applied sciences

MDPI

Article

Methodology of Testing the Security of Cryptographic Protocols Using the CMMTree Framework

Jacek Piątkowski [†] **and Sabina Szymoniak** [*,†]

Department of Computer Science, Częstochowa University of Technology, 42-200 Częstochowa, Poland; jacek.piatkowski@icis.pcz.pl

* Correspondence: sabina.szymoniak@icis.pcz.pl

† These authors contributed equally to this work.

Abstract: Internet communication is one of the significant aspects of modern civilization. People use banking, health, social, or shopping platforms and send a lot of data. Each communication should be secured and protected against dishonest users' activities during its transfer via network links. Cryptographic protocols provide such security and protection. Because of the evolution of the vulnerabilities and attackers' methods, the cryptographic protocols should be regularly verified. This paper presents a methodology for testing the security of cryptographic protocols using the CMMTree framework. We developed and adapted a software package for analyzing cryptographic protocols regarding compatibility with the CMMTree framework using a predicate according to the approach described in Siedlecka-Lamch et al.'s works. We optimized and strengthened the mentioned approach with tree optimization methods and a lexicographic sort rule. Next, we researched the well-known security protocols using a developed tool and compared and verified the results using sorted and shuffled data. This work produced promising results. No attacks on the tested protocols were discovered.

Keywords: Conditional Multiway Mapped Tree; security protocols; hierarchical data dependency analysis; verification

Citation: Piątkowski, J.; Szymoniak, S. Methodology of Testing the Security of Cryptographic Protocols Using the CMMTree Framework. *Appl. Sci.* **2023**, *13*, 12668. https://doi.org/10.3390/app132312668

Academic Editors: Frederico Branco, José Martins and Henrique Mamede

Received: 16 October 2023
Revised: 16 November 2023
Accepted: 23 November 2023
Published: 25 November 2023

1. Introduction

One of the significant aspects of modern civilization is Internet communication. People worldwide send vast amounts of information over the Internet at any given moment. Computer network users use banking, health, social, and shopping platforms and communicate via instant messaging. We can also observe the rapid development of communication between devices that make up the Internet of Things or Wireless Sensor Networks. Such networks connect various devices and sensors to exchange information between the devices and between them and other networks, nodes used to process and store data [1–3].

First, some honest users want to exchange data with other users or network nodes. However, on the web, we can meet rogue users called Intruders. The activity of an Intruder in the network depends on his imagination, needs and abilities. For example, an attacker might intercept messages sent by honest users, attempt to read, modify, and forward them, or intercept communications and retransmit messages to trick other network users. We should always be concerned that online activity can lead honest users to many losses, such as data loss and identity or material damage if an Intruder acquires credentials with a bank [4].

For this reason, securing communication between users plays an essential and pivotal role in exchanging messages. The communication course is supervised by complex communication protocols, constituting a message exchange sequence between users. Nevertheless, security protocols are used to secure essential parts of such communication. Such protocols may be one-way, mutual authentication, or establishing and exchanging a session key [5,6].

In recent decades, numerous security protocols have been put out (see [7–13]). Each of them has been thoroughly tested regarding the level of security that a given protocol should provide. The provided studies suggested that the protocols could be broken in many cases (for example, Ref. [7]). The broken protocols have some vulnerabilities, thanks to which an Intruder can intercept and use data transferred between users, modify them, and transmit them again.

The problem of potential attacks by dishonest users is related to verifying protocol security levels constantly. With this need in mind, we present a security protocol verification tool in this article. Primarily, this tool uses the Conditional Multiway Mapped Tree (CMMTree) methodology described in [14]. CMMTree is the primary framework that defines the structure and mechanism of operation of all the software we have created. The solution allows building trees reflecting various relationships between the processed data.

Moreover, the implemented tool considers the approach proposed by Siedlecka et al. in [15,16], enabling the modelling of information about the security protocol execution step. In the proposed approach, a single statement about the protocol execution step describes all actions performed by users during the protocol step (generating confidential information, knowledge acquisition) and the conditions for executing the protocol step (knowledge needs of the sender). This approach is advantageous and practical because, thanks to it, we can model each piece of information necessary to find an attack upon the protocol. Therefore, we decided to strengthen the mentioned approach with tree optimization methods (pruners) and a lexicographic sort rule and implement an automated security protocol verification tool according to the CMMTree methodology.

The main contributions of our article are as follows:

- development and adaptation of a software package for the analysis of cryptographic protocols in terms of compatibility with the CMMTree framework [14];
- definition and implementation of the predicate for CMMTree according to the approach described in [15,16];
- strengthening of the mentioned approach with tree optimization methods (pruners) and a lexicographic sort rule;
- research conducted on the set of well-known security protocols using a developed tool;
- comparison and verification of research using sorted and shuffled data.

The rest of this article is organized as follows. Section 2 will present the related works associated with verifying the security protocols. We will consider several methods and tools used for the mentioned issue. Section 3 will introduce an example of a security protocol: the Needham Schroeder Public Key (NSPK) protocol [7]. The NSPK protocol is the best example for research concerned with security protocol verification. In Section 4, we will present the implemented methods. We will describe how CMMTree works and how we build the predicate for CMMTree. Section 5 will present the obtained results. We will present a parametric description of the execution tree shapes for the tested protocol. Also, we will describe the algorithms of the predicate function. In the last section, we will consider our conclusions and future work.

2. Related Works

Verifying the communication and security protocols is essential in securing network communication. Each network and used protocol are exposed to many attacks, for example, an impersonation attack during which the dishonest network user uses the identities of other network users. The choice of a security protocol depends on the implementation of the network and its purpose. Networks, like the Internet of Things or Wireless Sensor Networks, require secure authentication to protect the network against network node capture attacks. They use one protocol type. The vast area networks, where each stage of communication requires an appropriate level of security, will use different kinds of protocols (for example, communication, authentication, or key distribution) [1,3,17].

So far, many methods of verifying cryptographic protocols have been proposed. Among them, some approaches use timed automata. Thanks to these methods, it is possible

to demonstrate time's influence on communication security. In [18], Ali proposed a method translation attack tree into timed automata. The author can verify security protocol vulnerabilities using a set network of timed automata. Also, Andre et al. in [19] proposed attack tree models. Moreover, in [20], Siedlecka-Lamch employed probabilistic timed automata for security protocol verification. The author tried to find the probability of breaking the encryption key.

To verify the security of the protocol, we can also use formal proofs, e.g., BAN logic [21], Real-Or-Random (ROR) [22], Random Oracle Model (ROM) [23], or Syverson–Van Oorschot (SVO) [24]. However, these proofs are very complex, so for verifying protocols, we mostly used the tools that automatically implement the described methods for verifying protocols (Scyther [25,26], Tamarin [27,28], ProVerif [29–31], AVISPA [32], or the tool mentioned in [5]).

It is worth pointing out that Siedlecka-Lamch et al. proposed an interesting security protocol modelling and verification method in [15,16]. Authors consider the protocol step a tuple that includes information about each cryptographic primitive and users' activities. The tuple consists of information about:

- execution and step;
- sender and receiver;
- sender's needs, i.e., those cryptographic primitives that the sender should know before step execution;
- cryptographic primitives that should be generated before step execution;
- receiver knowledge, i.e., those cryptographic primitives the receiver learns during step execution.

The included information is enough for step definition because it describes each activity users perform during step execution. Also, they offer conditions and restrictions for step execution. Thanks to them, it is possible to create a protocol execution tree (ET) and find if there is any attack upon the security protocol. We observed that building a tree of each possible security protocol execution using CMMTree lets us thoroughly check all possible combinations of steps of these executions. For this reason, we decided to use Siedlecka-Lamch et al.'s approach as a predicate for CMMTree and find if the CMMTree model is suitable for security protocol verification.

3. Needham Schroeder Public Key Protocol

In recent decades, numerous communication protocols have been proposed and developed. Security or cryptographic protocols are part of large and complex communication protocols that aim to authenticate, establish, or exchange symmetric keys. The most known and tested security protocol is probably the Needham Schroeder Public Key protocol [7]. It is well-known that its original version is not a perfect protocol, and it can be broken. Nevertheless, in our opinion, it is very useful. It is like the fruit flies used in countless studies by biologists, geneticists, and other scientists.

This protocol is the best research example because it shows the main ideas of security protocols. It allows tracking the process and progress of protocol breaking, which is very useful in testing solutions for protocol analysis.

The NSPK protocol aims for mutual authentication with public keys.

$$\begin{array}{lll} \alpha_1 & A \to B: & \langle N_A, i(A)\rangle_{K_B}, \\ \alpha_2 & B \to A: & \langle N_A, N_B\rangle_{K_A}, \\ \alpha_3 & A \to B: & \langle N_B\rangle_{K_B}. \end{array}$$

In this notation:

- $\langle M\rangle_K$ means the message M encrypted by key K, for example, $\langle N_A, i(A)\rangle_{K_B}$ means the message composed by nonce N_A and identifier $i(A)$ encrypted by key K_B.
- A and B mean the honest protocol participants.
- $i(u)$ means the text identifier of the user u, for example, $i(A)$ means user A's identifier.

- N_u means the nonce generated by user u, for example, $N(A)$ means the nonce generated by user A.
- K_u means a public key of user u, for example, K_A means a public key of user A.

During this protocol, users A and B authenticate to each other. Authentication takes place with the use of users' nonces.

The NSPK protocol was widely used and acknowledged as secure for almost two decades. In 1995, Gavin Lowe found an attack upon this protocol [33]. This attack involves two concurrent executions, during which the Intruder tries to cheat honest users. The attack upon the NSPK protocol can be represented in Alice–Bob notation:

$$\begin{array}{lll} \alpha_1 & A \to I: & \langle N_A, i(A)\rangle_{K_I} \\ \beta_1 & I(A) \to B: & \langle N_A, i(A)\rangle_{K_B}, \\ \beta_2 & B \to I(A): & \langle N_A, N_B\rangle_{K_A}, \\ \alpha_2 & I \to A: & \langle N_A, N_B\rangle_{K_A}, \\ \alpha_3 & A \to I: & \langle N_B\rangle_{K_I}, \\ \beta_3 & I(A) \to B: & \langle N_B\rangle_{K_B}. \end{array}$$

We added to the earlier notation the following designations that refer to the Intruder's role in the protocol:

- I means the Intruder that occurs as himself;
- $I(u)$ means the Intruder that impersonates user u during the protocol's execution, for example, $I(A)$ means that the Intruder impersonates user A during communication.

In this attack, the Intruder sends during β execution messages received from user A in α execution. The Intruder impersonates user A in β execution. User B thinks he is communicating with user A, and both users are unaware of the dishonest user between them.

Next, in [34], Gavin Lowe proposed a corrected version of the NSPK protocol. The author suggested that an additional identifier in the second step would prevent honest users from engaging in dishonest activities. The structure of the corrected second step in Alice–Bob notation is as follows:

$$\alpha_2 \quad B \to A: \quad \langle N_A, N_B, i(B)\rangle_{K_A}.$$

To date, no documented attacks have been reported against the revised iteration of the NSPK protocol.

4. Description of the Methodology

As mentioned, the main tools used in this work were the proprietary ProToc [35] and the **C**onditional **M**ultiway **M**apped **T**ree (CMMTree) [14]. The ProToc is our universal language used to encode cryptographic protocols.

CMMTree was the main framework defining the structure and mechanism of operation of all the software we created. This solution allows building trees reflecting various relationships between the processed data.

To test our solution, we choose well-known protocols such as Andrew Secure RPC protocol [36] (Andrew), Lowe modified BAN concrete Andrew Secure RPC protocol [37] (Andrew$_{Lowe}$), Carlsen's Secret Key Initiator protocol [38], KaoChow v1 protocol [10], Needham Schroeder Public Key protocol [7] (NSPK), Needham Schroeder Symmetric Key protocol [7], Wide-Mouthed Frog protocol [39], Lowe modified Wide-Mouthed Frog protocol [40], Woo Lam Pi protocol and its first, second, and third versions [9], Yahalom protocol [39], BAN simplified version of Yahalom protocol [39], Lowe's modified version of Yahalom Lowe's protocol [41], and Paulson's strengthened version of Yahalom protocol [42].

In the following subsection, CMMTree will be briefly described. More details on this solution are presented in [14].

4.1. Properties and Functionalities of the CMMTree Model

In this section, we would like to present in the shortest possible way what CMMTree is, that is, an original solution; it has already been described in more detail in [14]. Although we are aware that some of the information presented here will repeat what is written in [14], we have decided that in the context of the entire work, a brief description of this solution is necessary. This is because CMMTree is a general tool designed to build and analyze multi-way conditional trees, and its key component is the predicate function, which can be formulated in many ways—better or worse—as we will try to show in the following sections. We deliberately decided to use many of the same symbols or phrases (as in [14]) so that a person who wants more details about CMMTree can quickly find them in [14].

In general, the CMMTree logic model is described by:

$$\mathbf{CMMTree} = (\mathbf{D}, p, \mathbf{T}) \tag{1}$$

where:

$\mathbf{D} = \{d_1, ..., d_x\}$ is a collection of a unique input data values;
$\mathbf{T} = \{v_i : 0 \leq i < n\}$ is the tree structure of relationships between input data;
$p : \mathbf{D}^k \times \mathbf{T} \rightarrow \{true, false\}, (k = 1, ..., x)$ is a predicate that defines the rules for joining $\mathbf{T}$ nodes.

In this model, three layers are clearly distinguished (see Figure 1): the data layer, predicate layer, and relationship layer. The elements of the input dataset $\mathbf{D}$ should be unique; however, their representations (references to individual values) may occur multiple times in the $\mathbf{T}$ structure. For example, in Figure 1, node v_1 represent the value d_1, node v_2 the value d_2, and so on.

The processed data are not required to be exactly the same type, and they can generally be of different types as long as they belong to a defined class hierarchy. A vector of pointers to them can easily represent such a set of data—see Figure 1. In the present case of the study of cryptographic protocols, the input data were a set of chain objects, described in Section 4.2.

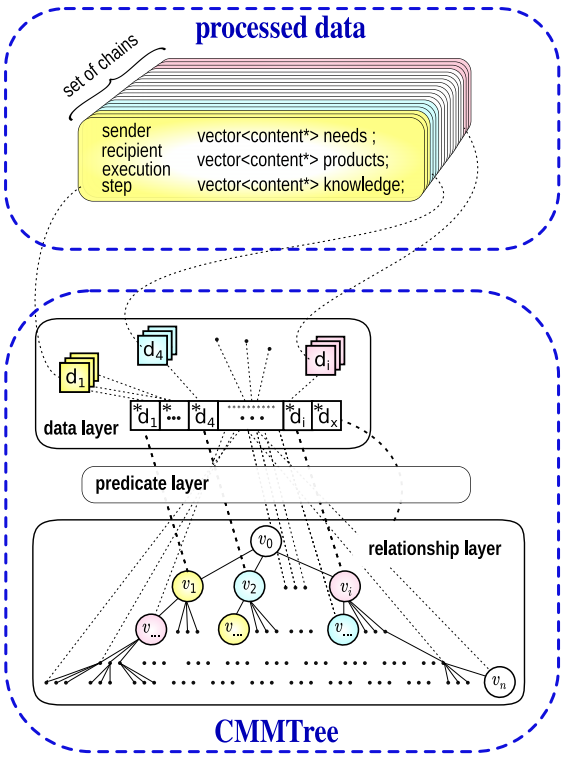


Figure 1. The logical structure of the CMMTree model.

The rules for joining subsequent nodes of the tree **T** are defined in a two-argument predicate function p:

$$p(\alpha, \beta) = \begin{cases} true & \text{if defined conditions are met} \\ false & \text{otherwise} \end{cases} \quad (2)$$

where:

α represents the current node and the set of information available through this node;
β represents the successive elements of the set **D** (candidates to be the α node child).

Each decision on whether another node can be attached is made based on the information accumulated so far in tree **T** and the information provided by candidate β. The amount and scope of information taken from the tree can be varied. Data can be extracted from a single node, nodes on a path, as well as any other combination of nodes. Moreover, if the analyzed data form a class hierarchy, then information about the type of an object or group of objects can also be used by the predicate function p and (if necessary) will be able to operate polymorphically at run-time. The predicate function p (PF) must always be defined. As said before, the situation in which it returns a true value may depend on the required or already existing structure of the relationship between the data. In the simplest variant, p can always return false, but in this scenario, tree **T** will consist of only the root node. Such an implementation of the p function, although not interesting from a practical point of view, is very helpful in the process of code development and testing.

There are many ways to implement multi-way trees, and some of the most commonly used include *left-child/right-sibling* [43–46] (also known as *first_child/next_sibling* [47]), *array of pointers* [44], and *dynamic array-based list of child pointers* [45]. In CMMTree, however, a different solution was used. All nodes of tree **T** are free-stored objects, allocated on the heap. CMMTree does not use node linking specific to binary tree linking [43–45,47]. Each node (except the root) stores the address of its parent and the address of a dynamic vector containing the addresses of its children—see Figure 2.

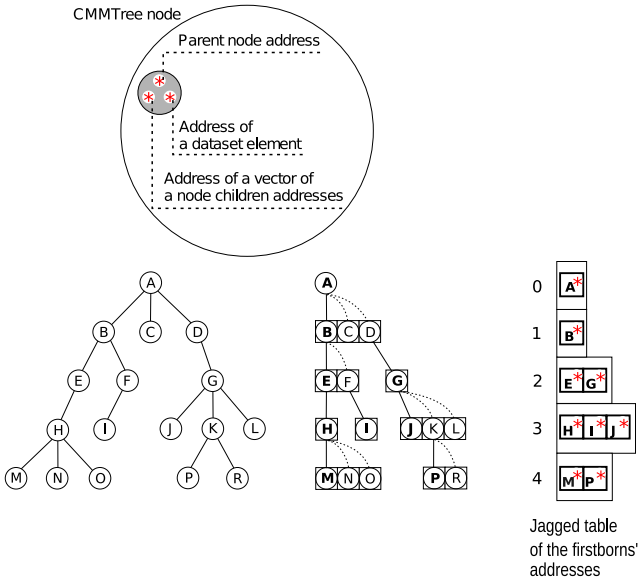


Figure 2. The CMMTree relationship layer implementation diagram. The mark (*) in the jagged table means that each row is a vector of firstborn node addresses.

Each node also stores the address to the appropriate element d_x of the input dataset **D**. In this way, data from the data layer are isolated from the structure created in the relationship layer. Since each node has the direct address of its parent, it is possible to directly determine paths (from a given node to the root) in $O(i)$ time.

The tree construction algorithm was implemented using the level-order strategy. However, unlike the traditional approach [46–48], in which the set of nodes from the last level of the tree is remembered, a custom solution was used in which the set of addresses of only the firstborn nodes is stored [14]—see Figure 2. By using the address of any firstborn node, we gain access to its parent, the parent's children, i.e., siblings of the firstborn, the parent's parent, and so on. Gaining access to the selected node, we immediately know what the size of the vector of addresses of its children is, that is, how many children it has or that it is a leaf. If we have a set of addresses of all the firstborns of one level (l) of the tree **T**, we can also determine the number of all nodes of this level and the number of leaves. Consequently, the set of firstborn addresses from all levels of the tree (see Figure 2) maps the entire structure **T**. Thus, we can see that the single-node structure presented earlier as well as the implementation of interconnections between nodes makes it possible, with the help of firstborn node addresses, to "look deep" into any part of the tree. This eliminates the need for time-consuming traversing of the tree, which in traditional solutions would always have to begin from the root node.

The collection of addresses of all firstborn nodes (see Figure 2, jagged table) is a kind of view of the whole tree **T**. The jagged table is indexed. Thus, it gives free access to the tree **T** of relationships existing between the input data. It directly provides information about the height h of tree **T**:

$$h = \texttt{MTree.size()-1}$$

as well as the number of firstborns (Φ_l) of the selected tree level l:

$$\Phi_l = \texttt{MTree[l].size()}.$$

By selecting any firstborn node address (from the jagged table), we can check the number of all the parent's children $X_{l,j}$:

$$X_{l,j} = children\Big(parent(\varphi_{l,j})\Big) = \texttt{MTree[l][j]->parent->children\_num()},$$

where `children_num()` is a member function of the class `node`. We can also gain direct access to the brother or sister $\nu_{l,k}$:

$$\nu_{l,k} = k\_child\Big(parent(\varphi_{l,j}), k\Big) = \texttt{*(MTree[l][j]->parent->children)[k]}.$$

From the above, we can see that each address of the firstborn node $\varphi_{l,j}$ (`MTree[l][j]`) maps a particular group of nodes, i.e., the parent and its children. This means that the set of addresses of all firstborn nodes maps the entire tree **T**.

Using Formula (3) allows us to determine the total number of nodes N_l of the selected tree level l, and Equation (4) can be used to determine the total number of leaves Λ_l at this level.

As a result, the shape of an entire tree **T** or forest **F** can be (parametrically) described using Formula (5).

$$\bigwedge_{l\in\mathbb{N},\ 0\le l\le h} N_l = \sum_{x=0}^{n_l-1} \nu_{l,x}\ ;\ n_l = \begin{cases} 1 & \text{if } l = 0 \\ \sum_{j=0}^{\Phi_l-1} X_{l,j} & \text{if } l > 0 \end{cases} \tag{3}$$

$$\bigwedge_{l\in\mathbb{N},\ 0\le l\le h} \Lambda_l = \sum_i \lambda_{l,i} = \begin{cases} N_l - \Phi_{l+1} & \text{if } l < h \\ N_l & \text{if } l = h \end{cases} \tag{4}$$

$$\bigwedge_{l\in\mathbb{N},\ 0\le l\le h} shape(T)_l = \{N_l, \Phi_l, \Lambda_l\} \tag{5}$$

Thanks to the parametric form of the CMMTree shape description (5), it is easy to identify nodes that have a significant impact on the shape of the tree. In this way, it is possible to selectively choose those places that should be analyzed first in order to obtain the relevant information. It is especially useful when the examined trees contain tens or hundreds of millions of nodes.

The research in [14] also proposes other measures using N_l, Φ_l, and Λ_l parameters to facilitate the study of huge trees. In this study, these measures were used at the stage of PF optimization.

4.2. Primary Structures and Generalization of the Problem

The implementation of the CMMTree model is written in C++. Its main component is a two-parameter class template:

```
template <class X, template<typename> class pred_type>
class CMMTree{/* ... */};
```

The first parameter (`X`) is a *type parameter*. It represents the type of data for which (as a result of source code compilation) a particular version of the CMMTree class will be created. The second (`pred_type`) is a *template parameter*, i.e., a template of the type parameter, that—in this particular case—is parameterized by the first parameter of the CMMTree class template. By the second parameter, a predicate is passed that specifies the conditions for connecting the tree's nodes. More details on the implementation of the PF are provided in [14].

Various types of so-called primitive structures are used to describe protocol specifications [49,50]. Each of these structures, to a greater or lesser extent, differs from the others regarding the scope of information stored and how to implement specific functionalities. For example, the decryption of messages encrypted with an asymmetric key and those encrypted with a symmetric key must be implemented differently. For this reason, at first glance, specifying only one type of parameter for a solution that is supposed to process many different types of data seems inconvenient. In practice, however, it is quite the opposite, because using the general programming technique allows the code developed in this way to operate on any data structure, provided that it meets certain syntactic and semantic requirements [51]. Therefore, it was decided that the primitives used to record the input data would have to form a hierarchy of classes, shown in Figure 3.

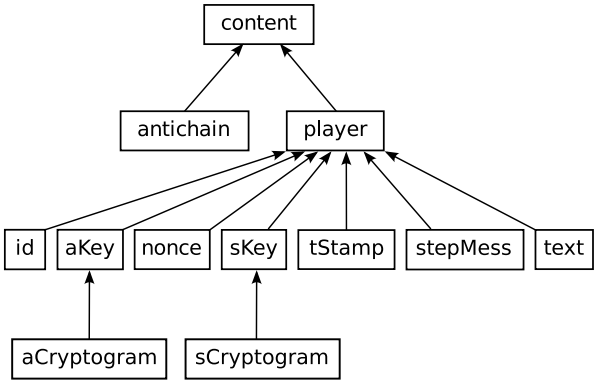


Figure 3. Class hierarchy of processed data.

From the perspective of generalizing the problem of cryptographic protocol processing, the **content** class plays the main role. It is an abstract base class whose main (and only) task is to provide a common interface for the other derived classes. Appropriate objects of derived classes store the real data.

Table 1 presents brief information on classes inheriting from the content class. If only because of the volume, their detailed description goes beyond the scope of this work. However, how objects of these types are used is important. Therefore, to be able to use

the described solution for examining various protocols, the input data (steps of generated protocol executions) are converted into the **chain** type—see Figure 1.

```
class chain{
player sender, recipient;
int execution, step;
std::vector<content*> needs, products, knowledge;
/* ... */
};
```

Table 1. Class hierarchy prepared for protocol predicate.

Class	Description
`player`	information about honest and dishonest protocol participants
`antichain`	information about the ways of step execution by Intruder
`id`	information about the identifiers of protocol participants
`aKey`	information about users' asymmetric keys
`nonce`	information about the random number that is used only once during the protocol execution
`sKey`	information about users' symmetric keys
`tStamp`	information about the user timestamp that indicates the moment of the message creation
`stepMess`	information about messages sent during the step
`text`	information about the message part that will be sent as plain text
`aCryptogram`	information about cryptogram encrypted by asymmetric key
`sCryptogram`	information about cryptogram encrypted by symmetric key

In this class, all information, the quantity, type, and scope of which are not known beforehand, is represented by the vectors of pointers to the **content** type. Having the addresses of individual objects, the polymorphism of virtual functions and the RTTI (**R**un-**T**ime **T**ype **I**nformation—a mechanism that provides information about an object's data type at run-time) mechanism [51] can be employed to extract detailed information specific to the actual type of data being processed.

Vector **needs** represent (that is, contain the addresses of the relevant objects) a collection of objects that the sender needs to complete a step. For example, this vector may contain addresses of `nonce`, `tStamp`, `aCryptogram`, or `sCryptogram` class objects. However, it cannot contain addresses of objects that represent publicly known objects to protocol participants (public keys, identifiers).

Vector **products** represent objects that the sender must generate before the protocol step execution. These objects are also needed for the sender to complete the protocol step. Usually there are addresses of objects of type `tStamp`, `nonce`, or `sKey`.
The **knowledge** vector is intended for objects containing information the recipient learns during a given protocol step. For instance, these can be objects of classes `nonce`, `tStamp`, or `aCryptogram`.

The **antichain** class refers to objects representing how an Intruder can execute a protocol step depending on his knowledge. For example, if an Intruder should send the following message $< N_A \cdot N_B >_{K_{AB}}$ according to the execution structure, he has two possible ways to execute this step. Firstly, he can send the message if he knows the entire ciphertext. In a second method, he can send the message if he has in his knowledge each element forming a ciphertext (N_A, N_B, and K_{AB}). Analogous to the chain class, vectors of pointers represent the primitives whose actual type can be known only during input processing to the content class.

At the predicate optimization stage, we used lexicographic sorting of protocol executions. The consistent use of a specific order of marking executions and steps definitely facilitates the analysis of both the tested protocol and the PF code under test. We compared the obtained ETs with trees built for randomized data to check the performance of the created PF.

Each analyzed chain is uniquely identified by a pair of $(execution\_number, step\_number)$. The key to sorting the execution of the protocol is information identifying players and information provided by sets of cryptographic objects that the Intruder can use during the execution of the protocol.

Also, it is worth mentioning that our methodology considers four Intruder models. The set of executions contains honest and dishonest (with Intruder) executions. Executions with Intruder are generated for the following Intruder models:

- Dolev-Yao model—in which the Intruder controls the network, accesses transmitted messages, can intercept, block, and process messages against the protocol, but requires a decryption key for ciphertext information [52].
- Lazy Intruder—which can be considered a type of virus that can deliver complete messages, yet cannot alter or modify them [53,54].
- Restricted Dolev-Yao—enabling a feature that the Dolev-Yao Intruder may only access messages specifically directed to them [5].
- Restricted Lazy Intruder—enabling a feature that the Lazy Intruder may only access messages specifically directed to them [5].

For example, in the NSPK protocol, we have two players (A and B). We add an Intruder (I) to the players' set. The sorted player set has an Intruder on zero indexes. Next, based on the players' set, we generate a set of all possible executions using variations without repetition. In this way, we obtained the following sorted set of executions (using sender $\rightarrow$ recipient designation):

$$I \rightarrow A,\ I \rightarrow B,\ A \rightarrow I,\ A \rightarrow B,\ B \rightarrow I,\ B \rightarrow A.$$

Also, for each mentioned execution with the Intruder, we generate three parallel executions different from each other by cryptographic primitives used by the Intruder. For such an ordered set of possible protocol executions, one of the four paths containing an attack on the NSPK protocol will be represented by steps (10,1), (8,1), (8,2), (10,2), (10,3), and (8,3), i.e., step one of execution no. 10, step one of execution no. 8, and so on.

We prepare the set of chains (SoC) for all possible protocol executions. Each chain has this same structure as described earlier. The content of each vector may be complex, and everything depends on the protocol structure. Tables 2 and 3 present two fragments of the SoC for protocol execution involving an Intruder. Both tables have the same structure. Column **Send.** $\rightarrow$ **Rec.** contains information about the sender and receiver in the current protocol step. Column **Execution & step** includes execution and step number information. The rest of the columns (**Needs**, **Products**, and **Knowledge**) contain information about the content of corresponding vectors.

Table 2. The fragment of SoC for NSPK protocol (execution no. 1).

Send. $\rightarrow$ Rec.;	Execution & Step;	Needs;	Products;	Knowledge;
$I \rightarrow A$;	(1, 1);	$\langle i(I), N_I \rangle_{K_A^+}$;	;	N_I;
$I \rightarrow A$;	(1, 1);	;	;	N_I;
$A \rightarrow I$;	(1, 2);	N_I;	N_A;	N_A;
$I \rightarrow A$;	(1, 3);	$\langle N_A \rangle_{K_A^+}$;	;	N_A;
$I \rightarrow A$;	(1, 3);	N_A;	;	N_A;

Table 2 contains the chains for execution no. 1 generated for the NSPK protocol in Alice–Bob notation. Each chain from this execution consists of each part described earlier in this section, separated by semicolons. The Intruder is a sender in the first and third steps, so he can execute each of these steps in two ways (sending the entire ciphertext or the ciphertext composed using his knowledge).

Table 3. The fragment of SoC for KaoChow protocol (execution no. 9).

Send. $\rightarrow$ **Rec.;**	**Execution & Step;**	**Needs;**	**Products;**	**Knowledge;**
$A \rightarrow S$;	(9, 1);	;	N_A;	N_A;
$S \rightarrow I$;	(9, 2);	N_A ;	K_{IA};	$\langle i(A)\vert i(I)\vert N_A\vert K_{IA}\rangle_{K_{AS}}, i(I), N_A, K_{IA}$;
$I \rightarrow A$;	(9, 3);	$\langle i(A)\vert i(I)\vert N_A\vert K_{IA}\rangle_{K_{AS}}, \langle N_A\rangle_{K_{IA}}$;	;	$i(I), N_A, K_{IA}, N_I$;
$I \rightarrow A$;	(9, 3);	$K_{AS}, N_A, K_{IA}, \langle N_A\rangle_{K_{IA}}$;	;	$i(I), N_A, K_{IA}, N_I$;
$I \rightarrow A$;	(9, 3);	$\langle i(A)\vert i(I)\vert N_A\vert K_{IA}\rangle_{K_{AS}}, K_{IA}, N_A$;	;	$i(I), N_A, K_{IA}, N_I$;
$I \rightarrow A$;	(9, 3);	$K_{AS}, N_A, K_{IA}, K_{IA}, N_A$;	;	$i(I), N_A, K_{IA}, N_I$;
$A \rightarrow I$;	(9, 4);	N_I;	;	N_I;

Table 3 contains the chains for execution no. 9, generated for the KaoChow protocol in Alice–Bob notation. The KaoChow protocol has a different and more complex structure than the NSPK protocol, so this chain contains more entries and objects in individual vectors. The Intruder is a sender in the third step only, but he can execute this step in four ways when his knowledge contains the following objects:

- two entire ciphertexts;
- the objects from the first ciphertext and the entire second ciphertext;
- the entire first ciphertext and objects from the second ciphertext;
- the objects from both ciphertexts.

In our opinion, the above-described method of combining many different data types into a hierarchy of classes that inherit from a very general base class is a very flexible solution. If a new data type appears, it is enough to "put" it in the right place in such a hierarchy to be able to study the impact of the new data on the overall structure of the relationship between all data. However, in each case of the use of any particular non-abstract type of data, we assume that it is provided with the right set of functionalities—that is, functions which, if necessary, can "cooperate" with other types of that hierarchy.

4.3. Predicate Function: Searching for the Best Possible Solution

The predicate function p (2) (PF) plays a key role in the CMMTree model (1). The rules for connecting successive tree nodes written there will directly impact the correctness of the obtained results and the data analysis time. It will depend on these rules whether analyzing a given problem at all with specific computing resources will be possible.

For such special data as chains of protocol executions, the use of only rules "**for yes**"—according to the general principle (if the conditions are met, the node can be attached)—is clearly insufficient. This is due to the fact that the ETs will grow to gigantic sizes, and very often, they will grow indefinitely. Therefore, rules "**for not**" are also necessary for the PF, to allow effective pruning of "non-perspective" branches during the construction of the ET.

One of the main goals of this work was to develop and test a tool that would allow the study of various cryptographic protocols. Therefore, it was necessary to find a compromise between a solution specialized for a specific protocol and a generalized solution allowing processing and analyzing a broader spectrum of protocols.

We used an approach in which our solution (and related protocol analysis) was tested in four main stages.

The text-encoded protocol (in the ProToc language [35]) was loaded into the program at stage one. There, its validation was carried out for the correctness of the writing. Then the protocol was converted into object (binary) form, according to the class hierarchy described in Section 4.2. From this point on, all subsequent protocol processing was carried out using objects of this class hierarchy.

In the second step, the original form of the protocol was processed into a set of protocol executions. These executions were ordered lexicographically (at the stage of searching for the best possible form of the PF) or shuffled when a given form of the PF was evaluated.

In the third step, the input data were transformed into a set of chains, according to the methodology described in [15,16].

In the last—fourth—stage of our research, different variants of the protocol ETs were created and evaluated.

It was at this stage that the general CMMTree model—along with its functionalities (see Section 4.1)—was concretized for the chain type containing primitives (see Section 4.2) defining individual protocols.

We tested various conditions that decide the attachment of subsequent tree nodes. These conditions were evaluated for their impact on tree construction time and the time to find possible attacks, as well as their impact on the overall size of the ET.

We tried to prune the branches of each tree as much as possible (during its construction), ensuring that no important data were unnecessarily omitted at any stage of data processing.

As already mentioned, the executions of each tested protocol were first sorted lexicographically. Thanks to this, we were able to ensure the repeatability of testing conditions for both the protocol and the PF. Then, the tested version of the PF was run several times on randomized data. We used the std::mersenne_twister_engine algorithm [55] to shuffle the set of chains of all possible protocol executions.

We considered a tested rule to be valuable (in the context of the entire predicate) if the use of a given version of the PF allowed us to:

- find attacks for those protocols for which attacks are known;
- significantly reduce the size of the ET of each of the tested protocols;
- obtain ETs with the same CMMTree shape description (5), both for sorted and shuffled data.

We have also implemented mechanisms to prevent the tree from growing indefinitely, i.e., time safeguard and protection that controls the total number of nodes in the tree. The program stopped its work after saving all current results when 3 h of operation elapsed (for one protocol) or the tree reached one billion nodes. The values of these protections were determined experimentally for the computer we used. It was a desktop computer with an AMD Ryzen 9 3900X processor (12 cores, 24 threads) and 64 GB RAM, with the Linux Mint 20.3 operating system. The CMMTree construction algorithm has been implemented as a single separate thread of the program. Separate threads responsible for implementing the protections mentioned above did not block the processed data.

Figure 4 illustrates the main idea behind the pruning process of the protocol ET. An example is the ET (a small fragment of it, to be precise) of the NSPK protocol. Here we can see that the CMMTree is actually a forest, that is, a collection of the root node subtrees. This is a case where the CMMTree root node does not represent any element from the input dataset. The nodes corresponding to the possible executions of the first step of the protocol are the roots of these subtrees. CMMTree takes the form of a forest with other protocols as well. We also see two exemplary protocol paths, resulting in the execution of its last step, and one path that can be considered "non-perspective" and suitable for pruning. The path of steps (18,1), (18,2), and (18,3) is the proper realization of the protocol. The path of (10,1), (8,1), (8,2), (10,2), (10,3), and (8,3) is the realization of the protocol with the participation of the Intruder, which ends with a successful attack. The path of (10,1), (1,1), (3,1), and so on is not prospective because it does not guarantee the expected progress in the execution of the protocol. Furthermore, each of these nodes could be treated as the root of another subtree, and thus, the forest would grow to infinity.

Direct analysis of trees containing millions or tens of millions of nodes is beyond human perceptual capabilities—certainly beyond our abilities. Therefore, to evaluate the effects of our experiments, we used quantitative and qualitative measures describing the shape of created trees, as discussed in [14]. These measures are based on the parametric form of the CMMTree description expressed by Formula (5). Among other things, they allow us to immediately identify places where—in general—something happened, or something started to happen. Since CMMTree is a mapped structure, such a place can be

reached in $O(1)$ time without traversing the entire tree, which would always have to start at the root.

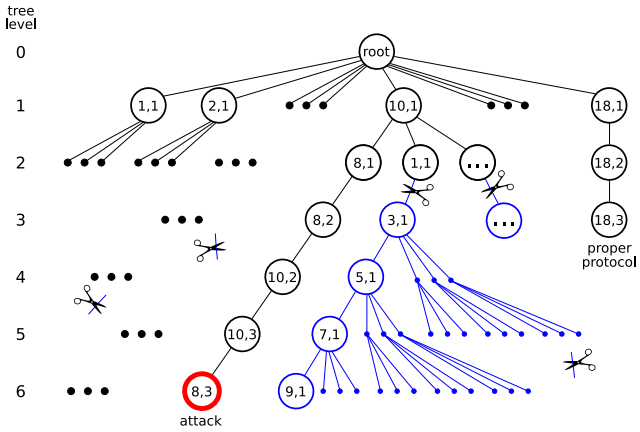


Figure 4. Illustration of the CMMTree pruning process.

In further explanations, we will also use:

$$\Delta\Phi = \Phi_l - \Phi_{l-1} \text{ and } \Delta\Lambda = \Lambda_l - \Lambda_{l-1}.$$

Figure 5 and Table 4 portray three examples of CMMTree shape descriptions of the execution tree (ET) of the NSPK protocol obtained during the development of the predicate function (PF).

Figure 5a is an image of the ET constructed using the PF without conditions that cause the pruning of branches—without "for not" conditions. Tree construction was terminated (on tree level $l = 10$) after exceeding the 1 billion node limit. The tree building time was almost 1005 s, and the entire CMMTree structure took 50.7 GB of RAM.
This graph clearly shows the exponential increase in the number of nodes at each tree level. Each level has an average of 10 times more nodes than the previous level. The tree grows almost symmetrically. Up to level 5, each node has descendants. This is obvious because the Λ_l, and consequently Λ_l/N_l measures (see Table 4, part 1), have values equal to zero. As can be seen, the measure Φ_l/N_l runs nearby values of 0.1, so each node has an average of 10 children. A careful analysis of the Φ_l/N_l and $\Delta\Phi/(\Phi_l + \Phi_{l-1})$ measures shows a slight decrease in the tree's growth dynamics, but this tendency is practically insignificant. Most of the leaves appeared on level 9. This is directly due to the activation of the safety mechanism against the unlimited growth of the tree.

Figure 5b presents a tree image obtained during the evaluation of one of the variants of the PF using a lexicographically sorted set of input data. As in the previous case, the tree construction was terminated at level 10. This time, however, it resulted from activating one of the "for not" conditions, one of the applied pruning conditions (PCs). The tree contains 11,139 nodes. It was built in 65 milliseconds. Four expected attack paths were found in this tree.

Comparing Figure 5a,b, we can immediately see that the use of a few other PCs significantly changes the size and shape of the ET that is created. If we look at the measure Φ_l/N_l, we can see that from level 3, its value is greater than 0.5, meaning that from level 2, the average parent node has no more than two children. By analyzing the Λ_l/N_l measure, it can be seen that pruning at levels 2 and 3 proved to be the most important. Thirty-seven percent of level 2 nodes and almost thirteen percent of level 3 nodes are leaves. The identified four leaves at level 6 (see Table 4, part 2) correspond to the four executions of the protocol with a successful attack.

Table 4. The CMMTree shape descriptions of the NSPK protocol execution tree: part 1—the ET constructed without using PCs; part 2—the ET pruned using one of the tested sets of PCs; part 3—the ET pruned using the final set of PCs and the final form of the PF.

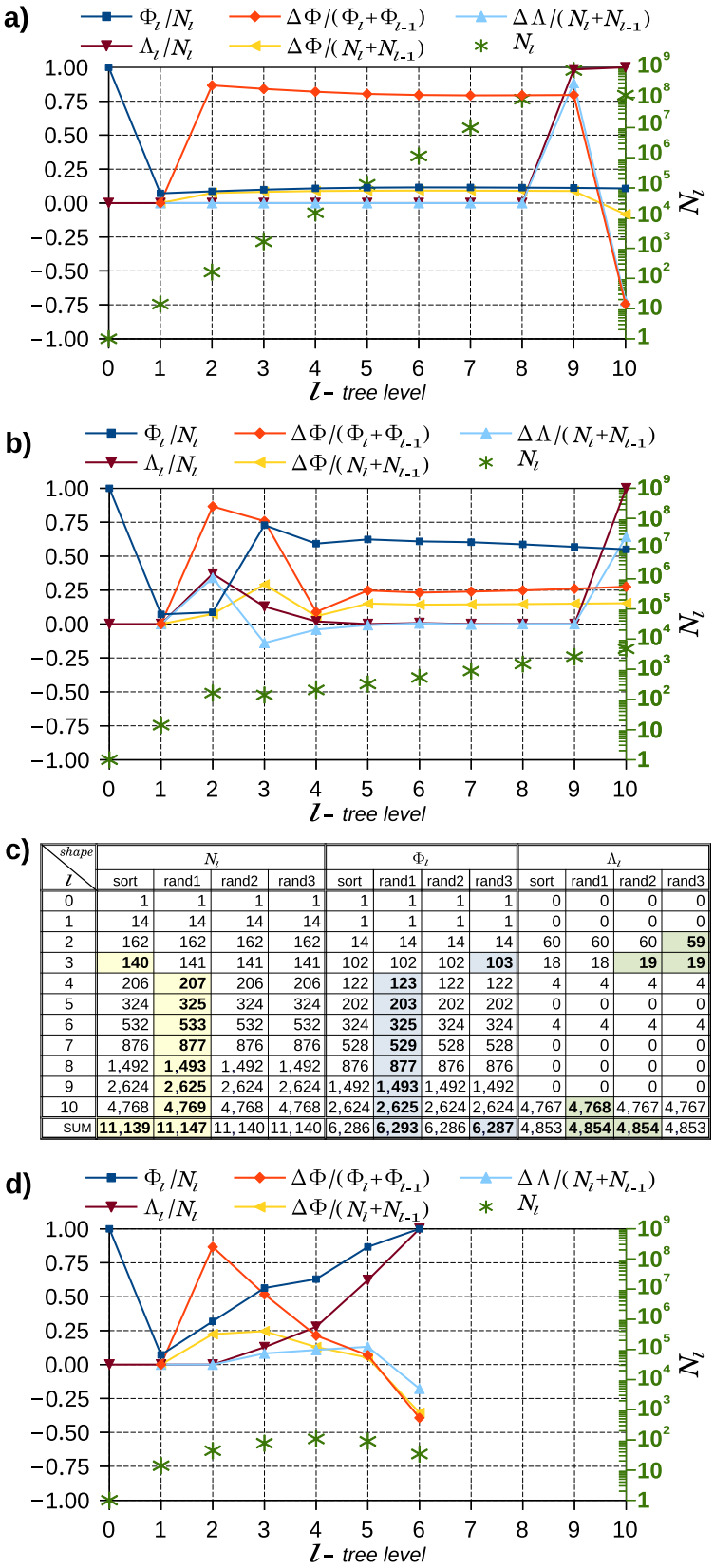
Level	N_l	Φ_l	Λ_l	Φ_l/N_l	Λ_l/N_l	$\Delta\Phi/N_l$	$\Delta\Lambda/N_l$	$\frac{\Delta\Phi}{(\Phi_l+\Phi_{l-1})}$	$\frac{\Delta\Phi}{(N_l+N_{l-1})}$
0	1	1	0	1	0	1	0	1	1
1	14	1	0	0.071429	0	0	0	0	0
2	162	14	0	0.086420	0	0.080247	0	0.866667	0.073864
3	1638	162	0	0.098901	0	0.090354	0	0.840909	0.082222
4	15,066	1638	0	0.108722	0	0.097969	0	0.82	0.088362
5	132,304	15,066	0	0.113874	0	0.101494	0	0.803879	0.091118
6	1,145,680	132,304	4	0.115481	3.49×10^{-6}	0.102330	3.49×10^{-6}	0.795535	0.091737
7	9,954,076	1,145,676	124	0.115096	1.25×10^{-5}	0.101805	1.21×10^{-5}	0.792948	0.091297
8	87,538,106	9,953,952	2278	0.113710	0.000026	0.100622	0.000025	0.793565	0.090349
9	782,124,012	87,535,828	769,269,060	0.111921	0.983564	0.099194	0.983561	0.795795	0.089209
10	119,088,942	12,854,952	119,088,942	0.107944	1	−0.627102	−5.45962	−0.743902	−0.082867
0	1	1	0	1	0	1	0	1	1
1	14	1	0	0.071429	0	0	0	0	0
2	162	14	60	0.086420	0.370370	0.080247	0.370370	0.866667	0.073864
3	140	102	18	0.728571	0.128571	0.628571	−0.3	0.758621	0.291391
4	206	122	4	0.592233	0.019418	0.097087	−0.067961	0.089286	0.057804
5	324	202	0	0.623457	0	0.246914	−0.012346	0.246914	0.150943
6	532	324	4	0.609023	0.007519	0.229323	0.007519	0.231939	0.142523
7	876	528	0	0.602740	0	0.232877	−0.004566	0.239437	0.144886
8	1492	876	0	0.587131	0	0.233244	0	0.247863	0.146959
9	2624	1492	0	0.568598	0	0.234756	0	0.260135	0.149660
10	4768	2624	4767	0.550336	0.999790	0.237416	0.999790	0.275024	0.153139
0	1	1	0	1	0	1	0	1	1
1	14	1	0	0.071429	0	0	0	0	0
2	44	14	0	0.318182	0	0.295455	0	0.866667	0.224138
3	78	44	10	0.564103	0.128205	0.384615	0.128205	0.517241	0.245902
4	108	68	30	0.62963	0.277778	0.222222	0.185185	0.214286	0.129032
5	90	78	56	0.86667	0.622222	0.111111	0.288889	0.068493	0.050505
6	34	34	34	1	1	−1.29412	−0.647059	−0.392857	−0.354839

Despite finding attack paths (also for shuffled data) and a huge reduction in the number of tree nodes and their construction time, the examined PF could not be considered satisfactory by us for two main reasons. First, if the condition interrupting the tree's construction at level $l = 10$ had not been applied, it could have continued to grow. No leaf was identified for the next three levels of the tree; the Λ_l and Λ_l/N_l measures have values equal to zero. This means that none of the tree's other branches have been pruned, i.e., none of the remaining paths have been terminated. The second reason was the change in the parameters of the tree shape description after applying the input data shuffle operation.

Figure 5c shows a table of the parameters of the CMMTree shape description for sorted data and three samples of shuffled data. We can see that changing the processing order of chains of the protocol executions caused (small, but noticeable) changes in parameters N_l, Φ_l, and Λ_l. Nevertheless, this should not be the case when processing the same data.

Figure 5d presents the description of the CMMTree shape obtained using the final form of PF. This tree contains 369 nodes. It was built in about 5 milliseconds, and the parametric description of its shape does not depend on the order in which the input data are processed. Its construction was completed at level $l = 6$ after exhausting the possibilities of connecting further nodes without using any additional PCs that would be conditions of a safeguard nature. This graph shows that parameter N_l reaches its maximum value at level $l = 4$ and then decreases. From level $l = 3$, measure Λ_l/N_l takes on increasing values, meaning that more nodes of a given level become leaves. At level $l = 6$, measures Φ_l/N_l and Λ_l/N_l take

the value 1. This means that each node of this level is a firstborn node and has no more descendants.



c)

shape / l	N_l				Φ_l				Λ_l			
	sort	rand1	rand2	rand3	sort	rand1	rand2	rand3	sort	rand1	rand2	rand3
0	1	1	1	1	1	1	1	1	0	0	0	0
1	14	14	14	14	1	1	1	1	0	0	0	0
2	162	162	162	162	14	14	14	14	60	60	60	**59**
3	**140**	141	141	141	102	102	102	**103**	18	18	**19**	**19**
4	206	**207**	206	206	122	**123**	122	122	4	4	4	4
5	324	**325**	324	324	202	**203**	202	202	0	0	0	0
6	532	**533**	532	532	324	**325**	324	324	4	4	4	4
7	876	**877**	876	876	528	**529**	528	528	0	0	0	0
8	1,492	**1,493**	1,492	1,492	876	**877**	876	876	0	0	0	0
9	2,624	**2,625**	2,624	2,624	1,492	**1,493**	1,492	1,492	0	0	0	0
10	4,768	**4,769**	4,768	4,768	2,624	**2,625**	2,624	2,624	4,767	**4,768**	4,767	4,767
SUM	**11,139**	**11,147**	11,140	11,140	6,286	**6,293**	6,286	**6,287**	4,853	**4,854**	**4,854**	4,853

Figure 5. The example of the CMMTree shape descriptions of the NSPK protocol execution tree.

5. Experimental Results

The obtained results seem to be promising. They allow us to conclude that the solutions we use make it possible to generalize research on cryptographic protocols. For the CMMTree we were using, we were able to formulate one form of the predicate function (PF) that we could use to analyze sixteen different protocols. We were able to correctly find attacks for the Andrew, Andrew$_{Lowe}$, and NSPK protocol. For these protocols, we have also shared "draft reports" (https://cloud.icis.pcz.pl/s/zZmiiBA2YqxDprC, accessed on 14 November 2023) received during the operation of our software. They contain visualizations of the various stages of data processing, starting from the protocol encoded in the Protoc language, through chains of executions, parametric description of the CMMTree, and finally execution paths leading to attacks. These data allow us to see the difference between sorted and randomized data for building the tree. Input data used for building the tree are between headers "CHAINS—input data:" and "The time of the CMMTree calculation:" (for the NSPK protocol: lines 361–438, for the Andrew protocol: lines 608–767, and for AndrewLowe: lines 608–767).

Figure 6 presents a parametric description of the shape of the execution trees (ETs) of these three protocols. Next to the columns with the values of parameters N_l, Φ_l, and Λ_l, the creation times of these trees are also included. The time in the row with index 0 refers to the tree created based on lexicographically sorted data. Rows indexed by 1, 2, and 3 refer to results obtained from shuffled data. The result in row no. 4 is the arithmetic mean of the previous values.

protocol / l	ANDREW				ANDREW$_{LOWE}$				NSPK			
	N_l	Φ_l	Λ_l	[ms]	N_l	Φ_l	Λ_l	[ms]	N_l	Φ_l	Λ_l	[ms]
0	1	1	0	1.6	1	1	0	1.9	1	1	0	4.7
1	18	1	4	2.1	18	1	4	2.2	14	1	0	5.2
2	14	14	0	1.9	14	14	0	2.1	44	14	0	4.7
3	14	14	6	1.8	14	14	6	1.5	78	44	10	5.5
4	8	8	2	***1.9***	8	8	2	***1.9***	108	68	30	***5.0***
5	6	6	0		6	6	0		90	78	56	
6	6	6	0		6	6	0		34	34	34	
7	6	6	0		6	6	0					
8	6	6	6		6	6	6					
sum	**79**	**62**	**18**		**79**	**62**	**18**		**369**	**240**	**130**	

Figure 6. Parametric description of the execution tree shape for Andrew, Andrew$_{Lowe}$, and NSPK protocol obtained using the PF implementing our proposed pruning rules.

As we can see, the ETs of these protocols contain a negligible number of nodes. The vast majority of non-perspective paths have been successfully removed. However, all relevant data remained, and we could find all possible attack paths for each of these protocols.

It may be interesting to note that although protocols Andrew and Andrew$_{Lowe}$ are considered to be two different protocols, both the average times of tree creation and the parametric description of their shape (5) are identical.

In addition to paths representing attacking executions, our tool found paths that represent executions between honest users ($A \rightarrow B$ and $B \rightarrow A$ in Alice–Bob notation). They appeared at the level indicating protocol step number. For Andrew and Andrew$_{Lowe}$ protocols, at $l = 4$ in column Λ_l, we have two paths that represent honest executions. We have ten paths for the NSPK protocol at $l = 3$ in column Λ_l. Two of them represent honest executions, and the rest of them represent paths with an Intruder.

For the other thirteen protocols tested, we did not find attacks. But it is probably good news that we did not find them.

In the discussed CMMTree model, the PF is provided as a function object. In practice, it is implemented as a function called operator overloading. In its final form, which is the result of the tests described in Section 4.3, the PF can be written according to Algorithm 1.

Algorithm 1: ***bool*** *predicate<chain>::operator*()(α, β)

```
   Data:
   α—current node,
   β—candidate to be the current node child.
   Result: true or false
1  begin
2  {
3      P ← path(α) ;
4      if ( length(P) > 0 ) then
5      {
6          if ( check_attack(P, β) ) then
7          {
8              AP ← add_next_attack_path(P, β) ;
9          }
10         /*
11            The safeguard functions can be
12            included here--if necessary
13         */
14         if ( chronology(β, P)
15              and needs(β, P)
16              and nonces(β, P)
17            ) then
18         {
19             return true ;                          /* β may be attached  */
20         }
21     }
22     else
23     {
24         if ( needs(β) = ∅ and step(β) = 1 ) then
25         {
26             return true ;                          /* β may be attached  */
27         }
28     }
29     return false ;
30 }
```

At this point, we would like to remind the reader that CMMTree does not store data in the literal sense. Individual nodes only contain addresses to other data (see Figure 2). Thus, paths are also sets of corresponding addresses. We use pointers and references everywhere because there is no point in copying data objects that have already been created. Furthermore, the run-time polymorphism we use is available by late-binding function calls using only pointers or references.

However, to simplify the description of the presented algorithms, we will treat the nodes of the CMMTree, nodes on paths, etc., as objects (of the chain type) that store relevant data.

In Algorithm 1, we can see that this is a two-argument function, where the first argument represents the successive nodes of the current tree form, while the second argument represents the successive values from the input dataset (set of chains, in this case). It returns true if an element from the input dataset can be considered a descendant of the current node. **P** is a set of nodes on the path from the current node to the root (**P**[0]—current, **P**[1]—previous, and so on). So it is a set of information that was collected

after performing a certain number of steps of the protocol being tested. We can also see that attaching subsequent nodes can be carried out in two general cases:

- there are no nodes on the path $(length(\mathbf{P}) = 0)$,
  and the current chain (as a candidate) is the chain of the first step of the protocol $(step(\beta) = 1)$,
  and the set of needs necessary to perform the first step is the empty set $(needs(\beta) = \emptyset)$;
- there are appropriate nodes in the path $(length(\mathbf{P}) > 0)$,
  and connecting the next node will not disturb the chronology (defined by the protocol) of the steps on the path $(chronology(\beta, \mathbf{P}))$,
  and on the path, there is a set of information that allows satisfying the set of needs necessary to connect the next node $(needs(\beta, \mathbf{P}))$,
  and the number of nonces generated by one user on the path matches the number of nonces per user declared in the protocol structure $(nonces(\beta, \mathbf{P}))$.

The instructions in lines 6–9 do not affect the result returned by the operator(). They are auxiliary. They allow for potential attack paths (**AP**) to be remembered already during the construction of the tree.

Below, but always before the instructions of lines 14–20, safety functions can be placed. Meeting the conditions they check should always cause the operator() to return false. We used such functions in the initial stages of studying different variants of the whole PF. For example, we were finishing a tree construction at level 10.

Since CMMTree is a mapped structure (see Section 4.1), such controlled stopping of its construction allowed us to select single paths for detailed analysis to look for exploitable regularities.

The need to use security functions in the PF considered the target indicates that the PF can still be improved.

We are convinced that a properly defined PF does not require any additional protection against the uncontrolled expansion of the tree.

We also tested a predicate function using only the pruning rules proposed by Mödersheim in [56], just as rules to prevent unlimited tree growth. Comparing the results obtained (see Figures 6 and 7), we found that our algorithm builds ETs much faster with fewer nodes. For example, we found attacks upon the Andrew$_{Lowe}$ protocol more than 160,000 times faster than using the pruning rules from [56], creating a tree with 278,848 times fewer nodes.

protocol / l	ANDREW				ANDREW$_{LOWE}$				NSPK			
	N_l	Φ_l	Λ_l	[ms]	N_l	Φ_l	Λ_l	[ms]	N_l	Φ_l	Λ_l	[ms]
0	1	1	0	323	1	1	0	317,289	1	1	0	8,913
1	18	1	0	292	18	1	0	301,806	14	1	0	8,333
2	176	18	18	318	176	18	8	297,542	162	14	0	8,646
3	350	158	0	289	386	168	0	300,912	1,638	162	0	8,670
4	682	350	2	***306***	1,462	386	0	***304,387***	15,066	1,638	0	***8,641***
5	1,310	680	0		11,226	1,462	0		132,304	15,066	0	
6	2,552	1,310	0		125,994	11,226	0		1,145,680	132,304	1,145,679	
7	5,060	2,552	0		1,562,870	125,994	0					
8	10,250	5,060	10,249		20,326,888	1,562,870	20,326,887					
sum	**20,399**	**10,130**	**10,269**		**22,029,021**	**1,702,126**	**20,326,895**		**1,294,865**	**149,186**	**1,145,679**	

Figure 7. Parametric description of the execution tree shape for Andrew, Andrew$_{Lowe}$, and NSPK protocols obtained using the PF implementing only pruning rules proposed in [56].

We also see clearly that Andrew and Andrew$_{Lowe}$ protocols are in fact two different protocols, and how different the parametric descriptions are of the shape of their ETs.

In Algorithm 2, we have specified the function chronology in pseudo-code. This function checks whether the candidate (β) for the current node's child matches the chronology of the information increment on the path.

Algorithm 2: ***bool*** *predicate<chain>::chronology*(β, **P**)

```
Data:
β—candidate to be the current node child,
P—data on path.
Result: true or false
1  begin
2  {
3      L ← length(P);
4      for i = 0 to L − 1 do
5      {
6          if (β_exec() == P[i]_exec()) then
7          {
8              for j = i to L − 1 do
9              {
10                 if (can_be_prune(β, P[j])) then
11                 {
12                     return false ;
13                 }
14             }
15             if (β_step() == (P[i]_step() + 1)) then
16             {
17                 return true ;
18             }
19         }
20         else
21         {
22             for j = i to L − 1 do
23             {
24                 if (can_be_prune(β, P[j])) then
25                 {
26                     return false ;
27                 }
28             }
29             if (β_step() == 1) then
30             {
31                 return true ;
32             }
33         }
34     }
35     return false ;
36 }
```

In this function, we have placed the rules "for not", as mentioned in Section 4.3, i.e., rules that result in pruning non-perspective branches. Here, we mainly rely on information regarding the protocol execution and step numbers. This information is returned, respectively, by the functions *exec*() and *step*().

The general rule is very simple. If the compared chain objects come from the same execution (β_*exec*() == **P**[*i*]_*exec*()), **and** candidate (β) will not disturb the chronology of the data on the path (the *can_be_prune*(β, **P**[*j*]) function returns false), then adding a new node is possible if the candidate's step is 1 larger than the last step executed (β_*step*() == (**P**[*i*]_*step*() + 1)).

However, if the candidate is an object from another execution, it must be an object representing the first step of the protocol ($\beta\_step() == 1$). As in the previous case, we call a function that checks whether this new path can be pruned before deciding to attach another node.

Algorithm 3, shown below, is pseudo-code of the $can\_be\_prune(\beta, \mathbf{P}[j])$ function. Here the role performed by the Intruder is checked. As mentioned earlier, the Intruder can impersonate an honest participant in the protocol. However, he cannot once again perform a step that was already performed by the other participant, regardless of whether it was a step performed by the sender or the receiver. Any such path, which after joining the candidate (β) could contain nodes of repeated steps, was classified as prunable, i.e., a path with a disturbed order of the steps performed.

Algorithm 3: ***bool*** *predicate<chain>::can_be_prune*(β, $\mathbf{P}[j]$)

```
Data:
β—candidate to be the current node child,
P[j]—j-th chain on the path.
Result: true or false
1  begin
2  {
3      if (β_step() == P[j]_step()
4          and
5          ( β_recipient()_role() ==
6            P[j]_recipient()_role()
7            or
8            β_recipient()_role() ==
9            P[j]_sender()_role()
10         )
11       ) then
12     {
13         return true ;
14     }
15     return false ;
16 }
```

6. Conclusions

This paper offers a comprehensive exposition and evaluation of cryptographic protocols. We have adapted the methodology described in [15,16] to the principles of generalized programming and clearly distinguish the individual stages of the protocol study. We have proposed a way to group the primitive structures used to describe protocols into a coherent class hierarchy. This approach allowed us to make successive transformations of data types (not only and only their values), adapting them to a form convenient to use at a given level of abstraction.

Using the proprietary CMMTree framework and its tools, we were able to focus on the analysis of the tested protocols. Without interfering with the code responsible for creating trees, often containing huge numbers of nodes, we could study both sorted and randomly shuffled data. We could examine selected parts of the ETs of these protocols to look for patterns, which we later used to create rules used to prune paths "leading nowhere".

We tested our solution using 16 protocols. We could define a general form of the predicate function (which defines the rules for creating protocol ETs), which allowed us to test these protocols. We were able to find attacks for three of them correctly. We were surprised by how effectively non-perspective paths can be pruned while leaving all relevant data in the ET.

We are aware that there may or will be other protocols for which our solutions may not be effective or efficient enough. We are convinced, however, that in the approach we propose, it is possible to find a compromise between a solution specialized for one protocol and a general solution for a wider group of protocols.

Author Contributions: Conceptualization, J.P. and S.S.; methodology, J.P.; software, J.P.; validation, J.P. and S.S; formal analysis, S.S.; investigation, J.P. and S.S.; resources, S.S.; data curation, J.P. and S.S.; writing—original draft preparation, J.P. and S.S.; writing—review and editing, J.P. and S.S.; visualization, J.P.; supervision, J.P. and S.S.; project administration, J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: https://cloud.icis.pcz.pl/s/zZmiiBA2YqxDprC (accessed on 14 November 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BAN logic	Burrows, Abadi, and Needham logic
CMMTree	Conditional Multiway Mapped Tree
ET	Execution Tree
NSPK protocol	Needham Schroeder Public Key protocol
PC	Pruning condition
PF	Predicate Function
ROM	Random Oracle Model
ROR	Real-Or-Random
RTTI	Run-Time Type Information
SoC	Set of Chains
SVO	Syverson–Van Oorschot logic

References

1. Attkan, A.; Ranga, V. Cyber-physical security for IoT networks: A comprehensive review on traditional, blockchain and artificial intelligence based key-security. *Complex Intell. Syst.* **2022**, *8*, 3559–3591. [CrossRef]
2. AbuAlghanam, O.; Qatawneh, M.; Almobaideen, W.; Saadeh, M. A new hierarchical architecture and protocol for key distribution in the context of IoT-based smart cities. *J. Inf. Secur. Appl.* **2022**, *67*, 103173. [CrossRef]
3. Mo, J.; Hu, Z.; Shen, W. A Provably Secure Three-Factor Authentication Protocol Based on Chebyshev Chaotic Mapping for Wireless Sensor Network. *IEEE Access* **2022**, *10*, 12137–12152. [CrossRef]
4. Kubanek, M.; Bobulski, J.; Karbowiak, L. Intelligent Identity Authentication, Using Face and Behavior Analysis. In Proceedings of the Effectiveness of ICT Ethics—How Do We Help Solve Ethical Problems in the Field of ICT?—ETHICOMP, Turku, Finland, 26–28 July 2022; pp. 42–51.
5. Szymoniak, S. Security protocols analysis including various time parameters. *Math. Biosci. Eng.* **2021**, *18*, 1136–1153. [CrossRef] [PubMed]
6. Szymoniak, S. Amelia—A new security protocol for protection against false links. *Comput. Commun.* **2021**, *179*, 73–81. [CrossRef]
7. Needham, R.; Schroeder, M. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* **1978**, *21*, 993–999. [CrossRef]
8. Denning, D.E.; Sacco, G.M. Timestamps in Key Distribution Protocols. *Commun. ACM* **1981**, *24*, 533–536. [CrossRef]
9. Woo, T.Y.C.; Lam, S.S. A Lesson on Authentication Protocol Design. *SIGOPS Oper. Syst. Rev.* **1994**, *28*, 24–37. [CrossRef]
10. I.-Lung Kao, R.C. An efficient and secure authentication protocol using uncertified keys. *ACM SIGOPS Oper. Syst. Rev.* **1995**, *29*, 14–21.
11. Berguig, Y.; Laassiri, J.; Hanaoui, S. Anonymous and lightweight secure authentication protocol for mobile Agent system. *J. Inf. Secur. Appl.* **2021**, *63*, 103007. [CrossRef]
12. Safkhani, M.; Rostampour, S.; Bendavid, Y.; Sadeghi, S.; Bagheri, N. Improving RFID/IoT-based generalized ultra-lightweight mutual authentication protocols. *J. Inf. Secur. Appl.* **2022**, *67*, 103194. [CrossRef]

13. Szymoniak, S.; Siedlecka-Lamch, O. Securing Meetings in D2D IoT Systems. In Proceedings of the Effectiveness of ICT Ethics—How Do We Help Solve Ethical Problems in the Field of ICT?—ETHICOMP, Turku, Finland, 26–28 July 2022; pp. 30–41.
14. Piatkowski, J. The Conditional Multiway Mapped Tree: Modeling and Analysis of Hierarchical Data Dependencies. *IEEE Access* **2020**, *8*, 74083–74092. [CrossRef]
15. Siedlecka-Lamch, O.; Szymoniak, S.; Kurkowski, M. A Fast Method for Security Protocols Verification. In Proceedings of the Computer Information Systems and Industrial Management—18th International Conference, CISIM 2019, Belgrade, Serbia, 19–21 September 2019; pp. 523–534.
16. Siedlecka-Lamch, O.; Szymoniak, S.; Kurkowski, M.; Fray, I.E. Towards Most Efficient Method for Untimed Security Protocols Verification. In Proceedings of the 24th Pacific Asia Conference on Information Systems, PACIS 2020, Dubai, United Arab Emirates, 22–24 June 2020; Vogel, D.; Shen, K.N., Ling, P.S., Hsu, C., Thong, J.Y.L., Marco, M.D., Limayem, M., Xu, S.X., Eds.; Association for Information Systems: Atlanta, CA, USA, 2020; p. 189.
17. Dai, C.; Xu, Z. A secure three-factor authentication scheme for multi-gateway wireless sensor networks based on elliptic curve cryptography. *Ad. Hoc. Netw.* **2022**, *127*, 102768. [CrossRef]
18. Ali, A.T. Simplified timed attack trees. In Proceedings of the International Conference on Research Challenges in Information Science, Limassol, Cyprus, 11–14 May 2021; Springer: Cham, Switzerland, 2021; pp. 653–660.
19. André, É.; Lime, D.; Ramparison, M.; Stoelinga, M. Parametric analyses of attack-fault trees. *Fundam. Inform.* **2021**, *182*, 69–94. [CrossRef]
20. Siedlecka-Lamch, O. Probabilistic and Timed Analysis of Security Protocols. In Proceedings of the 13th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2020), Seville, Spain, 13–15 May 2019; Herrero, Á., Cambra, C., Urda, D., Sedano, J., Quintián, H., Corchado, E., Eds.; Springer: Cham, Switzerland, 2021; pp. 142–151.
21. Burrows, M.; Abadi, M.; Needham, R.M. A logic of authentication. *Proc. R. Soc. Lond. Math. Phys. Sci.* **1989**, *426*, 233–271.
22. Abdalla, M.; Fouque, P.A.; Pointcheval, D. Password-based authenticated key exchange in the three-party setting. In Proceedings of the International Workshop on Public Key Cryptography, Les Diablerets, Switzerland, 23–26 January 2005; Springer: Berlin, Germany, 2005; pp. 65–84.
23. Xue, K.; Meng, W.; Li, S.; Wei, D.S.; Zhou, H.; Yu, N. A secure and efficient access and handover authentication protocol for Internet of Things in space information networks. *IEEE Internet Things J.* **2019**, *6*, 5485–5499. [CrossRef]
24. Syverson, P.F.; Van Oorschot, P.C. On unifying some cryptographic protocol logics. In Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, 16–18 May 1994; pp. 14–28.
25. Barbosa, M.; Barthe, G.; Bhargavan, K.; Blanchet, B.; Cremers, C.; Liao, K.; Parno, B. SoK: Computer-Aided Cryptography. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 777–795. [CrossRef]
26. Cremers, C.; Fontaine, C.; Jacomme, C. A Logic and an Interactive Prover for the Computational Post-Quantum Security of Protocols. In Proceedings of the S&P 2022—43rd IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 23–25 May 2022.
27. Cortier, V.; Delaune, S.; Dreier, J. Automatic generation of sources lemmas in Tamarin: Towards automatic proofs of security protocols. In *Lecture Notes in Computer Science, Proceedings of the ESORICS 2020—25th European Symposium on Research in Computer Security, Guilford, UK, 14–18 September 2020*; Springer: Cham, Switzerland, 2020; Volume 12309, pp. 3–22. [CrossRef]
28. Basin, D.; Cremers, C.; Dreier, J.; Sasse, R. Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols. *IEEE Secur. Priv.* **2022**, *20*, 24–32. [CrossRef]
29. Blanchet, B.; Cheval, V.; Cortier, V. ProVerif with lemmas, induction, fast subsumption, and much more. In Proceedings of the IEEE Symposium on Security and Privacy (S&P'22), San Francisco, CA, USA, 23–25 May 2022; pp. 205–222.
30. Blanchet, B.; Smyth, B. Automated reasoning for equivalences in the applied pi calculus with barriers. *J. Comput. Secur.* **2018**, *26*, 367–422. [CrossRef]
31. Yao, J.; Xu, C.; Li, D.; Lin, S.; Cao, X. Formal Verification of Security Protocols: ProVerif and Extensions. In Proceedings of the International Conference on Artificial Intelligence and Security, Qinghai, China, 15–20 July 2022; Springer: Cham, Switzerland, 2022; pp. 500–512.
32. Alegria, J.A.H.; Bastarrica, M.C.; Bergel, A. Avispa: A tool for analyzing software process models. *J. Softw. Evol. Process.* **2014**, *26*, 434–450. [CrossRef]
33. Lowe, G. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Inf. Process. Lett.* **1995**, *56*, 131–133. [CrossRef]
34. Lowe, G. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Passau, Germany, 27–29 March 1996; Margaria, T., Steffen, B., Eds.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 147–166.
35. Grosser, A.; Kurkowski, M.; Piatkowski, J.; Szymoniak, S. ProToc—An Universal Language for Security Protocols Specifications. In Proceedings of the ACS, San Francisco, CA, USA, 10–14 August 2014.
36. Satyanarayanan, M. Integrating security in a large distributed system. *ACM Trans. Comput. Syst.* **1989**, *7*, 247–280. [CrossRef]
37. Lowe, G. Some new attacks upon security protocols. In Proceedings of the 9th IEEE Computer Security Foundations Workshop, Haifa, Israel, 7–10 August 1996; pp. 162–169.

38. Carlsen, U. Optimal Privacy and Authentication on a Portable Communications System. *Oper. Syst. Rev.* **1994**, *28*, 16–23. [CrossRef]
39. Burrows, M.; Abadi, M.; Needham, R. A Logic of Authentication. *ACM Trans. Comput. Syst.* **1990**, *8*, 18–36. [CrossRef]
40. Lowe, G. *A Family of Attacks upon Authentication Protocols*; Technical Report; Department of Mathematics and Computer Science, University of Leicester: Leicester, UK, 1997.
41. Lowe, G. Towards a completeness result for model checking of security protocols. *J. Comput. Secur.* **1999**, *7*, 89–146. [CrossRef]
42. Paulson, L.C. Relations between secrets: Two formal analyses of the Yahalom protocol. *J. Comput. Secur.* **2001**, *9*, 197–216. [CrossRef]
43. Tremblay, J.P.; Sorenson, P.G. *An Introduction to Data Structures with Applications*; McGraw-Hill Inc.: Irvine, CA, USA, 1984.
44. Aho, A.V.; Ullman, J.D. *Foundation of Computer Science in C*; W. H. Freeman and Co.: New York, NY, USA, 1994.
45. Shaffer, C. *A Practical Introduction to Data Structures and Algorithm Analysis (C++ Version)*, 3rd ed.; Department of Computer Science Virginia Tech: Blacksburg, VA, USA, 2010.
46. Weiss, M. *Data Structures and Algorithm Analysis in C++*; Addison Wesley: New York, NY, USA, 2014.
47. Kruse, R.L.; Ryba, A.J. *Data Structures and Program Design in C++*; Prentice-Hall, Inc.: Hoboken, NJ, USA, 2000.
48. Barnett, G.; Tongos, L. *Data Structures and Algorithms: Annotated Reference with Examples*; NETSlackers: Pierrefonds, CA, USA, 2008.
49. Mileva, A.; Dimitrova, V.; Kara, O.; Mihaljevic, M.J. Catalog and Illustrative Examples of Lightweight Cryptographic Primitives. In *Security of Ubiquitous Computing Systems*; Springer: Cham, Switzersland, 2021.
50. Longo, R.; Mascia, C.; Meneghetti, A.; Santilli, G.; Tognolini, G. Adaptable Cryptographic Primitives in Blockchains via Smart Contracts. *Cryptography* **2022**, *6*, 32. [CrossRef]
51. Stroustrup, B. *The C++ Programming Language*; Addison Wesley: New York, NY, USA, 2013.
52. Dolev, D.; Yao, A.C. On the Security of Public Key Protocols. In Proceedings of the 22nd Annual Symposium on Foundations of Computer Science—SFCS '81, Washington, DC, USA, 28–30 October 1981; pp. 350–357.
53. Kassem, A.; Lafourcade, P.; Lakhnech, Y.; Mödersheim, S. Multiple Independent Lazy Intruders. In Proceedings of the 1st Workshop on Hot Issues in Security Principles and Trust (HotSpot 2013), Rome, Italy, 17 March 2013.
54. Mödersheim, S.; Nielson, F.; Nielson, H.R. Lazy Mobile Intruders. In *Lecture Notes in Computer Science, Proceedings of the POST, Rome, Italy, 16–24 March 2013*; Basin, D.A., Mitchell, J.C., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7796, pp. 147–166.
55. Available online: https://en.cppreference.com/ (accessed on 1 September 2023).
56. Mödersheim, S.; Vigano, L.; Basin, D. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *J. Comput. Secur.* **2010**, *18*, 575–618. [CrossRef]