_A

It is worth paying attention to step 14.2. In this step, Trudy receives a ciphertext that he cannot decrypt. In this situation, the user's knowledge increases the ciphertext without hulling its components.

3.5. Tree of the protocol executions

We build the tree of the protocol executions on the CMMTree model base mentioned in [29]. This model separates data from their structure. CMMTree includes three main principles. The first is the predicate that defines the conditions for joining the nodes of the created tree. The second rule allows operating on different data types, and it is possible to use those data types that combine existing input types. The third rule defines the simplicity of changing the rules of connecting nodes without interfering with the code implementing the tree.

A triple describes the logical model of CMMTree:

$$\text{CMMTree} = (D, p, T), \tag{5}$$

where

- $D = \{d_1, ..., sd_x\}$ is a set of a unique data values,
- $T = \{v_i : 0 \leqslant i < n\}$ is the tree-structure of the data dependencies,
- $p : D_k \times T \to \{true, false\}$, $(k = 1, ..., x)$ is a predicate defining the rules of connecting nodes of T.

Figure 2 presents the schema of the CMMTree logical model. There is a schema for data of the same type in Fig. 2a. In Fig. 2b, there is a schema for different data types belonging to a defined class hierarchy.

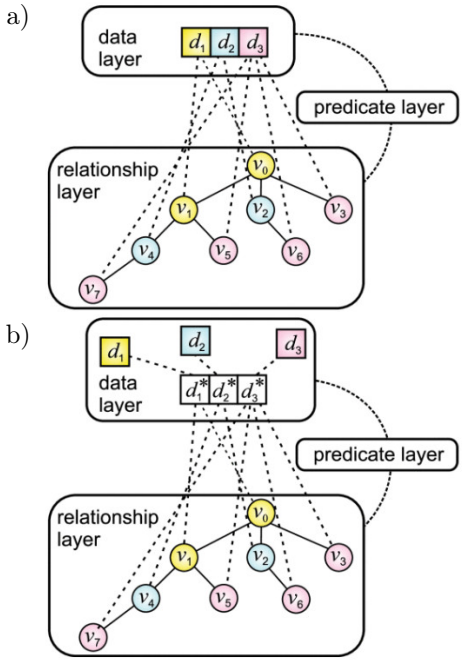

Fig. 2. The schema of the CMMTree logical model: a) for data of the same type, b) for different types of data belonging to a defined class hierarchy [29].

The values in dataset D should be unique. However, they can be represented multiple times in the T tree structure. The nodes v_i do not store the values of the processed data. They contain references to d_x. For example, in Fig. 2, the nodes v_0 and v_1 represent the value d_1 of the set D. On the other hand, nodes v_3, v_5, v_6 and v_7 represent the value d_3. We do not require the processed data to be of the same type. In Fig. 2b, d_1, d_2, and d_3 represent three unique values and three different types of objects. It is enough for these types to belong to a specific class hierarchy. Then, such data can be represented by a vector of pointers.

The function p defines the conditions for joining each T node. This action is crucial for the final result of processing dataset D. A positive or negative decision to attach another node is made based on the information provided by the candidate to the descendant of the current node. Information represented by the other nodes of the T tree can be made based on the relationship between the two objects (the current node and the candidate to join, as well as the knowledge represented by a larger group of nodes (nodes on the path or sibling of the current node and other combinations of nodes). In addition to the values of individual d_x elements, it is also possible to use the knowledge of their real type. The p function will be able to behave polymorphically at runtime. The discussed solution gives almost unlimited possibilities for data analysis.

The predicate has to address three significant issues while modelling the execution of security protocols. The first is the order of the steps. As the first node of the tree, we can add only the node relating to the first step of the protocol. Next, we can add steps from the already-started execution or the first one from the new execution. The second issue is users' knowledge. At each stage of building a tree, the sender of the attached step must have the appropriate knowledge to perform this step. The last issue concerns the cryptographic objects generated by an honest protocol user. In this case, the number of fair users generated objects must be, at most, the number of generated objects defined in the protocol specification header.

We prepared a tree-building predicate based on the possibility of an attack by an Intruder for cryptographic protocols. The node representing the protocol execution step can be attached to the tree if the sequence of steps is correct. The nodes from one execution may only be attached to the tree in order other than dictated by the protocol step numbering.

Also, the knowledge of the protocol's participants plays a significant role in attaching the steps to the tree. Attempting to attach another node to the tree must involve checking the knowledge of each protocol participant and the possibility of performing the given step. If the sender's knowledge set contains objects necessary for execution, such a node can be attached to the tree. Additionally, the sender's knowledge should be increased if he has generated any objects,

and the recipient's knowledge should be increased by the objects obtained from the received message.

It is necessary to remember the number of nonces generated by honest users. As mentioned, each honest user generates one nonce during the protocol only. Thus, the tree cannot attach a node that refers to the step where the same honest user will have to generate a second nonce on the path.

4. Experimental results

We used the same computer unit with an Intel Core i7 processor, 16 GB main memory and Ubuntu Linux operating system for our research. If the attack upon the protocol exists, the CMMTree will find it. We do not signify an attack type, but generally found attacks aimed at security features such as confidentiality, integrity, authentication and non-repudiation. We will present the experimental results on the examples of the Needham–Schroeder protocol and Amelia protocol from [34]. For the NSPK protocol, the tool prepared a set of 18 executions described in Table 2. For the Amelia protocol, the tool prepared a set of 36 executions.

First, we compare verification times for three tools: our tool, the ProVerif tool described in [5] and Tamarin described in [9]. We observed that our tool verified mentioned protocols faster than the ProVerif and Tamarin tools (Table 4).

Table 4. Comparison of verification times.

Tool	NSPK protocol [ms]	Amelia protocol [s]
ProVerif	17	26
Tamarin	40	117
Our tool	3.79	12.08

Next, we execute our research. During building the tree, our tool found two attacks on Needham–Schroeder protocol: Lowe's attack and the Man-in-the-Middle attack. The syntax of Lowe's attack in Alice and Bob notation is as follows:

$$\begin{aligned}
&\alpha_1 \quad A \quad \to T : \langle N_A \cdot i(A)\rangle_{K_T^+},\\
&\beta_1 \quad T(A) \to B : \langle N_A \cdot i(A)\rangle_{K_B^+},\\
&\beta_2 \quad B \quad \to T(A) : \langle N_A \cdot N_B\rangle_{K_A^+},\\
&\alpha_2 \quad T \quad \to A : \langle N_A \cdot N_B\rangle_{K_A^+},\\
&\alpha_3 \quad A \quad \to T : \langle N_B\rangle_{K_T^+},\\
&\beta_3 \quad T(A) \to B : \langle N_B\rangle_{K_B^+}.
\end{aligned} \tag{6}$$

The syntax of the Man-in-the-Middle attack in Alice and Bob notation is as follows:

$$\begin{aligned}
\alpha_1 \quad & A \quad \rightarrow \; T(B) \; : \; \langle N_A \cdot i(A) \rangle_{K_B^+}, \\
\beta_1 \quad & T(A) \; \rightarrow \; B \; : \; \langle N_A \cdot i(A) \rangle_{K_B^+}, \\
\beta_2 \quad & B \quad \rightarrow \; T(A) \; : \; \langle N_A \cdot N_B \rangle_{K_A^+}, \\
\alpha_2 \quad & T(B) \; \rightarrow \; A \; : \; \langle N_A \cdot N_B \rangle_{K_A^+}, \\
\alpha_3 \quad & A \quad \rightarrow \; T(B) \; : \; \langle N_B \rangle_{K_B^+}, \\
\beta_3 \quad & T(A) \; \rightarrow \; B \; : \; \langle N_B \rangle_{K_B^+}.
\end{aligned} \tag{7}$$

Each attack consisted of six steps from different executions (interlacing the two executions). During Lowe's attack, execution no. 3 and 18 were interlaced, and during the Man-in-the-Middle attack, execution no. 5 and 18 were interlaced.

Table 5 shows the summary of the tree shape built for the NSPK protocol. We included here a number of nodes (NoN row), the number of firstborns (NoF row), and the number of leaves (NoL row) on each built tree level. The time of the CMMTree calculation was equal to 3.79 ms.

Table 5. Results for tree building for NSPK protocol.

Level	0	1	2	3	4	5	6	7
NoN	1	5	43	258	175	187	123	62
NoF	1	1	5	43	123	141	99	62
NoL	0	0	0	135	34	88	61	62

During building the tree, our tool did not find attacks on Amelia protocol. Table 6 shows the summary of the tree shape created for the Amelia protocol. The time of the CMMTree calculation was equal to 12.08 s.

Table 6. Results for tree building for the Amelia protocol.

Level	0	1	2	3	4	5	6	7	8	9	10
NoN	1	10	150	1230	3040	5860	11490	22350	41250	67350	91350
NoF	1	1	10	150	1230	3040	5850	11280	21750	41250	67350
NoL	0	0	0	0	0	10	210	600	0	0	7350

5. Conclusions

This paper has adopted the CMMTree to verify cryptographic protocols. CMMTree makes it possible to build tree structures that reflect relationships

between the input data elements. The mentioned model can operate on different types of data. Also, it can quickly identify the characteristic places that determine the shape of trees (including extensive trees with tens of millions of nodes).

The main element of this adaptation is a specially constructed predicate. Our predicate contains rules describing adding a new node into the tree according to protocol structure. Thanks to this predicate, we built a tree of protocol executions. The predicate uses an approach from [31, 32].

We examined this predicate using selected protocols. Implemented CMMTree found an attack on this protocol. For some protocols, the implemented CMMTree has found an attack. For the other protocols, CMMTree has built its tree of execution trees. We compared our results with the ProVerif and Tamarin tools, and the results are promising.

In future work, we will continue testing our Conditional Multiway Mapped Tree to verify more complex problems.

Acknowledgements

The project was financed under the program of the Polish Minister of Science and Higher Education under the name "Regional Initiative of Excellence" in the years 2019–2023, project number 020/RID/2018/19, and the amount of financing PLN 12 000 000.

References

1. A.V. Aho, J.D. Ullman, *Foundations of Computer Science*, 1st ed., W.H. Freeman & Co., USA, 1994.
2. G. Barnett, L. Del Tongo, *Data Structures and Algorithms: Annotated Reference with Examples*, DotNetSlackers, 2008.
3. D.A. Basin, C. Cremers, C.A. Meadows, Model checking security protocols, [in:] *Handbook of Model Checking*, E. Clarke, T. Henzinger, H. Veith, R. Bloem [Eds], pp. 727–762, Springer, Cham, 2018, doi: 10.1007/978-3-319-10575-8_22.
4. B. Blanchet, Modeling and verifying security protocols with the applied pi calculus and ProVerif, *Foundations and Trends in Privacy and Security*, **1**(1–2): 1–135, 2016, doi: 10.1561/3300000004.
5. B. Blanchet, V. Cheval, V. Cortier, ProVerif with lemmas, induction, fast subsumption, and much more, [in:] *IEEE Symposium on Security and Privacy (S&P'22)*, pp. 205–222, IEEE Computer Society, San Francisco, CA, 2022, https://hal.inria.fr/hal-03366962/.
6. R. Bouroulet, R. Devillers, H. Klaudel, E. Pelz, F. Pommereau, Modeling and analysis of security protocols using role based specifications and Petri nets, [in:] K.M. van Hee, R. Valk [Eds], *Applications and Theory of Petri Nets*, pp. 72–91, Springer, Berlin, Heidelberg, 2008.
7. M. Burrows, M. Abadi, R. Needham, A logic of authentication, *ACM Transactions on Computer Systems*, **8**(1): 18–36, 1990, doi: 10.1145/77648.77649.

8. Y. Chevalier *et al.*, A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols, [in:] *Workshop on Specification and Automated Processing of Security Requirements – SAPS'2004*, pp. 13, Austrian Computer Society, Linz, Austria, 2004.
9. V. Cortier, S. Delaune, J. Dreier, Automatic generation of sources lemmas in tamarin: Towards automatic proofs of security protocols, [in:] L. Chen, N. Li, K. Liang, S. Schneider [Eds], *Computer Security – ESORICS 2020*, pp. 3–22, Springer, Cham, 2020.
10. A. David, K.G. Larsen, A. Legay, M. Mikuăionis, D.B. Poulsen, UPPAAL SMC tutorial, *International Journal on Software Tools for Technology Transfer*, **17**(4): 397–415, 2015, doi: 10.1007/s10009-014-0361-y.
11. D. Dolev, A.C. Yao, On the security of public key protocols, [in:] *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, *SFCS '81*, pp. 350–357, IEEE Computer Society, Washington, DC, USA, 1981.
12. D. Gregor, J. Järvi, J. Siek, G. Reis, B. Stroustrup, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, *ACM SIGPLAN Notices*, **41**(10): 291–310, 2006, doi: 10.1145/1167515.1167499.
13. A. Grosser, M. Kurkowski, J. Piątkowski, S. Szymoniak, ProToc – An universal language for security protocols specifications, [in:] A. Wilinski, I.E. Fray, J. Pejas [Eds], *Soft Computing in Computer and Information Science. Advances in Intelligent Systems and Computing*, Vol. 342, pp. 237–248, Springer, Cham, 2014, doi: 10.1007/978-3-319-15147-2_20.
14. D. Hercog, *Communication Protocols. Principles, Methods and Specifications*, Springer, 2020, doi: 10.1007/978-3-030-50405-2.
15. A. Hess, S. Mödersheim, A typing result for stateful protocols, [in:] *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 374–388, IEEE, 2018, doi: 10.1109/CSF.2018.00034.
16. J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming – Challenges of constrained generics in C++, *ACM SIGPLAN Notices*, **41**(6): 272–282, 2006, doi: 10.1145/1133255.1134014.
17. A. Kassem, P. Lafourcade, Y. Lakhnech, S. Mödersheim, Multiple independent lazy intruders, [in:] *1st Workshop on Hot Issues in Security Principles and Trust (HotSpot 2013)*, 15 pages, 2013, http://www.cs.bham.ac.uk/∼mdr/research/projects/HotSpot-2013/papers/paper%2012.pdf.
18. B. Kordy, S. Mauw, S. Radomirović, P. Schweitzer, Foundations of attack–defense trees, [in:] P. Degano, S. Etalle, J. Guttman [Eds], *International Workshop on Formal Aspects in Security and Trust*, FAST 2010. Lecture Notes in Computer Science, Vol. 6561, pp. 80–95, Springer, Berlin, Heidelberg, 2010, doi: 10.1007/978-3-642-19751-2_6.
19. R.L. Kruse, A.J. Ryba, *Data Structures and Program Design in C++*, Prentice-Hall, USA, 1998.
20. M. Kurkowski, *Formalne metody weryfikacji własności protokołów zabezpieczających w sieciach komputerowych* [in Polish], Akademicka Oficyna Wydawnicza Exit, Warszawa, 2013.
21. J. Liang, Q. Nguyen, S. Simoff, M. Huang, Divide and conquer treemaps: Visualizing large trees with various shapes, *Journal of Visual Languages & Computing*, **31**: 104–127, 2015, doi: 10.1016/j.jvlc.2015.10.009.
22. S. Liu, T. Xiao, J. Liu, X. Wang, J. Wu, J. Zhu, Visual diagnosis of tree boosting methods, *IEEE Transactions on Visualization and Computer Graphics*, **24**(1): 163–173, 2017, doi: 10.1109/TVCG.2017.2744378.

23. S. Mauw, M. Oostdijk, Foundations of attack trees, [in:] *International Conference on Information Security and Cryptology*, pp. 186–198, Springer, 2005, doi: 10.1007/11734727_17.

24. J.K. Millen, CAPSL: Common authentication protocol specification language, [in:] *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, 1996, doi: 10.1145/304851.304879.

25. P. Morin, *Open Data Structures (in C++)*, 2013, https://opendatastructures.org/.

26. S. Mödersheim, F. Nielson, H.R. Nielson, Lazy mobile intruders, [in:] D.A. Basin, J.C. Mitchell, [Eds], *POST*, *Lecture Notes in Computer Science*, Vol. 7796, pp. 147–166, Springer, 2013.

27. R.M. Needham, M.D. Schroeder, Using encryption for authentication in large networks of computers, *Communications of the ACM*, **21**(12): 993–999, 1978, doi: 10.1145/359657.359659.

28. B.C. Neuman, T. Ts'o, Kerberos: An authentication service for computer networks, *IEEE Communications Magazine*, **32**(9): 33–38, 1994, doi: 10.1109/35.312841.

29. J. Piątkowski, The conditional multiway mapped tree: Modeling and analysis of hierarchical data dependencies, *IEEE Access*, **8**: 74083–74092, 2020, doi: 10.1109/ACCESS.2020.2988358.

30. P.Y.A. Ryan, S.A. Schneider, M.H. Goldsmith, G. Lowe, A.W. Roscoe, *The Modelling and Analysis of Security Protocols: The CSP Approach*, 1st ed., Addison-Wesley Professional, Harlow, London, 2000.

31. O. Siedlecka-Lamch, S. Szymoniak, M. Kurkowski, A fast method for security protocols verification, [in:] *Computer Information Systems and Industrial Management: Proceedings of the 18th International Conference, CISIM 2019*, Belgrade, Serbia, September 19–21, 2019, pp. 523–534, 2019, doi: 10.1007/978-3-030-28957-7_43.

32. O. Siedlecka-Lamch, S. Szymoniak, M. Kurkowski, I.E. Fray, Towards most efficient method for untimed security protocols verification, [in:] *24th Pacific Asia Conference on Information Systems, PACIS 2020 Proceedings*, Dubai, UAE, June 22–24, 2020, p. 189, 2020, https://aisel.aisnet.org/pacis2020/189/.

33. J.G. Siek, A. Lumsdaine, A language for generic programming in the large, *Science of Computer Programming*, **76**(5): 423–465, 2011, doi: 10.1016/j.scico.2008.09.009.

34. S. Szymoniak, Amelia—A new security protocol for protection against false links, *Computer Communications*, **179**: 73–81, 2021, doi: 10.1016/j.comcom.2021.07.030.

35. S. Szymoniak, M. Kurkowski, J. Piątkowski, Timed models of security protocols including delays in the network, *Journal of Applied Mathematics and Computational Mechanics*, **14**(3): 127–139, 2015 doi: 10.17512/jamcm.2015.3.14.

36. J.-P. Tremblay, P.G. Sorenson, *An Introduction to Data Structures with Applications*, 2nd ed., Computer Science Series, McGraw-Hill, Auckland, 1984.

37. I.H. Witten, E. Frank, M.A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed., Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Amsterdam, 2011.

Received September 22, 2022; revised version November 30, 2022; accepted December 7, 2022.